Performance Modeling of Parallel Systems

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof.ir. K.F. Wakker,
in het openbaar te verdedigen ten overstaan van een commissie,
door het College van Dekanen aangewezen,
op dinsdag 23 april 1996 te 16.00 uur
door

Arie Jan Cornelis VAN GEMUND

informatica ingenieur
geboren te Eindhoven

Rector Magnificus (voorzitter)	Technische Universiteit Delft
Prof.ir. G.L. Reijns (promotor)	Technische Universiteit Delft
Prof.dr. G. Haring	Universität Wien, Oostenrijk
Prof.dr.ir. L. Dekker	Technische Universiteit Delft
Prof.dr.ir. J. van Katwijk	Technische Universiteit Delft
Prof.dr.ir. H.J. Sips	Universiteit van Amsterdam
Prof.dr. S. Vassiliadis	Technische Universiteit Delft
Dr. E.M.R.M. Paalvast	Bakkenist Management Consultants B.V., Diemen

Published and distributed by:

Delft University Press
Stevinweg 1
2628 CN Delft
The Netherlands

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Gemund, Arie Jan Cornelis van

Performance modeling of parallel systems /
Arie Jan Cornelis van Gemund. - [S.1. : s.n.]. - I11.
Proefschrift Technische Universiteit Delft. -
Met lit. opg. - Met samenvatting in het Nederlands.
ISBN 90-407-1326-X
Trefw.: prestatie-analyse / computersystemen /
computernetwerken.

To Karen

Contents

Acknowledgments

Perhaps even more than parallel computing itself, performance modeling of parallel computer systems is a relatively new and challenging field. This thesis is an account of my research in this area in which I have had the great pleasure to study existing work, develop an alternative modeling technique, and, last but not least, study its adequacy.

First of all, I am deeply grateful to my advisor Professor Gerard L. Reijns for giving me the opportunity to do this research while joining up with his group as an assistant professor in 1992. The four years that I worked with him have been a most enjoyable start of my academic career.

Although PAMELA came to birth after I was given the above opportunity, many of the foundations were laid earlier during my involvement in the *ParTool* project, a nationally funded parallel processing research project, initiated by Henk Sips, Edwin Paalvast, and Maarten van Steen. Thanks are due to Henk Sips, who got me involved in parallel computing in the first place, Maarten van Steen, with whom I have had many stimulating discussions on concurrency (and planning), and, last but not least, Edwin Paalvast, my first comrade-in-research, later on head of our research group, who has always supported me during my research. It pleases me that the four of us are still in touch.

I consider it a privilege to have a graduation committee, apart from Gerard Reijns and the Rector Magnificus consisting of the distinguished members Len Dekker, Günter Haring, Jan van Katwijk, Edwin Paalvast, Henk Sips, and Stamatis Vassiliadis. I gratefully acknowledge their efforts.

It is a pleasure to express my gratitude to former graduate student and room mate Henk Jonkers, with whom I had many stimulating discussions on the subject of parallel systems performance modeling, both at Delft University as well as during our enjoyable conference trips to Grenoble and Maui. Thanks are also due to former under-graduate students Solechoel Arifin, Azzedine Benchellal, Marcel Bontekoe, M'hammed Farahi, Januar Himantono, Alexander van Lomwel, and Arun Persad for providing valuable feedback on the application of the modeling methodology. In particular, credit goes to Marc Nijweide, Ronald Pulleman, and Mark Roest who have actively contributed in the tool development. Also the pleasant association with my (former) colleagues André Bos, Hai-Xiang Lin, Teus Vogel, and Pieter van der Wolf is gratefully acknowledged.

The number of people from whom I learned so much through the years, is simply too large to permit a listing. Many of them are mentioned in the bibliography. Without their knowledge I would not have come this far.

This especially applies to my dear wife Karen. Of all the valuable things I have learned, she has definitively taught me most.

Arjan van Gemund

Summary

Performance modeling plays a fundamental role in the design of computer systems. This applies especially to parallel systems where high performance is of key interest. While performance modeling of sequential computer systems already poses a number of important problems, the problem involved with performance modeling of parallel systems is even more fundamental. This is essentially due to the prominent role synchronization plays in parallel computing. Apart from the inevitable overhead introduced by parallelization, especially for badly designed systems the additional synchronization delays can easily cause a tremendous loss of performance.

In parallel systems synchronization can be distinguished into a static and a dynamic form. The static form, called condition synchronization, corresponds to precedence relations between tasks that are predetermined as a result of the parallelization. The alternative form, called mutual exclusion, applies to the dynamic resolvement of precedence order during contention for a limited number of software or hardware resources. While the performance analysis of condition synchronization entails quite some computational costs, an accurate analysis of mutual exclusion can be extremely computation-intensive due to the inherent non-determinism involved.

There exists a wide variety of approaches to the performance modeling of parallel systems, each representing a different trade-off between the accuracy of the analysis and the computational cost involved. On the one hand, there exist performance modeling approaches based on representation formalisms such as (stochastic) Petri nets that aim for high modeling power such that every synchronization structure can be described with a high level of accuracy. The computational cost of the associated state space analysis, however, is exponential in the problem size. On the other hand, there are performance modeling approaches, based on simple task graph representations, that only account for condition synchronization. In turn, they entail only a linear analysis complexity. However, as the performance loss due to mutual exclusion is ignored the accuracy of such methods is inherently limited.

In this dissertation a new approach to the performance modeling of parallel systems is described. Similar to some of the existing techniques, the approach is primarily aimed to support the initial phases in the design of parallel systems where the emphasis is on extremely low solution cost, rather than on high accuracy. Unlike current low-cost approaches, however, a minimum degree of accuracy is sustained by introducing an approximate analysis of mutual exclusion, next to condition synchronization, without sacrificing the low solution cost. Furthermore, the analysis technique yields explicit, analytic performance models, such that program and machine parameters are symbolically retained in the resulting performance model. Apart from providing low solution cost, in this way, parameter studies or possibly automated parameter optimization procedures can be conducted without remodeling effort. Next to the low solution cost, this feature is essential to an optimal design efficiency.

The approach is based on the use of a new simulation formalism, called Pamela (PerformAnce ModEling LAnguage). Although the language features synchronization constructs necessary to avoid *a priori* limitations with regard to modeling accuracy, Pamela also features structured operators, especially for the description of mutual exclusion. Used within a material-oriented modeling paradigm important information concerning the synchronization patterns involved can be retained in the model. As a consequence, apart from simulation, Pamela can be used as a source language for an automatic, compile-time analysis technique that yields an explicit, analytic performance model. The model approximately accounts for the performance loss due to mutual exclusion in terms of a lower bound on the execution time. The novelty of the approach is that it integrates a language approach, a material-oriented modeling paradigm, and the compile-time analysis method within one methodology.

While Chapter 1 presents a problem analysis and formulates the goals of the research, Chapter 2 presents a survey of related work on performance modeling of parallel systems in order to put the approach into perspective. The work discussed includes approaches based on representation formalisms such as task graphs, queuing networks, Petri nets, simulation languages, and process algebras.

Chapter 3 presents Pamela, essentially comprising the concurrent modeling language and the underlying analysis technique. It is shown that the explicit, and highly structured way in which the material-oriented modeling method describes condition synchronization and mutual exclusion, offers great advantages with respect to model analyzability. The analysis technique is described as well as a number of typical examples.

Chapter 4 describes the principles underlying the application of Pamela to parallel computer systems modeling. The methodology towards modeling shared-memory and distributed-memory programs and machines are presented through a large variety of examples. It is shown that the restricted modeling formalism allows to capture the dominant performance aspects that are relevant in the context of the approximate analysis.

Chapter 5 presents various applications of the Pamela methodology. The case studies address performance compilation, showing how Pamela models are compiled into analytic models, synthetic applications on a distributed-memory system, demonstrating the accuracy of the modeling approach compared to actual measurements, a comparison between the analytic technique and simulation, and, a case study showing how Pamela is applied to program optimization.

Chapter 6 investigates the accuracy of the analysis method. By studying the relation between the analytic estimate and the simulation result it is shown that the approximation based on a lower bound provides a good estimator. In addition it is shown that for any system in which resource usage is random, the average estimation error due to contention effects is limited to 50 % worst case regardless of the system parameters involved. In view of the highly parameterized, low-cost models that are compiled, this prediction robustness forms an essential justification of the approach.

Finally, Chapter 7 recapitulates the work, and presents a number of recommendations for future improvements.

Chapter 1

Introduction

This dissertation presents a methodology to predict the performance of parallel computer systems. In this definition, a parallel computer *system* constitutes an imperative parallel *program* (application) and the parallel von Neumann *machine* on which it is executed. With respect to parallel programs we restrict ourselves to *explicit* parallel programs that are *native* to the machine. Thus we avoid the problem of dealing with the semantical gap between implicitly parallel, possibly declarative, problem descriptions and their explicitly parallel, imperative implementations, as well as the complex optimization properties of compilers in general. With respect to parallel machines we consider any computer system that involves some form of concurrency. As such, the methodology also applies to sequential computers, that, while providing a sequential programming model, exhibit parallelism at the hardware level. The focus of this work, however, are shared-memory and distributed-memory (vector) computers that provide explicit concurrency at the programming level. Although we will discuss performance modeling in the context of parallel (and distributed[1]) computer systems its scope is much wider. In fact, any concurrent system like traffic systems, production plants, or office environments are essentially a collection of concurrent (man or machine-executed) *processes*, jointly involved in *synchronization* either due to work partitioning or due to the use of common *resources*. In this respect, a (parallel) computer is just another (data) processing system.

In the design of concurrent systems, performance engineering is often conducted as an afterthought. Often systems (prototypes) are already built and functionally tested before their performance is evaluated, in many cases with disappointing results. This especially applies to parallel computing where the performance awareness of program (and, to a less extent machine) designers is minimal, whereas the performance implications of coding decisions can be profound. As a result of this, there is a growing interest in performance prediction techniques that provide the programmer some feedback in the design (or selection) process as illustrated by Fig. 1.1. Given a computational problem, the application design process starts with some initial choice (in the figure denoted "synthesis") of a program (algorithm) and machine (architecture), partially characterized by various program and machine parameters (denoted π_i and μ_i, respectively). A program parameter might be the problem size or the way in which the problem is mapped onto logical processors.

[1]In this dissertation a distributed computer system is treated as a parallel system, the distinction being a matter of architecture.

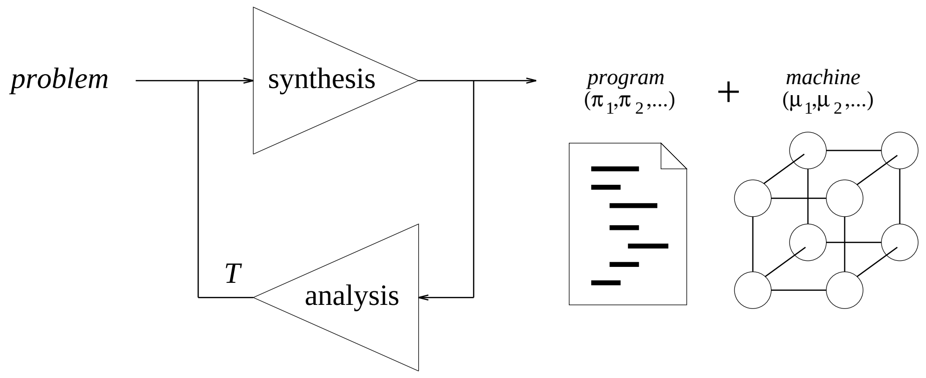


Figure 1.1: Application design loop

An obvious machine parameter is the number of physical processors. In order to derive optimum performance (typically, minimum execution time), the design is analyzed (in the figure denoted "analysis") in order to obtain performance feedback on the parameter choices (in the figure denoted T, the estimated execution time of the application).

As in any feedback system, the ultimate design result is determined by the prediction accuracy of the analysis. In contrast to sequential systems, performance prediction of parallel systems is far from trivial given the large amount of academic work spent in this area. In particular, there exists a large trade-off between solution *cost* and solution *accuracy*, as can be seen from the large variety of approaches that have been undertaken. Typically, performance feedback is organized in terms of a prediction hierarchy, providing different types of performance feedback, each with a different quality and cost. At the low end detailed techniques such as simulation are used that provide realistic predictions, yet at high computational cost. At the high end, crude techniques such as compile-time prediction provide much faster performance feedback, however, at the inherent expense of prediction accuracy. Despite this accuracy sacrifice, this alternative is quite attractive during the initial phase where the design space is still large.

As shown in the above figure, performance modeling aims to map a (parallel) program in conjunction with a (parallel) machine onto some computable *model*, ranging from a simple expression to some complex algorithm, either which can be evaluated numerically. For instance, consider the sequential computation of the 3rd-order polynomial

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 \tag{1.1}$$

For the purpose of the example we will assume a traditional computer architecture capable of executing floating point multiply and add operations that take τ_m time, and τ_a time, respectively (neglecting other instructions). Then the *performance model* of the polynomial computation is given by the simple algebraic expression[2]

$$T = 5\tau_m + 3\tau_a \tag{1.2}$$

[2]For the purpose of the example this analysis is based on a naive algorithm comprising a sequence of 5 multiplications and 3 addition operations. Thus, Horner's scheme is not applied, the reason being that a parallelization of the naive algorithm will be considered later on.

As this model is analytic, subsequent analysis immediately reveals the (linear) dependency of the above design's performance on the parameters of the floating point unit. Of course, deriving a performance model of a complicated parallel computation is by far less trivial.

1.1 The Challenge

As in any area there exist numerous approaches, yielding performance models that are specific, either for the program, or machine, or both. In the worst case, all system parameters are (numerically) hard-wired into the model. Clearly, an ideal performance modeling technique would map a program and machine combination into a *symbolic* performance model where all the system parameters of interest are still *retained*, rather than being numerically instantiated prior to the modeling effort. Figure 1.2 shows the performance

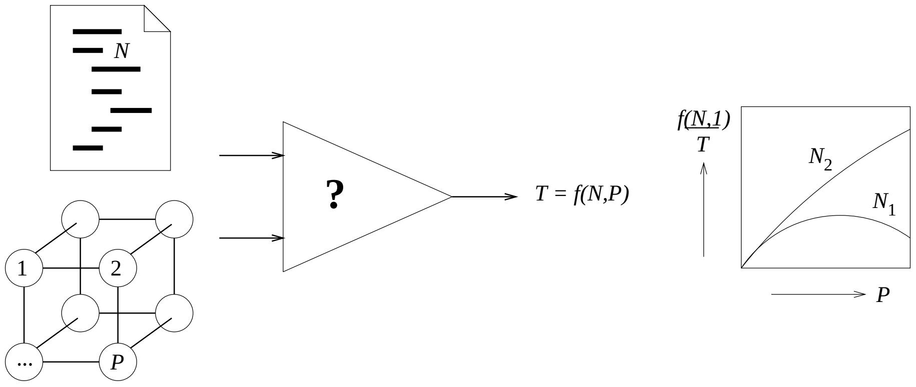


Figure 1.2: Performance modeling process

modeling process involving (only) two parallel application parameters, i.e., the program parameter N (e.g., problem size) and the machine parameter P (number of processors). The performance modeling process (in the figure denoted "?") yields an estimate of the execution time T that is a function f of both parameters (of course, f will also reflect the other properties of the program and machine). Once the symbolic model is derived, various parameter studies may be conducted (*without* remodeling effort) in order to analyze the application's performance characteristics, possibly using standard mathematical tools. The parameter study is illustrated in the figure by the graph in which the *speedup*[3] is plotted as a function of P for two values of N.

Of course, in terms of our definition mentioned earlier, a symbolic performance model is any formal description that computes a number (T). Consequently, a fully parameterized *simulation* model would also qualify as a symbolic performance model. However, the *ideal* performance model has the *lowest* solution complexity possible without loss of prediction accuracy. Although a simulation model may be accurate, its solution (i.e., mean

[3]The speedup is the relative gain in execution speed when more processors are added. Usually, speedup is defined as the ratio between uniprocessor execution time ($T = f(N, 1)$ in the figure) and parallel execution time ($T = f(N, P)$). Ideally, the speedup is linear in P. In practice, however, speedup is much lower. Adding too many processors eventually results in *slow down*, as illustrated by the plot.

value) requires many simulation runs because of the model's inherent non-determinism. Preferably, the model should be *analytic* (computes a deterministic time domain result) and should have the form of an *explicit* equation (such as Eq. (1.2)). Apart from its low solution cost, standard calculus can be used that provides a well-established framework for the use of symbolic techniques rather than just numeric evaluation. Examples are the use of model *reduction* (thus decreasing evaluation complexity) and/or gradient analysis (cf. Eq. (1.2)). An interesting possibility offered by symbolic models is the application of optimization techniques such as linear programming (provided the model, or target function, is linear). In terms of Fig. 1.1, the analysis is strongly coupled with the synthesis (parameter optimization) component. In fact, this approach exemplifies the ultimate purpose of performance modeling, namely, to transform the design problem in the parallel computation domain into a "regular" optimization problem in the mathematical domain. (This problem reformulation in terms of applied mathematics is essentially what makes computer science to become a mature exact science such as physics.) Within our symbolic approach towards performance modeling, the actual challenge lies in developing a technique that strikes an optimal balance between modeling accuracy and solution cost. Before we develop our approach in more detail, we first analyze the performance aspects involved with parallel computing.

1.2 Parallel Computing

In order to describe the specific aspects of parallel systems performance we discuss a simple example in which we consider a parallelization of the 3rd-order polynomial computation, presented earlier. The example is also used to introduce some basic terminology that is used throughout the dissertation.

Recall the 3rd-order polynomial, given by Eq. (1.1). In order to assess the opportunities for parallelism, we consider the (directed, acyclic) *task graph* representation of Eq. (1.1), shown in Fig. 1.3. Like in the analysis of the sequential case, only 5 multiplication tasks (t_1, t_2, t_3, t_4, t_6) and 3 addition tasks (t_5, t_7, t_8) are distinguished. The execution model of the task graph is based on full task concurrency only restricted by *condition synchronization* [8], as designated by the precedence arcs. The condition upon each task fires is that *all* predecessor tasks must have finished (i.e., all *data dependencies* must have been obeyed). Let each (multiply and add) task take 1 unit time. If an unbounded number of processors were available the execution time would be $T = 4$ as can be seen from the execution trace shown in Fig 1.4. In this case, execution time is only limited by the inherent problem characteristics and is determined by the *critical path* in the graph. In practice, however, the *potential* parallelism in the problem can not always be realized due to an unsufficient number of processing *resources*. For instance, let $P = 2$. One of the optimal *schedules* is shown in Fig. 1.5. Due to the limited number of resources, the *actual* parallelism is reduced, resulting in $T = 5$. Note that the limited number of resources induces additional condition synchronizations on the task graph, forcing tasks, originally concurrent, to be serialized.

Thus far, in the analysis of the computation's properties we only considered the floating point operations, which implies an *abstract* parallel machine architecture simply comprising a collection of P (floating point) processors. As a first-order approximation, restricting

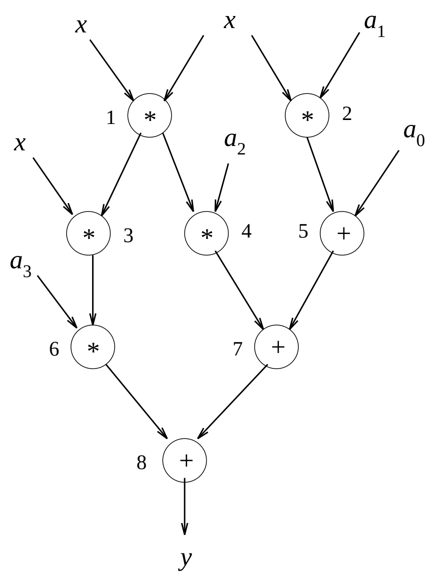


Figure 1.3: Task graph representation of Eq. (1.1)

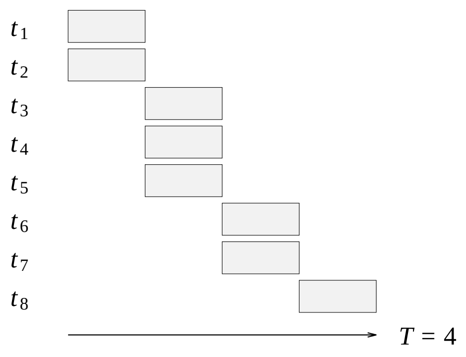


Figure 1.4: Inherent parallelism of Fig. 1.3

the analysis to the algorithm's most dominant operation is quite common as illustrated by the large body of work in PRAM-based [45] complexity analysis. In a more detailed performance analysis, however, the fact that the necessary *data transfers* may also take considerable time, must be accounted for. At this point, we assume the existence of P processors that are capable of moving data as well as executing floating point operations on that data. For the purpose of this example, we just consider a simple *shared-memory* architecture, comprising $P = 2$ processors, connected to a shared memory through some idealized interconnection network. Throughout the example we will assume the schedule as given in Fig. 1.5. Apart from executing multiply and add instructions, each processor can *load* from shared memory and *store* to shared-memory. We also assume the presence of *local* registers that have a much lower access time than the *global* shared memory. We will explicitly account for this *memory hierarchy*, by neglecting local data moves and only accounting for the global loads and stores. Consequently, we introduce two extra task types, i.e., a *load task* and a *store task*. For instance, the intermediate result of task t_1 is

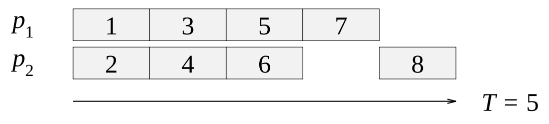


Figure 1.5: Performance loss for $P = 2$

directly available for task t_3, since t_3 is mapped onto the same processor. However, with regard to t_4, the intermediate result must first be stored by p_1 to shared memory, and loaded by p_2, before t_4 may commence. The introduction of the load and store tasks is represented by the task graph given in Fig. 1.6 where the mapping of each load and store task directly follows from the original mapping of the floating point operations (Fig 1.5). For ease of interpretation, each task is annotated by brackets delimiting the (processing) resource it is mapped onto. Let both a load and store task represent a work load of half a unit time[4]. The corresponding schedule is given in Fig. 1.7, that shows the dramatic impact of data transfer overhead on performance ($T = 9.5$ compared to $T = 5$ in Fig. 1.5). For $P = 1$ we would have $T = 11$ (5 loads, 8 flops, and 1 store). Thus, even for a modest value of P hardly any speedup is obtained, a result that is typical for many practical situations.

Thus far, compared to the potential task graph performance, we have encountered two sources of performance loss, i.e., the introduction of additional condition synchronization as a result of the static schedule (insufficient number of processors), and the introduction of additional (data transfer) work load as a result of the underlying machine. Apart from performance loss due to condition synchronization, we will now introduce a second form of synchronization loss, that results from *mutual exclusion* [8]. Again, consider the polynomial example. We will now account for the fact that, in reality, a shared memory location can not be accessed *simultaneously* by more than one processor. Let us assume, that the shared memory system, in fact, comprises only one physical memory module with one access port. Then all memory locations used to load and store (intermediate) results can only be accessed sequentially. Thus, the shared memory is to be regarded as *one* memory resource, providing exclusive service to each processor. Unlike the case of the (limited) processing resources, we can not resolve the problem by a static schedule as the necessary condition synchronizations would typically involve shared memory as well. Instead, a typical implementation of the shared-memory is by *dynamically* scheduling *contending* load and store requests at run-time by hardware. If the memory resource is occupied, all other requesting processors face a *memory conflict* and are *blocked* for the duration of the current memory access (hence, the performance loss). Due to the dynamic contention model, this phenomenon is usually referred to as *resource contention*.

As memory contention is handled at run-time, the static schedule of Fig. 1.7 still applies. However, the actual execution *trace* (or dynamic schedule) will now correspond to Fig. 1.8, in which each load and store task are serialized according to how the contention is resolved (i.e., the particular conflict arbitration). Due to the additional mutual exclusion the execution time has increased even further to $T = 11.5$. Like condition

[4]Thus, global memory latency is assumed to be quite considerable compared to floating point operations. This is not unrealistic for many parallel systems.

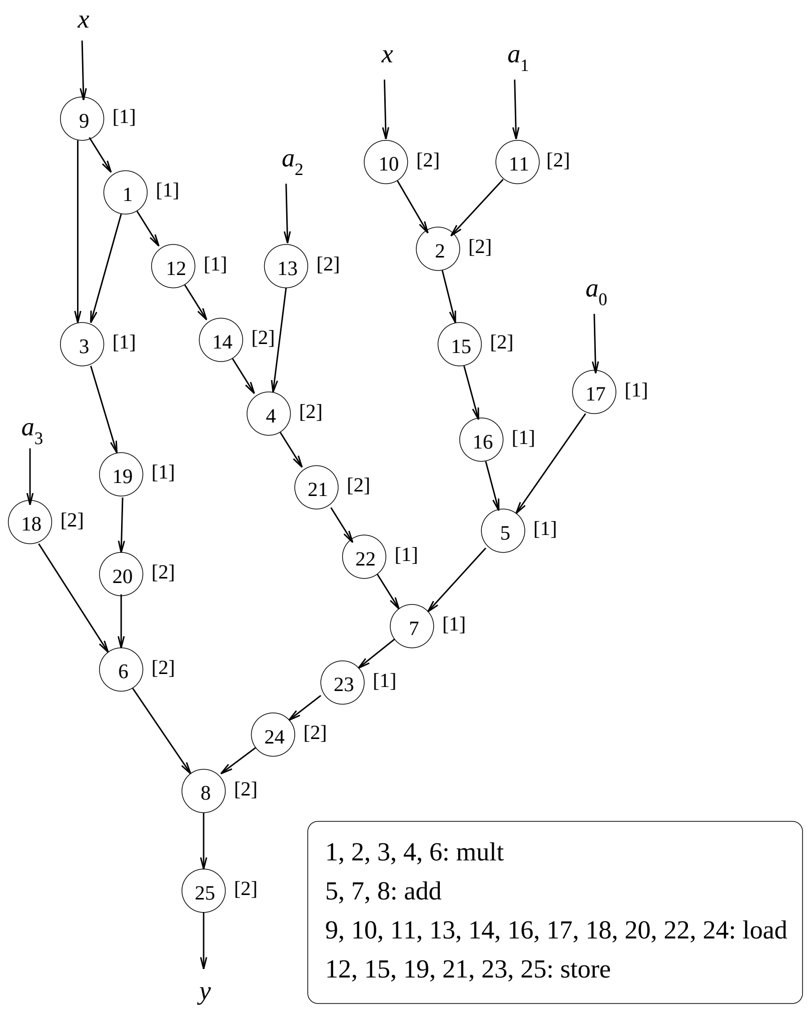


Figure 1.6: Annotated task graph including load/store tasks

synchronization, mutual exclusion ("contention") can easily dominate performance loss. For instance, consider a parallel computation in which a large number of processors are involved, all addressing a single memory resource. Due to the request serialization instead of parallelization, the performance result may be quite dramatic as is shown later on.

In general, the problems involved in performance analysis are related to four aspects,

- Condition synchronization
  This form of synchronization is induced either by data dependencies within parallel computations or by resource limitations, either at program level or machine level. In contrast to mutual exclusion, the precedence relation enforced by condition synchronization is static. Consequently, the analysis of the associated delays can be performed simply by determining the longest path in the task graph which, for deterministic task times, has polynomial solution complexity. Note that, although the task precedence relation is static, the actual tasks to which the synchronization applies may not be determined until run-time. Nevertheless, the same analysis technique can be used.

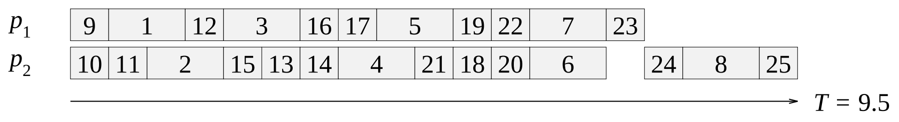

p_1 9 1 12 3 16 17 5 19 22 7 23
p_2 10 11 2 15 13 14 4 21 18 20 6 24 8 25
T = 9.5

Figure 1.7: Performance loss due to memory work load

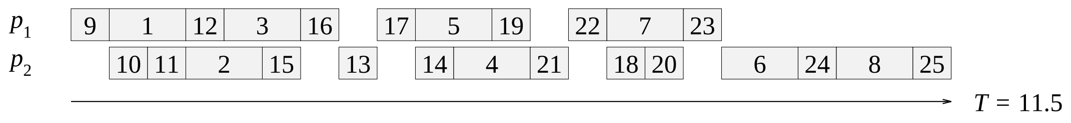

p_1 9 1 12 3 16 17 5 19 22 7 23
p_2 10 11 2 15 13 14 4 21 18 20 6 24 8 25
T = 11.5

Figure 1.8: Performance loss due to memory contention ($P = 2$)

- Mutual exclusion
  As shown in the example, mutual exclusion is often associated with contention for resources, either at machine level (e.g., CPU, memory) or program level (e.g., critical sections). Mutual exclusion can be regarded as a dynamic form of condition synchronization in which the actual precedence relation is not determined until run-time. Due to the fact that conflict resolution is typically approached as being non-deterministic[5] the analysis essentially involves considering all the execution traces possible which potentially implies exponential solution complexity. Because of this, the analysis of mutual exclusion poses a far greater challenge than the analysis of condition synchronization.

- Conditional control flow
  While conditional control flow is typically associated with program-level branching, machine-level examples include caching and communication routing. As exemplified by the Halting Problem, the undecidability that arises with conditional control flow is already a fundamental problem in sequential programs. There has been a long-standing interest in performance modeling of programs in which branches that are not compile-time deducible are modeled in terms of branching probabilities, typically based on auxiliary profile information on some representative data set (e.g., see [126, 136, 153]). Unlike both synchronization factors, the problems posed by conditional control flow are not fundamentally different for parallel systems, nor are the solutions (e.g., see [13, 41, 133]).

- Basic calibration
  Essential for any performance model is the work load calibration of its basic model components (e.g., the basic instruction timings in the previous example). Although at the basic component level the work loads may be largely deterministic (e.g., fixed number of clock cycles), for models in which the basic components are defined at a higher (aggregate) level, the work loads are often expressed using stochastic

[5]Either by definition or as a result of modeling abstraction. For instance, an arbitration that is essentially deterministic (e.g., round robin) at clock cycle level, is often modeled as non-deterministic at higher abstraction level.

variables to account for the non-determinism at lower level induced by conditional control flow and/or both forms of synchronization. Similar to mutual exclusion, the introduction of non-determinism in combination with condition synchronization introduces a potentially exponential analysis complexity. An equally important issue related to aggregate components is the determination of the parameters themselves (e.g., mean and variance). Because of the problems just mentioned the parameters are typically measured, rather than computed in terms of their constituent parts.

While the latter two performance aspects apply to both sequential and parallel systems, the specific challenge associated with performance modeling of parallel systems is the analysis of both forms of synchronization delays.

1.3 Approach

As stated earlier, our major aim is to develop a methodology to derive performance models with a high level of parametrization. As also mentioned, however, there are various representation formalisms in terms of which the model can be described, such as a simulation model or a low-cost explicit analytical model. Consequently, a central theme in the development of a methodology is an investigation of the trade-off between analysis accuracy and cost. Although the accuracy needed will depend on the application context, especially in the case of a parametric model, its accuracy must be *sustained* across the *entire* parameter space as the design process may involve many (possibly erratic) parameter settings.

As discussed in the previous section, the challenge that is specific to performance modeling of *parallel* systems is the analysis of synchronization. The other two factors are not specific to parallel systems, neither do they present fundamentally different complications compared to the sequential domain (although they pose a formidable challenge in their own right). Hence, in our performance modeling approach we focus on the analysis of *synchronization delays* in terms of the accuracy/cost trade-off mentioned earlier. With respect to their performance impact, condition synchronization and mutual exclusion are equally important. Which one actually dominates performance depends on the particular system. Although in the polynomial example demonstrates the impact of condition synchronization (critical path, the sequential task schedule per processor), mutual exclusion can be just as damaging. This applies especially to simple analytic techniques that are based on a simple critical path analysis only. For example, consider a parallelization of the following algorithm

$$\forall i \in \{1, \ldots, N\} : y_i = f(x_i)$$

in which some computation f is applied to each of the N vector elements x_i, the result being stored in y_i. Again, we assume an abstract, P processor shared-memory machine architecture similar to the one discussed earlier. For each element this implies executing a shared-memory load (x_i, taking τ_m units time), executing f (taking τ_f units time), and executing a shared-memory store (y_i, taking τ_m units time). Figure 1.9 shows the speedup S for a simple data parallel scheme based on a regular block decomposition where each processor is responsible for processing at most $\lceil N/P \rceil$ elements ($N = 100$, $\tau_f = 10\tau_m$).

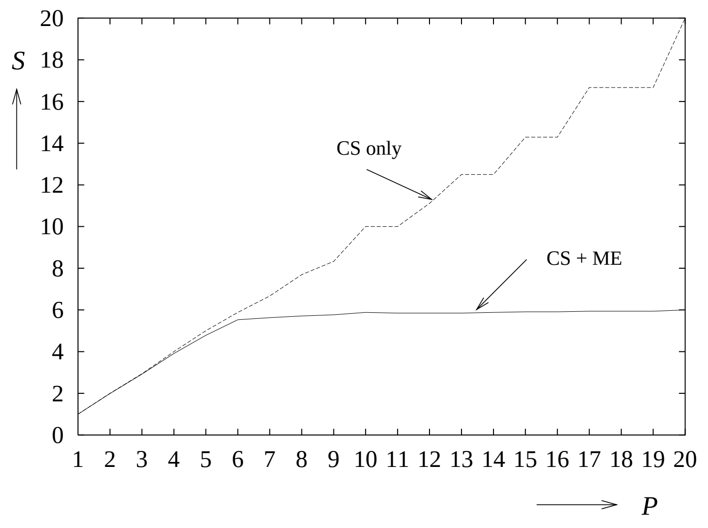


Figure 1.9: The impact of mutual exclusion (ME) on performance

The figure shows two plots (obtained through simulation). The upper plot ("CS only"). denotes the speedup based on ignoring the effect of memory contention and accounting for condition synchronization (CS) only (the discontinuities are caused by the load imbalance in cases when P does not divide N). The lower plot ("CS + ME") shows the "actual" speedup where the effect of mutual exclusion (ME) is included. For large P memory contention delay (queuing) dominates performance. Thus, any prediction technique that ignores the effects of mutual exclusion (such as conventional critical path techniques) may seriously under-estimate the execution time by orders of magnitude.

As mentioned earlier, mutual exclusion introduces a major analysis complication compared to condition synchronization as a result of the inherent non-determinism involved. Therefore, an investigation of the trade-off between analysis accuracy and analysis cost specifically applies to the analysis of mutual exclusion. From this perspective we approach the problem of symbolic performance modeling of parallel systems.

We present a novel performance modeling formalism, called PAMELA (PerformAnce ModEling LAnguage) that serves as a vehicle to express our approach. The formalism has basically two purposes. First, it serves as a concurrent performance simulation language, thus allowing a model to be described *dynamically* without introducing *a priori* limitations with regard to modeling accuracy. At the same time, however, it serves as a source language for a second, *static* performance modeling technique that yields an analytic (compile-time) model. More specifically, the nature of our contribution can be characterized as follows.

- description formalism.
  PAMELA presents a new departure for performance modeling that combines a number of existing modeling techniques in terms of one framework. Due to the use of highly *structured* language operations to describe synchronization important information on the problem's structure with respect to both types of synchronization can be retained in the model.

- modeling methodology.
  In contrast to usual practice, the approach to modeling parallel systems is *material-oriented* [89], in which system components are modeled as subroutines rather than as processes. Combined with the structured synchronization operators, mentioned earlier, a model description is derived that is amenable to a low-cost analytic solution technique.

- analysis technique.
  The above choice of modeling formalism and paradigm allows for the application of a low-cost static technique enabling Pamela models to be automatically *compiled* into *analytic* models. Unlike traditional approaches to static analysis, this novel technique approximately accounts for resource contention. The analytical model approximates the simulation result in terms of a lower bound.

The philosophy behind our contribution is to optimize the trade-off between *modeling power* and the yield in terms of *analysis cost* by a careful choice of *constraints* with respect to the modeling paradigm. The novelty of our approach is that it uniquely integrates and extends a number of existing concepts (performance simulation, material-oriented program/machine modeling, compile-time analysis) within one methodology. The underlying thesis is that the vast majority of parallel computer systems can be expressed in terms of our structured formalism, and that the accuracy of the generated analytical models is acceptable across the entire parameter range. In this dissertation we will substantiate these claims.

1.4 Outline

Before presenting the Pamela methodology, in Chapter 2 we present a survey of related work on performance modeling of parallel systems in order to put our approach into perspective. The work discussed includes approaches based on representation formalisms such as task graphs, queuing networks, Petri nets, simulation languages, and process algebras. In the survey we introduce a new categorization scheme in order to compare the various approaches in terms of one framework.

In Chapter 3 we present our performance modeling formalism, essentially comprising the concurrent modeling language and the underlying analysis technique. It is shown that the explicit, and highly structured way in which the material-oriented modeling method expresses condition synchronization as well as mutual exclusion offers great advantages with respect to model analyzability. The analysis technique is described as well as a number of typical examples.

While Chapter 3 presents Pamela from the perspective of concurrent models in general, in Chapter 4 we present the application of Pamela to parallel *computer* systems modeling. The principles of the modeling methodology towards shared-memory and distributed-memory programs and machines are described through a large variety of examples. It is shown that the use of our restricted modeling formalism still allows us to capture the dominant performance aspects that are relevant in the context of our approximate analysis.

In Chapter 5 we present a number of case studies that demonstrate various applications of the PAMELA methodology. The subjects addressed include performance compilation, showing how PAMELA models are compiled into analytic models, a macro data flow application on a distributed-memory system, demonstrating the accuracy of the modeling approach compared to actual measurements, a discussion on the relation between our analytic technique and simulation, and, last but not least, "optimization modeling", showing how the PAMELA calculus is applied to program optimization, one of the ultimate applications of the methodology.

After this presentation of the utility of the methodology, in Chapter 6 we revisit the analysis technique by explicitly studying the *accuracy* of the analysis method compared to simulation. By studying the relation between our analytic estimate and the simulation result it is shown why the approximation based on a lower bound provides a good estimator. In addition it is shown that for any system in which resource usage is random, the average estimation error due to contention effects is limited to 50 % worst case, throughout the entire range of the model parameters.

In Chapter 7 we summarize our work, and present a number of recommendations for future improvement.

Chapter 2

Performance Modeling

2.1 Introduction

In this chapter we discuss the main approaches to performance modeling of parallel systems, based on representation formalisms such as task graphs, queuing networks, stochastic Petri nets, stochastic process algebras, and simulation languages. Apart from being an account in its own right of the many and very inspiring concepts that have been put forward in this area, it provides the background needed in order to present the rationale for our methodology. For the purpose of comparison, we will introduce a taxonomic framework in terms of which each approach will be described. In this survey we assume a basic understanding of the techniques and formalisms. In cases where the techniques overlap with our approach a more detailed description is given.

Formally, a performance modeling approach deals with evaluating a concurrent system $S \in \mathcal{S}$ comprising a program and machine according to

$$\mathcal{S} = \mathcal{P} \times \mathcal{M}$$

that eventually leads to a numeric result. The system is modeled in terms of some representation formalism $R \in \mathcal{R}$. The set of representation formalism includes deterministic graphs (DG, ordinary task graphs with deterministic time delays), stochastic graphs (SG, task graphs with stochastic time delays), queuing networks (QN), (stochastic) Petri nets (PN), Markov chains (MC), but also analytical models, generally expressed in terms of a system of equations[1] (SE). Thus

$$\mathcal{R} \subset \{DG, SG, QN, PN, MC, \ldots, SE\}$$

Since we are primarily interested in execution time we will denote the set of possible performance results by $T \in \mathcal{T}$ that include temporal representations like a simple deterministic scalar, a full distribution function, or just a mean-variance tuple.

In most performance modeling approaches a number of intermediate, symbolic transformations can be distinguished between the original system and the eventual, numeric performance result. The choice for some intermediate stage is characteristic for a specific

[1]This includes implicit systems, such as an MVA recurrence or a (Markov) matrix equation, but also explicit models, such as a linear pipeline model (each explained later on). Each analytical modeling technique eventually yields an SE representation.

analysis technique. Note that each intermediate transformation may involve a further abstraction from the original system. In general, a performance modeling approach can be characterized in terms of

$$\begin{tikzcd} S \arrow[r] \arrow[d] & R^{(1)} \arrow[r] \arrow[d] & \ldots \arrow[r] & R^{(n)} \arrow[d] \\ T & T^{(1)} & & T^{(n)} \end{tikzcd}$$

in which each $\rightarrow$ represents an intermediate transformation and in which each $\downarrow$ stands for the ultimate numeric evaluation. Thus the first $\downarrow$ corresponds to actual execution timing. The other evaluations yield predictions $T^{(i)}$ that may become less accurate with increasing superscript.

The terms modeling and analysis are often used loosely. Modeling is the process of transforming a system into a performance model. This model can be an analytical time expression (SE) or a timed Petri net (PN). Analyzing a model may yield another model (with higher superscript) and the ultimate numeric evaluation (e.g., simulation result). Thus the steps that can be called modeling cover $S \rightarrow \ldots \rightarrow R^{(n)}$ while analysis covers the steps $R^{(1)} \rightarrow \ldots \rightarrow T^{(n)}$. In our terminology modeling refers to the step $S \rightarrow R^{(1)}$. Subsequent transformations, possibly including the ultimate numerical evaluation will be called (model) analysis.

As an example of the use of this reference framework, we demonstrate the procedure followed in static analysis, a compile-time prediction technique based on critical path analysis of a deterministic task graph representation of the parallel computation. The reason to choose this particular approach is because of the fact that our approach originates from this technique. Recall the polynomial

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3$$

Let S denote the parallel computation. The first step is the transformation from S to a task graph representation $R^{(1)} = G \in DG$. For the ease of this discussion we only consider an ideal abstract machine (i.e., unlimited number of processors, no work loads due to shared-memory load/stores, nor memory contention). Consequently G is shown in Fig 2.1. in which each task g_i represents a deterministic time delay (hence, the deterministic graph) given by τ_i (either τ_m or τ_a). The domain in terms of which G is expressed is still a concurrent execution domain, i.e., G expresses a concurrent process in terms of time delays and condition synchronizations. Consequently, the execution semantics are based on the notion of a concurrent machine in which evaluation of G (yielding $T^{(1)}$) corresponds to performance simulation in which each task execution and synchronization is executed by a simulator (of course, a practical performance simulation approach would involve more factors than covered by just a static task graph representation, but that is beside the point).

In static analysis, however, a next transformation $R^{(1)} \rightarrow R^{(2)}$ is performed from the concurrent execution domain to the time domain, i.e., the domain of real numbers that represent execution time. The transformation is based on following the interpretation of the execution semantics of $R^{(1)}$. Let r_i denote the time at which task g_i finishes (thus, $T^{(1)} = r_8$). Realizing that a task may only commence once *all* predecessors have finished,

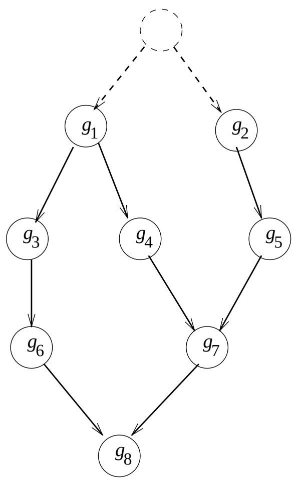


Figure 2.1: Task computation graph of polynomial example

we directly obtain the following set of equations.

$$\begin{aligned}
r_1 &= \tau_1 \\
r_2 &= \tau_2 \\
r_3 &= r_1 + \tau_3 \\
r_4 &= r_1 + \tau_4 \\
r_5 &= r_2 + \tau_5 \\
r_6 &= r_3 + \tau_6 \\
r_7 &= \max(r_4, r_5) + \tau_7 \\
r_8 &= \max(r_6, r_7) + \tau_8
\end{aligned}$$

Note that for series-parallel (SP) graphs[2], the above equations would immediately reduce to a single expression. The above equations, in fact, specify a second (deterministic) graph $R^{(2)} = H \in DG$ of the static *analysis* computation, that is isomorphic to $R^{(1)}$. The transformation is based on the following mapping. Let g_i be a task with predecessor tasks $g_{p(i,j)}$, $j = 1 \ldots P(i)$ ($P(i)$ is the task fanin). Then each task g_i in G maps to a task h_i in H that represents the following computation

$$h_i = \{r_i = \tau_i + \max_{j=1 \ldots P(i)} r_{p(i,j)}\}$$

Application of the transformation to G yields the task graph H for the computation[3] of $T^{(2)}$ ($T^{(2)} = T^{(1)}$), as shown in Fig. 2.2. Thus, $R^{(2)}$ is isomorphic to $R^{(1)}$. The evalua-

[2]An SP graph (also called "simple graph" [131]) is a possibly nested sequence of parallel fork/join sections. An SP graph can be reduced to a single node by applying a sequence of series reductions (i.e., reducing a sequence of two nodes to one node) or parallel reductions (i.e., reducing a parallel section of two nodes to one node).

[3]Note that in this context H is only used to express the analysis *computation*, rather than to investigate the opportunities for parallel processing. Of course, performance analysis can be parallelized as well, but this is not the point of this discussion.

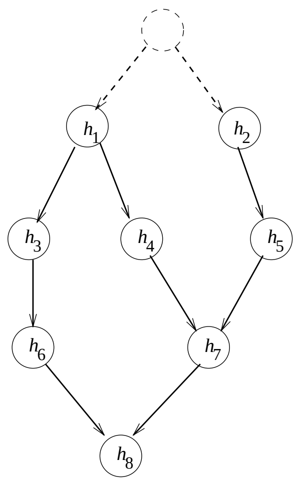


Figure 2.2: Critical path computation graph $R^{(2)}$ of polynomial example

tion of $R^{(2)}$ (i.e., $T^{(2)}$, $T^{(2)} = T^{(1)}$) is commonly known as *critical path analysis* as the execution time of the graph is determined (bounded) by the critical path. (In general graph analysis terminology, the technique is also known as "longest path algorithm" [56]). Thus, static analysis involves an intermediate task graph representation, followed by critical path analysis based on the evaluation of a second, isomorphic computation graph. Of course, the representation $R^{(2)}$ is usually not generated, but merely exists in terms of the above set of equations (or in general in terms of a generic longest path algorithm that dynamically determines the partial order in which the equations (h_i) are to be evaluated (order induced by the data dependencies).

In summary, static analysis can be characterized by the process

$$S \rightarrow DG \rightarrow DG \rightarrow SE$$

in which the ultimate scalar evaluation ($T^{(3)}$) has been ignored for simplicity. The reason for explicitly including $R^{(2)} = DG$ (i.e., two DG representations) in the above process is to distinguish the graph analysis method from techniques used in the analysis of *stochastic* graphs (having stochastic task times) that may also involve state analysis (thus using an MC representation) instead of a critical path method (represented by the second DG, note that the first DG represents the original computation whereas the second DG represents the computation of T).

The crucial abstraction (and consequent loss of accuracy[4]) is performed in the first modeling step. After that, given the deterministic graph, the analysis is exact. The example also illustrates the cost reduction that is generally involved in a transformation. In the simulation approach ($R^{(1)}$), a state machine is simulated that involves the updating of an event list (in case of a discrete event simulator). In the critical path analysis ($R^{(2)}$ or $R^{(3)}$) also a global time variable is updated, essentially involving the same '+' and 'max' operations as in the maintenance of the event list, however, with potentially less overhead as the set of equations may be compiled in contrast to the interpretor-based simulator (i.e., a compiled machine interpretor).

[4]Data communication and memory contention are both ignored.

In the following we present a survey of modeling techniques in which we refer to the above reference model in order to characterize the particular approach used. Consequently, we will only consider on *generic* techniques, i.e., techniques that apply to any program or machine due to the use of a general representation formalism such as QN, PN, DG, etc. Despite the importance of system-specific techniques, the proposed taxonomy provides ample room to accommodate practically all of the well-known approaches.

As discussed earlier, the accuracy as well as the cost of parallel systems performance modeling is greatly determined by four factors

- Condition synchronization
- Mutual exclusion
- Conditional control flow
- Basic Calibration

of which the synchronization factors are of particular interest in parallel systems modeling. Consequently, while discussing each particular approach we will focus on the above factors to determine the merit of each approach, especially in terms of accuracy and cost.

2.2 Queuing Networks

Traditionally, performance modeling of concurrent systems is based on queuing theory (see, e.g., [93]). The fact that a resource (or service in this context) is mutually exclusive between contending clients is symbolized by the queuing center, the basic model element in queuing networks. Conditional control flow is accounted for by calculating the mean service demand on each center based on the branching probabilities that can be assigned to each branch. When the service times are exponentially distributed queuing networks can be mapped to Markov chains due to the memory-less property of the exponential distribution. The Markov chain is solved either using transient or steady-state analysis yielding all state probabilities. Due to the state space explosion this method has exponential complexity. For the class of separable networks [15] the mean value of the system variables can be computed, based on less computation-intensive techniques. A well-known recursive technique is Mean Value Analysis [129] that yields an exact solution for exponential distributions in polynomial time. Bard [14] and Schweitzer [138] describe an iterative approximation. An other approximate method is the use of bounding analysis [93] (for balanced systems see [160]).

Although queuing networks are appropriate for the modeling of systems with independent jobs (or tasks), the formalism has not been intended to account for the condition synchronization between tasks. Hence, traditional queuing networks cannot be used for the performance modeling of parallel systems other than those equivalent to a single parallel task section. Although traditional queuing theory can be applied to the estimation of communication and/or memory delay during various regular phases of a parallel computation[5], aimed towards a more fundamental solution, alternative approaches have

[5]Typically employing a fixed point iteration in which contention delay predicted by, e.g., an M/D/1-like queuing model is used to estimate processor request rate [110, 64].

been described that are based on a hybrid representation. In order to capture the task synchronizations at program level a task graph representation is used while a queuing network is used to model the non-determinism due to conditional control flow and contention for machine resources (i.e., processors, switches, memories). Compared to a full Markov model of the total system, i.e., task graph *and* queuing model, the technique may be viewed as an efficient approximation through the separation into two submodels (a.k.a. hierarchical decomposition) resulting in a considerable reduction in (exponential) complexity [65]. The trade-off is an abstraction from reality as discussed later on.

Thomasian and Bay [150] describe a method that computes the mean execution time based on analyzing the Markov chain derived from the task graph using steady-state analysis as discussed earlier. The transition rates follow from the throughput analysis of the queuing network based on the task load corresponding to the current task composition (Markov state). A comparable approach has been described by Kapelnikov *et al.* [81] in which the solution complexity is decreased by using an aggregation technique in which the throughputs of submodels ("segments") are approximated (based on Markov analysis). The method is especially advantageous for program loops. The above approaches can be characterized by the process

$$S \rightarrow HQ \rightarrow MC \rightarrow SE$$

where the tuple $HQ = (DG, QN)$ denotes the hybrid representation.

Although applying hierarchical decomposition, the above approaches still suffer from the state space explosion. In order to decrease the state space analysis complexity, Mak and Lundstrom [98] describe a method that is optimized for SP graphs. Instead of using Markov analysis an approximate form of SP reduction[6] is applied at task level in which the task times are assumed to be exponentially distributed (thus mean and variance are analytic expressions). From the resulting task residence times the queuing delay is computed using the underlying queuing network. The queuing delays are fed back to the reduction mechanism, that, accounting for the additional queuing delay, closes the loop. Thus an overall polynomial complexity is achieved. Recently, an extension of the above path analysis approach to general task graphs is presented by Adve [1] and by Jonkers *et al.* [78]. The technique is based on critical path analysis in which for each task activity set the proportional contribution of the queuing delay is calculated. A critical but realistic assumption in this approach is the fact that the service accesses for each task are uniformly distributed over the task residence time. For the critical path analysis deterministic task times are used. The underlying premise is that the actual variance at task level does not always necessitate a high-variance distribution like the exponential distribution, as mentioned earlier [3]. Both approaches are characterized by the process

$$S \rightarrow HQ \rightarrow DG \rightarrow SE$$

As mentioned earlier, the hybrid approach not only features an approximate analysis method, but also only partially alleviates the problem of modeling task synchronization. For instance, neither condition synchronization at machine level, nor mutual exclusion at program level can be expressed in this approach.

[6]An efficient implementation of critical path analysis for SP graphs.

2.3 Petri Nets

Petri nets are an effective modeling formalism for the description and analysis of concurrent systems. Since their introduction [118, 119], PN have been deeply investigated, yielding a well-developed theory (see, e.g., [107, 117, 130]). Following [117], we will consider (standard) PN to be Condition Event nets. PN with inhibitor arcs or high-level PN such as Colored PN [77] are considered *extended* PN.

Especially designed for concurrent systems, Petri nets express both CS and ME. Unlike queuing systems the description of ME includes simultaneous resource possession, i.e., the ability to describe atomic access involving several resources. Consequently, the modeling power of Petri nets is larger than of queuing networks or P/V systems[7]. For performance modeling one considers timed Petri nets in which a time delay is associated with each transition. Conditional control flow is modeled by a conflict using the relative firing rate of each of the consuming transitions to determine the mean control flow over the branch. Furthermore, to allow for a meaningful analysis the net is assumed to be live and bounded, such that it is cyclic. As a result of the high modeling power of Petri nets their decision power [117] is low. Hence, for general cyclic nets only *verifying* a certain bound on performance is already shown to be NP-complete [127].

In the context of real-time system analysis Ramamoorthy [127] has investigated nets with deterministic time delays. It is shown that for (cyclic) marked graphs (or decision-free nets[8]) a polynomial time analysis is possible. The vast majority of the approaches, however, is based on using nets with exponentially distributed firing delays, called Stochastic Petri Nets (SPN). Based on their reachability tree, the nets can be mapped onto a Markov chain as shown by Molloy [104]. The Markov chain is subsequently solved using steady state analysis. Thus the predominant approach can be characterized by the process

$$S \rightarrow PN \rightarrow MC \rightarrow SE$$

where PN denotes the (stochastic) Petri net representation. An extension to SPN, called Stochastic Activity Networks, including mechanisms to control transition firing based on any function of the markings is described in [103, 135]. Due to the exponential growth of the transition matrix in the size of the problem, analysis complexity of SPN quickly becomes prohibitive[9]. Aimed to reduce the analysis complexity, Ajmone-Marsan *et al.* [4] describe an extension called "Generalized Stochastic Petri Nets" with immediate transitions to model control flow or activities with negligible time delays. As the analysis complexity of GSPN is effectively dominated only by the number of timed transitions, the distinction of immediate transitions may reduce analysis costs by orders of magnitude. Different approaches to realize state space reduction are described by Plateau [121], Buchholz [22], and Siegle [141]. Although a significant computational reduction can be achieved, the analysis complexity of Petri net approaches is essentially exponential. An alternative approach to reduce complexity is described by Wabnig and Haring [152] which

[7]The increase in modeling power is illustrated by e.g., the Dining Philosopher problem [37] or the Sigarette Smoker's problem [116]. A simple P/V solution entails the risk of deadlock (see Peterson [117] on the issue of modeling power).

[8]In a marked graph each place has one input and one output (i.e., the dual of the state machine).

[9]Even in the case where replication can be exploited by the use of *folding* [5] (in which the net is reduced while maintaining equivalence).

is based on using a hybrid modeling approach. While the machine is modeled in terms of a Petri net, the program is modeled in terms of a task graph. Unlike hybrid queuing networks, the combined model is analyzed using simulation, thus completely avoiding the combinatorial explosion inherent to Petri net analysis at the expense of a somewhat stochastic result. The interesting trade-off between analytic techniques and simulation will be further discussed in Section 5.4.

2.4 Languages

While the above representation formalisms are essentially different from the computational systems under study, in this section we review concurrent imperative languages, that are to a large degree similar. Like in the case of Petri nets, we specifically focus on simulation languages (SL) and timed process algebras (PA) that incorporate the notion of time.

Based on the use of either message-passing constructs (message-oriented paradigm [8]) or semaphore-type constructs (procedure-oriented paradigm [8]), simulation languages naturally account for condition synchronization as well as mutual exclusion. Unlike the other formalisms, data dependent control flow is naturally supported[10], although in a performance simulation context, the usual probabilistic abstraction is applied.

One of the natural advantages of languages is that they provide simple compositional constructs for building large and parametric models. Usually, a message-oriented modeling paradigm is used that corresponds to the object-oriented modeling approach taken by most modelers in which the concepts of inheritance and information hiding are useful for model engineering. Characteristic examples are SIMULA [33] and, more recently, the CSIM17 library described by Schwetman [139] and the language Pearl described by Muller [105].

As the modeling detail of simulation can be chosen to be arbitrary high, this form of dynamic performance evaluation is most near to actual system execution. Even when the actual system is available (i.e., does not need to be predicted) simulation is often chosen above actual execution because the data measurement and collection processes do not perturb the system's dynamic behavior (in terms of virtual time). Due to the above advantages a vast number of prediction approaches based on simulation modeling have been described (e.g., [108, 146, 101, 120, 128, 142]), including the simulators within the program cost estimation tools of Van Halderen [62] and Qin *et al.* [125].

Unlike the earlier approaches, simulation languages typically lack an explicit *analytic* tradition. Hence, their characterization is simply

$$S \rightarrow SL$$

In contrast, stochastic process algebras, a temporal extension to classical process algebras (e.g., ACP [11], CSP [69]) do have an underlying calculus. Typical examples of this approach are the work described by Götz *et al.* [57], and Hillston [67]. Both approaches are based on the introduction of exponentially distributed delays associated with the actions (as in stochastic Petri nets). Similar to other stochastic approaches the model is

[10]Despite the higher modeling power of Petri nets in terms of synchronization compared to, e.g., simple P/V languages (concerning simultaneous resource possession) it would take an extended Petri net (featuring inhibitor arcs) to achieve Turing power. Any simulation language, in contrast, can determine conditional control flow as a result of its inherent capability to compute numbers.

subsequently transformed into a Markov chain, that is solved using standard techniques. Consequently the approach can be characterized by the process

$$S \rightarrow PA \rightarrow MC \rightarrow SE$$

Although there are many differences between simulation languages and process algebras, there are similarities. Process algebras are much like message-oriented performance simulation languages. Synchronization is based on cooperation or communication, that implies that mutual exclusion is based on the fact that a process can only engage in a rendez-vous with one partner (typically selected non-deterministically through the '+' or '$\Box$' operator) at the same time.

2.5 Stochastic Graphs

As mentioned in the introduction, task graphs are a popular representation form for the analysis of parallel computation structures (algorithms) in which task precedence relations are of primary interest. Due to their static structure, task graphs cannot model mutual exclusion, nor can they model conditional control flow (unless by weighting all the work loads of conditional tasks in a branch with all the associated branching probabilities, e.g., like in queuing networks). Due to the fact that mutual exclusion is not accounted for (unless explicitly modeled in terms of additional delay nodes [99]), the predictive value of task graphs is limited. The trade-off, of course is the potential for a cost-effective analysis. Because of the relevance of task graph analysis for our approach in the following we will treat task graph approaches in somewhat more detail.

Because of the static nature of task graphs in many modeling approaches the task times are often chosen to be stochastic in order to still enable some sort of representation of the non-deterministic effect of conditional control flow and/or contention at the subtask level. In this case the task graphs are commonly termed *stochastic graphs* (SG).

As illustrated in the earlier example, in task graphs the analysis procedure is based on evaluating the effect of task precedence relations on time. While for deterministic graphs the procedure is known as critical path analysis, in the general case of stochastic graphs this procedure involves a more elaborate analysis in order to compute the distribution of T. Still, the approach can be characterized by the process

$$S \rightarrow SG \rightarrow DG \rightarrow SE$$

where DG denotes the deterministic (critical path) analysis method. Let $G \in SG$ denote a task graph comprising N tasks $g_1 \ldots g_N$ where g_1, g_N denote, the top and bottom task, respectively. Let T_i denote the distribution of the execution time of g_i, i.e., the time that g_i finishes. Consequently, $T = T_N$. Let $F_i(t)$ denote the task delay time distribution of g_i. Let π_i denote the set of predecessors of g_i. Let $S_i(t)$ denote the distribution of the task start time, i.e., the time when all predecessors are finished. Due to the barrier synchronization at g_i it holds

$$S_i(t) = \prod_{j \in \pi_i} T_j(t) \tag{2.1}$$

where it is assumed that the F_i are mutually independent. The distribution of T_i is determined by $S_i(t)$ and the local task delay distribution $F_i(t)$ according to the convolution

$$T_i(t) = (S_i * F_i)(t) = \int_0^t S_i(\tau) f_i(t-\tau) d\tau \tag{2.2}$$

where $f_i(t)$ denotes the probability density function of the task delay time.

As illustrated in the introduction, the analysis computation can be described by a computation graph that is isomorphic with G. For general graphs the number of distribution products and convolutions is given by $\mathcal{O}(N^2)$. For SP graphs the complexity reduces to $\mathcal{O}(N)$.

While for deterministic graphs path analysis entails $\mathcal{O}(N^2)$ scalar operations at worst, for stochastic graphs of any practical size the calculation of T based on path analysis is prohibitive unless restrictions are introduced. Basically, two approaches can be distinguished, i.e., (1) by limiting the distribution functions representation, and (2) by limiting the scope of graph structures. The first restriction entails a reduction of the complexity of Eqs. (2.1) and (2.2). The use of deterministic task times is, of course, the most extreme example in which the above equations reduce to a scalar 'max' and '+', respectively (in terms of f, that is). The second restriction limits the number of computation steps necessary to compute the result. The prominent subclass are the SP graphs, in which case critical path analysis reduces to SP reduction. Another reason for the popularity of this approach is that many computations can be expressed in terms of SP graphs, or can be approximated in terms of SP graphs[11]. Both approaches may be combined, of course.

An example of the first approach is the work by Lester [95] who uses the geometric transform of f (a.k.a. z-transform or generating function) in terms of which Eq. (2.1) is a product of polynomials and Eq. (2.2) is called "join product". One of the motivations for this approach is that non-deterministic conditional control flow is naturally accounted for in terms of this discrete transform. An example of the second approach is the work by Gelenbe *et al.* [47] in which a closed form expression is derived for the execution time distribution of a class of random SP graphs that obey certain stochastical rules concerning their construction. An example of the combined approach is the work by Sahner and Trivedi [134] who describe a method to compute this function for task time distributions that have an exponential polynomial form (i.e., $f_i = \sum_k a_k t^{l_k} e^{b_k t}$). Since exponential polynomials are closed under the SP reduction operations T will also be an exponential polynomial. Thus, unlike the geometric transform this (intermediate) representation does not necessarily grow during the analysis.

Without the above restrictions, for practical graphs the analysis cost of the distribution of T is prohibitive. As a result, many approaches restrict to the characterization of the execution time in terms of *scalar* metrics only, such as the mean and variance, or lower and upper bounds. One example is the bound approach that is described by Yazici-Pekergin and Vincent [159] who compute stochastic bounds on the mean completion time for general graphs. The basic idea is that a deterministic version of the graph produces a lower bound while the upper bound is provided by a version that assumes independence of all paths in the task graph. A related approach is described by Lester [95] that applies SP reduction

[11]For instance, Hartleb and Mertsiotakis [63] describe a bounding analysis by deriving SP approximations with the use of heuristics.

for mean values based on the reduction rules for deterministic values. While appropriate for series reduction and conditional control constructs, parallel reduction based on a simple max function (i.e., without using order statistics) yields an underestimation for stochastic variables (i.e., for which $\sigma > 0$). Consequently, the result is a lower bound for T. In order to account for task variance an enhanced scheme is also described in which both mean and variance are propagated in the course of the SP reduction. Parallel reduction is based on an approximation of order statistics assuming a normal distribution. A comparable approach including the use of order statistics is described by Robinson [131] in which the SP graphs are called “simple graphs”. Sarkar [136] describes a comparable technique for sequential control graphs in order to determine the effect of conditional control flow.

Another popular approach to computing the mean execution time is based on restricting the task distributions to the exponential type. Thus, a Markov chain can be constructed of all the ($\mathcal{O}(2^N)$) activity states of the graph. Typically, a transition is added corresponding to a cycle from t_N to t_1 in order to allow for steady state analysis. Unlike most Petri nets and queuing networks the task graph itself has no internal cycles. Consequently, the transition matrix is triangular in which case the direct solution complexity is approximately $\mathcal{O}(2^N)$ instead of the usual $\mathcal{O}(2^{3N})$ for full matrices [150]. The approach is characterized by the process

$$S \rightarrow SG \rightarrow MC \rightarrow SE$$

Although the use of exponential distributions is quite popular due to the Markov property, this distribution is not particularly realistic for the time behavior of tasks exhibiting low-level non-determinism (control flow, queuing). Usually such tasks have much less variance as observed by Adve and Vernon [3] (Lester [95] reports that unconditional loops with even a few iterations can be approximated by *normal* distributions within a few percents accuracy). The use of Erlang-k distributions with a high number of phases in order to decrease variance, however, results in a significant increase in complexity. Hence, approximations are used. An example is the approach used by Sötz [144] in which he approximates low-variance task distributions by a series combination of a deterministic and exponential task term such that the first two moments are equal to the Erlang-k distribution. As the memory-less property of the exponential distribution is lost an approximate method is used to solve the Markov chain.

2.6 Deterministic Graphs

As mentioned earlier for deterministic graphs the analysis is based on critical path analysis. In terms of the stochastic graph analysis the product and convolution formulae reduce to a scalar addition and maximum, respectively. As our symbolic approach is based on the use of deterministic graphs we present a more elaborate treatment of the related work that employs this representation type. As discussed in the introduction, the approach is characterized by the process

$$S \rightarrow DG \rightarrow DG \rightarrow SE$$

Almost every approach is restricted to SP reduction (SE is one expression) that yields the fastest analysis possible ($\mathcal{O}(N)$ complexity) which partly explains why all compile-time prediction approaches are based on this technique. Since most reductions are stated in terms of an (intermediate) language, the actual process is even linear in the size of the program as can be seen as follows. Due to the predominant procedure-oriented style in parallel programming (either a fork/join style in the case of an explicitly parallel dialect, or a data parallel style in case of implicit parallelism), programs can be represented by an SP task graph that is amenable to SP reduction. Expressed in terms of the source language itself, the program is consequently termed an SP program. In program terms SP reduction is implemented by the following function that is recursively applied to a (compound) statement S (initially, S is the program).

- if $S =$ `S1; .. Sn;` then $T(S) = T(S1) + .. + T(Sn)$
- if $S =$ `for i = 1 .. n do S(i)` then $T(S) = \sum_{i=1}^{n} T(S(i))$
- if $S =$ `forall i = 1 .. n do S(i)` then $T(S) = \max_{i=1...n} T(S(i))$
- else $T(S)$ represents time cost of the basic instruction.

The use of a language enables the conditional control flow analysis to be included within this scheme according to

- if $S =$ `if c then S1` then $T(S) = pT(S1)$

where p denotes the probability that `c` evaluates true. This is equivalent to weighting all the work loads of tasks within a branch with all the associated branching probabilities (note that like in e.g., queuing networks, in this "mean value" approach, all the branching probabilities involved are assumed independent).

In terms of the above SP reduction, all the deterministic approaches are practically similar. In contrast to the use of stochastic task times, in compile-time techniques *ad hoc* approaches are typically followed in order to account for mutual exclusion (if accounted for at all). In view of the importance to account for resource limitations (contention) in parallel systems, we will discuss the various reduction techniques with respect to this particular aspect. In the estimation approaches for shared-memory systems described by Allen *et al.* [7], Sarkar [137], and So *et al.* [143], the effect of dynamically scheduling the task graph given a limited number of processor resources is approximated in terms of bounds by applying Graham's result for list scheduling [58]. Also Polychronopoulos and Banerjee [122] describe bounds on the speedup for **doacross** loops (a generalization over **do** and **doall** due to Cytron [32]) when cyclically scheduled on a limited number of processors. The method presented by Wang [155] includes an approximation of cache performance and hot spot contention for memory resources. Targeted at real-time systems, Shaw [140] presents an estimation scheme that computes both a lower and upper bound on the execution time. The scheme approximately accounts for contention, be it for non-preemptive resource sharing only (e.g., critical sections). The approach does not consider preemptive processor sharing and memory sharing.

While for shared-memory machines resource limitations primarily relate to (dynamic) processor scheduling and memory contention, in distributed-memory machines, the contention issues relates to network contention as the processor mapping is typically static. In the approach by Balasundaram *et al.* [13] the underlying distributed-memory machine is characterized in terms of its collective message-passing interface, thus automatically accounting for network contention. The above approaches are more or less based on reduction down to a single estimate (or derivative). Clearly, however, one single number does not provide the diagnostic information that enables an efficient optimization process. Based on this argument, Fahringer and Zima [41] present a number of specific diagnostics for distributed-memory systems, such as processor load balance, communication volume, cache performance, as well as an indicator for network contention.

A number of approaches have been described in which the prediction is still expressed in terms of certain symbolic parameters, thus providing another form of diagnostic information (suitable for, e.g., scalability analysis). Atapattu and Gannon [10] describe a partially symbolic estimation approach for shared-memory systems that includes an analytical approximation of memory contention (based on a queuing model) and of cache behavior (i.e., conditional control flow) for a specific architecture. The shared-memory program performance estimation tool of Qin *et al.* [125] also generates a partially symbolic output in terms of the simulation model source text. The effect of (dynamic) processor sharing between tasks is accounted by computing the available processing bandwidth while traversing the computation graph. The analysis of non-preemptive memory contention however is approached by accounting (enumerating) for all the paths that are possible, thus leading to exponential complexity. Wang [156] describes a symbolic prediction technique that is used for super-scalar based processor design. Aim to enable efficient scalability analysis, Mendes, Wang and Reed [102] describe a method to derive a symbolic model in terms of the problem size, the number of processors and a number of system parameters that are derived by fitting the model on execution time measurements for different problem sizes and processor numbers. In the symbolic approach by Clement and Quinn [29] the system parameters are also determined from measured run times. However, they are computed from instruction counts as actually profiled, rather than predicted as with the model of Mendes *et al.*. Both approaches do not account for contention.

Essentially all static approaches have the same "mean value" approach towards conditional control flow in which the actual work load is weighted in terms of the (appropriate) branching probabilities. The approach is basically similar to sequential system analysis [136, 153]. In cases where the branching probability can not be deduced at compile-time, either a default is assigned (e.g., [13]) or the probability is based on statistics gathered during profiling. An underlying assumption is that program level conditional control flow is characteristic of the program and, consequently, is independent of other system parameters. Evidence to suggest this have been produced by Fahringer and Zima [41]) who successfully uses control flow statistics based on sequential profile runs. Unlike program level, at machine level, conditional control flow (e.g., caches), is usually dependent on system parameters. Again, when the hit ratio can not be deduced at compile-time (typically based on a loop model [10, 41, 155]), profile statistics (possibly obtained by simulation [38]) must be used.

Some of the analytic approaches feature calibration methods in which certain program parameters are determined by direct measurement (e.g., [30, 73]), or where aggregate submodels are timed as a whole. A typical example is the deduction of the startup and bandwidth parameters of vector instructions by fitting measurement data to the underlying linear (pipeline) model [70] for various vector lengths and for various access strides. In this way the effects of low level memory hierarchy and memory bank contention are automatically accounted for. One of the first approaches in this direction was taken by Gallivan*et al.* [46], in which program performance was predicted by characterizing program workload in terms of a kernel of basic (vector) routines (including associated memory traffic). A comparable approach is taken by Balasundaram *et al.* [13], in which the (distributed-memory) machine is characterized (i.e., measured) in terms of its collective communication interface. Again, the effect of message pipelining, link contention, etc. is automatically accounted for. Clearly, modeling the machine in terms of (judiciously chosen) aggregate submodels instead of at a more basic instruction level has a benefit in terms of modeling efficiency. Another approach in which communication cost is estimated in terms of an MPI-like abstract kernel instead of the actual code (as would be generated by the compiler) is described by Gupta and Banerjee [61]. In general, the trade-off in these approaches is the cost (and accuracy) of workload characterization at program level as now the program must be decomposed in terms of a (usually less orthogonal) high-level kernel instead of basic instructions. A detailed study into the feasibility of automatically recognizing kernel operations in parallel programs is described by Keβler [84]. Other static approaches feature calibration methods in which the basic model parameters are inferred rather than directly measured, based on the use of statistical techniques. Two approaches that have been discussed are those by Clement and Quinn [29] and by Mendes *et al.* [102]. An more phenomenological approach is taken by Candlin [27] in which the influence of several program statistics (e.g., granularity, task time variance) on system performance are studied using a (two-level) factorial analysis method.

2.7 Summary

We conclude this chapter with a short summary of the techniques considered in this review. The various approaches, along with their primary characteristics are listed in Table 2.1. The columns 'CS', 'ME', and 'CF' signify whether the approach accounts for condition synchronization, mutual exclusion, and, conditional control flow, respectively. The *complexity* column lists the solution complexity in terms of the characterization polynomial ('pol', ranging from high complexity to the quadratic or even linear complexity of the *DG* approach), or exponential ('exp'). Note that a less favorable "mark" (i.e., '-' or '□') should be strictly interpreted in the sense of *generic* applicability. Sometimes a formalism is not *intended* to be used in a system-level generic way. For instance, stochastic graphs are quite useful for the analysis of task systems *once* the underlying machine has been chosen. In this approach extending the model in order to include the machine would only entail a needless increase in analysis complexity. Thus, depending on the specific purpose, any of the above techniques may be an optimal choice. From the perspective of a *generic*, system-level modeling formalism, however, the above table holds. We will briefly summarize the approaches in terms of the basic properties of interest as we have defined earlier.

- Modeling ($R^{(1)}$):
  All the approaches inherently account for condition synchronization except traditional queuing networks. Note, however, that the hybrid approaches only account

for condition synchronization at program level. All the approaches inherently account for mutual exclusion, except for the techniques based on task graphs. The '□' sign for stochastic graphs denotes their (small) ability to account for the non-determinism associated with contention. Note that hybrid queuing techniques only account for mutual exclusion at the machine level. The only technique that is fully capable to model conditional control flow is simulation. All other techniques are based on a probabilistic abstraction (branching probabilities).

- Analysis ($R^{(2)}$):
  Most solution techniques are based on a numeric process such as simulation, the recursive/iterative solvers for (hybrid) queuing networks (SE[12]), as well as the direct solvers for Markov chains (MC). Only the techniques of which the solution can be expressed in terms of deterministic graphs are eligible for a symbolic process (except HQ due to the underlying network solver) as exemplified by the compile-time symbolic techniques. Associated with each type of analysis process is a trade-off between information and cost. Simulation is a computation-intensive solution technique partly since it yields only one sample of the execution time distribution for each run (i.e., a stochastic model). Analytic techniques, on the other hand, provide the solution in only one process run (i.e., a deterministic model). However, the cost may become prohibitively high. Methods based on Markov analysis yield probability information on each individual state, however, at exponential costs as the state space is exponential in the system size. Under certain conditions, solution techniques for (hybrid) queuing networks can be used that only provide mean values but which have costs that are only polynomial. Deterministic graph analysis has the lowest complexity (quadratic for general graphs, linear for SP graphs). The trade-off is a limited accuracy due to the fact that task delay distributions as well as mutual exclusion are ignored.

$R^{(1)}$	CS	ME	CF	$R^{(2)}$	*compl.*
QN	-	+	□	MC	exp
QN	-	+	□	SE	pol
HQ	□	□	□	MC	exp
HQ	□	□	□	DG	pol
PN	+	+	□	MC	exp
SL	+	+	+	-	pol
PA	+	+	□	MC	exp
SG	+	□	□	DG	pol
SG	+	□	□	MC	exp
DG	+	-	□	DG	pol

Table 2.1: Approaches to performance modeling of parallel systems

[12]The only occurence of SE in the table is due to the MVA technique, which simply maps QN onto a (recursive) set of equations, without using a second, intermediate representation formalism.

As mentioned before, due to the choice to survey performance modeling in terms of the (intermediate) representation formalism used, not every approach has been accounted for. An obvious example are the system-specific approaches that map some (sub)system into a (usually optimized) analytical model. The best way to characterize these approaches would be the default process

$$S \rightarrow SE$$

For instance, the derivation of the linear, symbolic performance model for vector processing (either arithmetic processing, or data movement) is based on an immediate, manual, derivation process, rather than through some intermediate representation (using, e.g., a Petri net will not yield the symbolic solution[13]).

Returning to the criteria mentioned in Chapter 1, none of the above modeling approaches adequately pairs accuracy with an analytic method that comes with sufficiently low solution cost. On the one hand, QN, HQ, SG, and DG do not always provide sufficient modeling accuracy. On the other hand, PN, and PA entail exponential costs. Although SL pairs accuracy with polynomial costs, they merely provide a stochastic result (a single draw) rather than an analytical model. Our approach aims to combine an analytical technique with sufficient accuracy. In terms of the earlier table, our approach can be described as given in Table 2.2.

$R^{(1)}$	CS	ME	CF	$R^{(2)}$	*compl.*
DG	+	+	□	DG	pol

Table 2.2: The Pamela approach to performance modeling of parallel systems

[13]When using Pamela the automatic derivation of a symbolic model is indeed possible, as is shown in the next chapter.

Chapter 3

PAMELA

3.1 Introduction

In this chapter we present our approach to the performance modeling of parallel systems. The methodology we propose is based on the use of a performance modeling formalism, called Pamela (PerformAnce ModEling LAnguage), that,

- with respect to *modeling* provides a concise, procedure-oriented performance simulation interface. Unlike most abstract representation formalisms, the language allows the description of models without forcing significant *a priori* loss of accuracy. Primarily intended as a description formalism for subsequent analysis, the language features operators to express structured forms of condition synchronization and, most notably, mutual exclusion.

- with respect to *analysis* introduces a novel static technique that, due to a procedure-oriented and structured synchronization paradigm, allows simulation models to be compiled into an analytic (parameterized) model that trades accuracy for cost. Unlike traditional compile-time analysis the method introduces an approximate analysis of mutual exclusion within the critical path analysis, thus yielding a lower bound estimation T^l of the simulation result T that is much tighter than conventional predictions, yet at the same cost.

In terms of the previous chapter, the approach can be characterized by the process

$$\begin{tikzcd}
S \arrow[r] & SL \arrow[r] \arrow[d] & DG \arrow[r] & SE \arrow[r] \arrow[d] & \ldots \arrow[r] & SE \arrow[d] \\
& T^{(1)} & & T^{(2)} & & T^{(n)}
\end{tikzcd}$$

In the Pamela methodology (see Fig. 3.1) both parallel program and parallel machine are converted into their Pamela model counterparts. After substitution of the machine model into the program model the combined model is either compiled into a simulation object returning a result T ($T^{(1)}$) or compiled into an analytical model that yields the lower bound estimate T^l ($T^{(2)}$) at much less cost. Based on the analytical model, subsequent reductions are possible (see figure), again possibly trading accuracy for cost ($T^{(n)}$). Thus, the approach offers a flexible trade-off between prediction accuracy and cost. In a typical

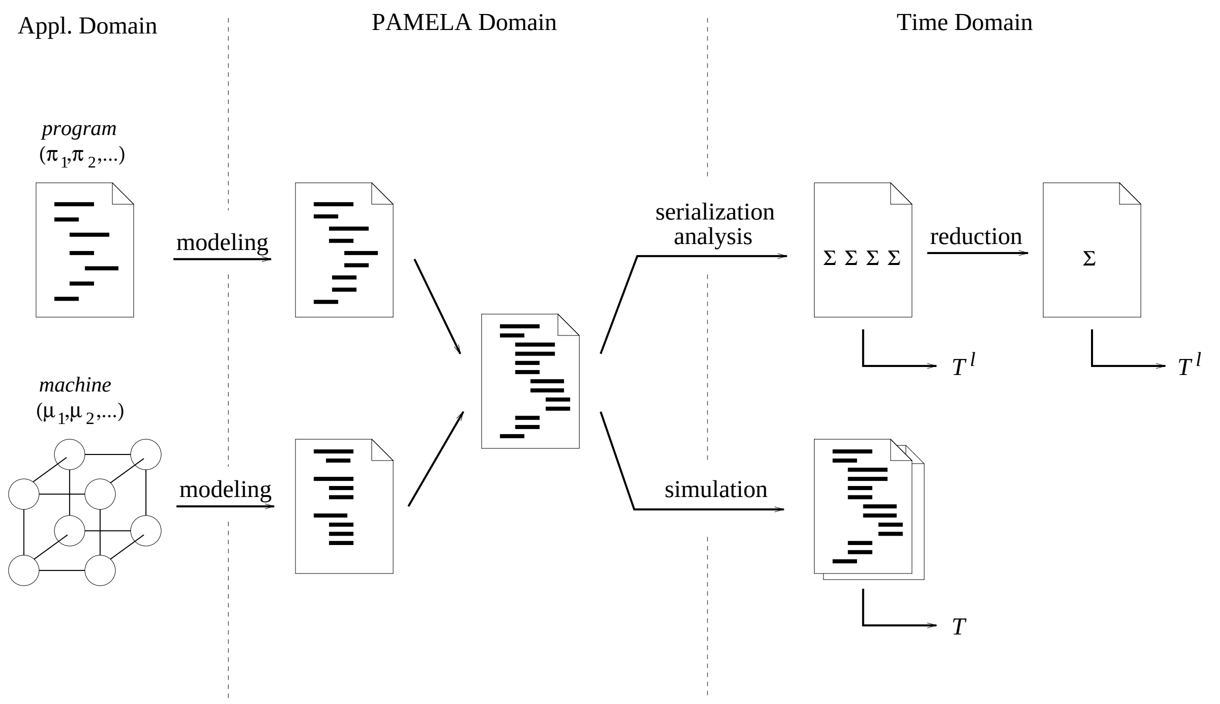


Figure 3.1: Modeling methodology

application environment, program model generation, machine model substitution, as well as subsequent model compilation are performed on line, whereas the machine model has been programmed in advance.

The rationale for the choice of a procedure-oriented simulation language combined with a symbolic compile-time calculus is the following.

- symbolic modeling
  Unlike other representation formalisms, languages provide a natural means to express parameterization, as well as a convenient set of constructors to easily express composition, replication, etc. Due to the procedure-oriented modeling paradigm a path analysis method can be used that yields a time domain model that is also completely symbolic, whereas the message-oriented paradigm of traditional languages and process algebras as well as the choice of other representation formalisms (queuing networks, Petri nets) entails a numerical solution process (with the exception of analytic solutions that are manually derived).

- modeling power
  With its ability to express condition synchronization, mutual exclusion, as well as (data dependent) conditional control flow, the language provides high modeling power to capture the performance behavior of parallel computer systems. Even when conditional control flow is modeled probabilistically, the language still combines the modeling power of task graphs, (hybrid) queuing networks, as well as process algebras and Petri nets as a result of its capability to express simultaneous resource possession.

- analyzability
  The choice for a structured, procedure-oriented paradigm, combined with structured operators to express mutual exclusion offers the possibility of a compile-time path analysis technique that approximately accounts for mutual exclusion. Hence, the resulting time domain model has the minimum robustness that is needed in view of the very large parameter space covered by the analytic model. The choice for a low-cost, deterministic path analysis technique is motivated by the fact that in many cases task time variance is limited [3]. Hence the analysis error due to the the assumption of deterministic task times (mean values) instead of accounting for task variance is acceptable in view of the overall approximation (indeed the use of a deterministic scheme is suggested in [3]).

Thus our methodology distinguishes itself from the other approaches by the combination of a procedure-oriented language, structured operators to describe mutual exclusion, and the (consequent) compile-time calculus that allows for the automatic compilation of symbolic performance models.

In the remainder of this chapter, the language and underlying calculus are presented. Rather than providing a formal language description, in Section 3.2 an informal introduction is given of the language as well as many examples. A rationale for the material-oriented modeling paradigm that is adopted in Pamela is given in Section 3.3. Section 3.4 presents the underlying calculus as well as a number of examples. Parts of this chapter have been presented in [49, 51].

3.2 Language

Being a research vehicle for the formalization of concurrent computations the syntax and semantics are not (yet) rigorously defined. Consequently, we will refrain from a formal language definition and loosely describe its syntax and semantics in an informal way using examples. A semantics description of the most important language constructs, expressed in terms of Deterministic and Stochastic Petri Nets (DSPN [6]) can be found in Appendix A.

Basically, Pamela is an imperative, process-oriented simulation language. Thus, like other simulation languages it is capable of functionally simulating concurrent computations on von Neumann machines. Unlike most simulation formalisms, however, its design is tailored to facilitate a compile-time performance analysis. Of course, this compile-time analysis is strictly defined for performance simulation models, i.e., the subset of simulation models that exclude original data computations.

Intended as a source language for compile-time analysis, the syntax of Pamela is reminiscent of that used in ordinary mathematics. Rather than defining a full fledged programming language (including type declarations, etc.) we simply borrow the equation syntax and substitution semantics as found in mathematical formalisms. Thus, much like in process algebra, a Pamela program or process (usually denoted by L) is written as a set of algebraic equations that describe the simulation model of the system under study. We will also often refer to a Pamela program as a Pamela model.

Like any simulation language PAMELA supports the notion of virtual execution *time* that is the key (performance) result of the simulation. By convention, T denotes the (simulated) execution time of program L. By definition, the time of the empty program is zero. The basic operation that increments virtual time is the **delay** construct that takes a time increment as argument. For example, the execution time of the PAMELA model

$$L = \textbf{delay}(\tau)$$

is given by

$$T = \tau$$

The time interval τ can be either deterministic or stochastic according to some specified distribution.

3.2.1 Control Flow

To enable basic model construction PAMELA provides the following *control flow* operators:

- sequential operator: ;
  For example, the following PAMELA program (or model) $L = \textbf{delay}(\tau_1)$; $\textbf{delay}(\tau_2)$ specifies a strict sequence of two processes (or submodels) $\textbf{delay}(\tau_1)$ and $\textbf{delay}(\tau_2)$. Consequently, $T = \tau_1 + \tau_2$. Clearly, ';' is associative.

- parallel operator: $\|$
  For example, the following PAMELA model $L = \textbf{delay}(\tau_1) \parallel \textbf{delay}(\tau_2)$ specifies two processes running in parallel without any intermediate form of synchronization. Similar to the well-known parallel constructs like **forall**, the **par** construct has an implicit barrier semantics. Consequently, $T = \max(\tau_1, \tau_2)$ due to the additional synchronization delay. Clearly, '$\|$' is associative and commutative.

- conditional operator: **if**
  For example, the following PAMELA program $L = \textbf{if}\ (r < p)\ L_1$ where r is a random variable uniformly distributed over $[0, 1]$, implements a branch with average branching probability p $(0 \leq p \leq 1)$. For programming convenience an **else** construct is included.

In order to specify sequential as well as parallel *replication* PAMELA provides reduction operators defined by

$$\begin{aligned}\textbf{seq}\ (i = a, b)\ L_i &= L_a\ ;\ \ldots\ ;\ L_b \\ \textbf{par}\ (i = a, b)\ L_i &= L_a \parallel \ldots \parallel L_b\end{aligned}$$

The simple lower/upper bound syntax is inspired by corresponding reduction functions in the time domain such as the $\sum$ operator. A more detailed discussion on the semantics of sequential and parallel composition is given in Section 3.4.

As a simple example, the following set of equations

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{delay}(\tau_1) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{delay}(\tau_3) \ ; \ L_3 \\ L_3 &= \mathbf{delay}(\tau_4) \parallel \mathbf{delay}(\tau_5) \end{aligned}$$

constitute a legal Pamela SP model that is equivalent to

$$L = \{\mathbf{delay}(\tau_1) \ ; \ \mathbf{delay}(\tau_2)\} \parallel \{\mathbf{delay}(\tau_3) \ ; \ \{\mathbf{delay}(\tau_4) \parallel \mathbf{delay}(\tau_5)\}\}$$

Although the first '{ }' pair is superfluous (the ';' operator has a tighter binding than '||'), in order to avoid confusion '{ }' or '( )' pairs are used throughout the text.

The above operators form the ingredients for a basic performance simulation kernel. In order to enable *functional* simulation as well this kernel is extended with

- iteration operator **while** (c) L
- usual computational data types including assignment in order to enable full data computation. Consider the sequence $L = \mathbf{delay}(1);\ x = 2;\ \mathbf{if}\ (x^2 < 4)\ \mathbf{delay}(1)$. It follows $T = 1$.

The **while** operator is required for simulation purposes only (where termination is not essential). In order to allow the analytical approach to be used as well, the **while** construct is typically replaced by a **seq** operator where the loop bound is specified by some symbolic user parameter. Although original data computations are typically left out of the model for analytical reasons, in some cases the use of data computation cannot be avoided in order to compute data-dependent (conditional) control flow. For the compile-time analysis, however, these computations are typically accounted for by branching probabilities, thus yielding a model only in terms of the basic performance simulation kernel.

3.2.2 Condition Synchronization

Apart from the implicit condition synchronization inherent in the parallel (and sequential) composition constructs, Pamela offers explicit condition synchronization operators, i.e., **wait** and **signal** in order to express inter-process precedence relations in a manner that is compile-time deducible and efficiently executable at run-time. Both operators take a set of boolean conditions as argument. Let $C = \{c_1, \ldots, c_N\}$ denote a set of N conditions. The operation **wait**(C) suspends a process until all conditions are true, i.e., $c_1 \wedge \ldots \wedge c_N = true$. The operation **signal**(C) assigns a true value on each of the member conditions. For single conditions ($|C| = 1$) the set notation may be omitted. For example, the Pamela model

$$\{\mathbf{delay}(1) \ \ \mathbf{wait}(c) \ ; \ \mathbf{delay}(5)\} \parallel \{\mathbf{delay}(5) \ ; \ \mathbf{signal}(c) \ ; \ \mathbf{delay}(1)\}$$

yields $T = 10$. Note that subsequent **wait** or **signal** operations on the same variable have no effect. Effectively, the number of processes that is allowed to **signal** a condition is constrained to be one.

While the implicit synchronization of the parallel (and sequential) operators discussed earlier enables the construction of SP graphs, the explicit **wait/signal** operators allow for the expression of *any* task graph (like the above non-SP model). Note that a construct

while $(\neg c)$ < do nothing >

instead of **wait**, in combination with a simple truth assignment $c = true$ (i.e., substitute for **signal**) does not implement a conditional synchronization in any simple discrete event simulator (like the PAMELA Run-Time Library, see Appendix E) as the virtual time of the executing process would never advance. In other words, the construct would block the process indefinitely. Thus, the **wait** construct should be used instead, in combination with **signal**.

The choice for **wait/signal** operators with the above constraint instead of simply using semaphores is deliberate. A crucial difference between the above condition synchronization operators and semaphore operators (as discussed later on) is that the **wait/signal** operations are assumed to be used only *once*, corresponding to satisfying the unique precedence between two synchronizing tasks. In contrast to a (counting) semaphore a condition variable implements no memory other than the state of the **signal**ing task. Thus it is a *single assignment* variable, that allows for the application of the functional analysis approach that is discussed in Section 3.4.

Example 3.1 In this example we demonstrate the description of parallelism in conjunction with condition synchronization. Recall the parallelization of the polynomial computation (cf. Fig. 1.3) in which we assume an idealized machine model. Let each multiplication and addition correspond to a work load of τ_m and τ_a time units, respectively. The PAMELA model is given by

$$\begin{aligned}
L &= \textbf{par}\ (i = 1, 8)\ L_i \\
L_1 &= \textbf{delay}(\tau_m)\ ;\ \textbf{signal}(\{c_{13}, c_{14}\}) \\
L_2 &= \textbf{delay}(\tau_m)\ ;\ \textbf{signal}(c_{25}) \\
L_3 &= \textbf{wait}(c_{13})\ ;\ \textbf{delay}(\tau_m)\ ;\ \textbf{signal}(c_{36}) \\
L_4 &= \textbf{wait}(c_{14})\ ;\ \textbf{delay}(\tau_m)\ ;\ \textbf{signal}(c_{47}) \\
L_5 &= \textbf{wait}(c_{25})\ ;\ \textbf{delay}(\tau_a)\ ;\ \textbf{signal}(c_{57}) \\
L_6 &= \textbf{wait}(c_{36})\ ;\ \textbf{delay}(\tau_m)\ ;\ \textbf{signal}(c_{68}) \\
L_7 &= \textbf{wait}(\{c_{47}, c_{57}\})\ ;\ \textbf{delay}(\tau_a)\ ;\ \textbf{signal}(c_{78}) \\
L_8 &= \textbf{wait}(\{c_{68}, c_{78}\})\ ;\ \textbf{delay}(\tau_a)
\end{aligned}$$

Each task is expressed as executing in parallel while the **wait/signal** pairs express the data dependencies between them, thus constraining the actual parallelism. □

3.2.3 Mutual Exclusion

Being a procedure-oriented language, the basic mechanism to implement mutual exclusion are counting semaphores [37], in conjunction with the *simultaneous* semaphore operators **P** and **V** [97]. Like process parallelism, the notion of mutual exclusion is central in PAMELA as it stands for the (exclusive) use of *resources*. Given their full-empty semantics the semaphore abstract data type is a natural mechanism to model the use of resources. Consequently, in the following we will often denote a (counting) semaphore by the term resource that can take any integer value greater than or equal to zero.

Let $R = \{r_1, \ldots, r_M\}$ denote a finite set of M integer variables (resources). Let $U = \{\ldots, r_i, \ldots\}$, $U \in R^\infty$, denote a *multiset* [117] of resources. Let $\# : R \times R^\infty \rightarrow N$ denote the multiplicity function such that $\#(r_i, U)$ returns the number of occurrences of r_i in the multiset U. The process

$$\mathbf{P}(U)$$

is suspended until in holds $\forall i : r_i \geq \#(r_i, U)$, after which it unblocks and it instantaneously holds $\forall i : r_i = r_i - \#(r_i, U)$. Conversely, the process

$$\mathbf{V}(U)$$

restores R according to the post condition $\forall i : r_i = r_i + \#(r_i, U)$. For example, let $r_1 = 1$ and $r_2 = 2$. While the operation $\mathbf{P}(\{r_1, r_2, r_2\})$ will acquire all resources, the operation $\mathbf{P}(\{r_1, r_1, r_2\})$ would block until the occurrence of a $\mathbf{V}$ operation that returns at least one unit r_1. Notice that the above definitions allow for the expression of both simultaneous resource possession as well as instantaneous consumption and production multiplicity due to the multiset definition of the argument[1]. For a single resource the above set notation is omitted in which case the above operations have the usual syntax of simple semaphore operations.

Example 3.2 In this example we present a typical application of counting semaphores. Consider a producer-consumer scheme around a bounded buffer with B storage cells that is initially zero. Let the production time and consumption time be τ_p and τ_c, respectively. Let N denote the number of items being produced, buffered, and consumed. The performance simulation model is given by

$$\begin{array}{rcl}
L & = & \mathit{producer} \parallel \mathit{consumer} \\
\mathit{producer} & = & \mathbf{seq}\ (i = 1, N)\ \{\mathbf{delay}(\tau_p)\ ;\ \mathit{put}\} \\
\mathit{consumer} & = & \mathbf{seq}\ (i = 1, N)\ \{\mathit{get}\ ;\ \mathbf{delay}(\tau_c)\} \\
\mathit{put} & = & \mathbf{P}(\mathit{room})\ ;\ <\text{store in buffer}>\ ;\ \mathbf{V}(\mathit{data}) \\
\mathit{get} & = & \mathbf{P}(\mathit{data})\ ;\ <\text{load from buffer}>\ ;\ \mathbf{V}(\mathit{room})
\end{array}$$

where the actual buffer access is assumed to be mutually exclusive (discussed below). The initial value of the resources *room* and *data* are B (available empty cells) and 0 (available full cells), respectively. □

The above example shows a case where resources are acquired and released by *different* processes. In most cases, however, resource acquisition and release is associated with the

[1]The multiset approach is similar to the approach used by Peterson to describe multiplicity in Petri nets [117].

same process in order to temporarily obtain exclusive access. For instance, a simple model of the shared buffer access in the above example is given by

$$\mathbf{P}(b) \; ; \; < \text{update buffer variables} > \; ; \; \mathbf{V}(b)$$

where b models the buffer resource (initially, $b = 1$) that needs to be accessed exclusively in order to maintain data integrity. Apart from the fact that resource access is often performed in the context of the one process, the essential performance aspect is that mutual exclusion is associated with *time delay* (otherwise, there would not be any reason to include mutual exclusion in performance models, except for maintaining data integrity). Thus, the principal model of resource access is given by the following "template"

$$access = \mathbf{P}(r) \; ; \; \mathbf{delay}(\tau) \; ; \; \mathbf{V}(r)$$

where τ accounts for the total time spent using resource r. For example, for $r = 1$ the model $L = access \parallel access \parallel access$ yields $T = 3\tau$ due to mutual exclusion of the three accesses. The frequent occurrence of the above template corresponds to the fact that each time delay can be associated with the use of some resource (multiset), like, e.g., a CPU, a memory (disk), a communication link, or simply some critical software section (e.g., file server). This is reflected in terms of Pamela by the **use** construct. Let U be defined as earlier. The Pamela model

$$L = \mathbf{use}(U, \tau)$$

specifies a process that exclusively (and instantaneously) acquires the multiset of resources specified by U for τ time units. Note that the **use** concept is similar to the (modular) server concept used in queuing networks. In the sequel we will often refer to resources as servers and vice versa.

In its basic definition, the **use** construct is equivalent to

$$\mathbf{use}(U, \tau) = \mathbf{P}(U) \; ; \; \mathbf{delay}(\tau) \; ; \; \mathbf{V}(U)$$

Hence, the scheduling discipline is FCFS (FIFO semaphores) with non-deterministic (fair) conflict arbitration. Note, however, that other disciplines can be modeled by explicitly modeling resource queues with some user-defined ordering in conjunction with the basic mutual exclusion mechanism provided by the **P** and **V** operators.

Let s denote an FCFS resource. Thus far, a $\mathbf{use}(s, \tau)$ operation is assumed to be associated with *one* service visit, i.e., the duration τ equals the basic service time. However, in many situations (i.e., at aggregate modeling level) the service demand will be a large multiple of the service time. Let τ_s denote the service time. Although this situation can be easily expressed by

$$\mathbf{seq} \; (i = 1, \tau/\tau_s) \; \mathbf{use}(s, \tau_s)$$

(where $\tau_s | \tau$), it is more convenient to be able to associate a service time with each resource such as, for instance, in queuing theory. In that way, without loss of information we can still write

$$\mathbf{use}(s, \tau)$$

By default, the operation will be interpreted as pure FCFS with $\tau_s = \tau$. However, if a service time $\tau_s < \tau$ is associated with s, the semantics of the operation is equal to the above $\textbf{use}(s, \tau_s)$ sequence. Although, from a modeling perspective the definition of a service time is merely a matter of convenience, during the analysis in Section 3.4 for cases where the resource multiplicity is larger than 1, it is advantageous to have information on the actual number of visit counts, rather than just the total service amount. However, in the sequel we will always assume a non-preemptive FCFS interpretation, unless noted otherwise.

One particular discipline next to FCFS that we will explicitly consider is processor sharing (PS), that is effectively an application of the FCFS model with associated service time definition $\tau_s \rightarrow 0$. Let s be a PS type resource and let τ denote the service demand. Based on the convention for resources with defined service time as described earlier, we can write

$$\textbf{use}(s, \tau)$$

instead of having to specify an (infinite) sequence. While the additional definition of a basic service time covers the whole spectrum, in typical modeling practice, we will only consider resources in terms of a default FCFS discipline (non-preemptive, $\tau_s = \tau$) and a PS discipline ($\tau_s \rightarrow 0$). Both types of resource usage are conveniently addressed by the same **use** operation that simply specifies the (aggregate) service demand. Although in the basic lower bound analysis technique we introduce in Section 3.4 the specific resource type is not of much influence, there are situations in which knowledge of the fact that resource access is "sliced" (large visit counts) rather than non-preemptive (one visit), can considerably improve the analysis accuracy that can be obtained. This subject will be treated in Section 5.4.

Example 3.3 In this example we show how the **use** operator is applied. Consider the dining philosophers problem [37]. Let N denote the number of think/eat cycles each philosopher performs. The Pamela solution is given by

$$\begin{aligned}
L &= \textbf{par}\ (p = 1, 5)\ philosopher(p) \\
philosopher(p) &= \textbf{seq}\ (i = 1, N)\ \{think(p)\ ;\ eat(p)\} \\
think(p) &= \textbf{delay}(\tau_t) \\
eat(p) &= \textbf{use}(\{c_p, c_{(p+1)\mathrm{mod}5}\}, \tau_e)
\end{aligned}$$

where the (FCFS-type) resources $c_1, \ldots c_5$ represents the five chopsticks. Note that the above model introduces the notion of time in contrast to the classical problem that only addresses the issue of concurrency and process synchronization. □

Considering the fundamental role of the **use** operation in modeling time delays as discussed earlier, like in queuing networks, it is appropriate to (re)define the **delay** operation in terms of an infinite-server ρ where $\rho = \infty$ according to

$$\textbf{delay}(\tau) = \textbf{use}(\rho, \tau)$$

Thus, every time delay can be expressed in terms of a **use** operation.

While the **use** operator provides a basic mutual exclusion mechanism, in order to model systems in which resource usage is *nested* PAMELA offers a generalization of the **use** operation in the form of the **using** construct according to

$$\textbf{using } (s)\ L$$

Its syntax resembles the PASCAL **with** construct in the sense that all statements within its scope (i.e., L) are executed under the condition that the specified resource (s) is acquired. In order to avoid various interpretation problems, the construct is only defined (and used) for FCFS scheduling according to

$$\textbf{using } (s)\ L = \mathbf{P}(s)\ ;\ L\ ;\ \mathbf{V}(s)$$

where s is of FCFS type. It follows $\textbf{use}(U, \tau) = \textbf{using } (U)\ \textbf{delay}(\tau)$. Note that nesting resource usage is different (and less powerful) than simultaneous resource possession. For example, $\textbf{using } (r_1)\ \textbf{using } (r_2)\ \textbf{delay}(\tau)$ is not equal to $\textbf{using } (\{r_1, r_2\}\ \textbf{delay}(\tau)$, that, in turn, is equivalent to $\textbf{use}(\{r_1, r_2\}, \tau)$.

Apart from its evident modeling convenience, the important reason to express mutual exclusion in terms of **use** (and **using**) constructs instead of the underlying **P** and **V** operations is that the additional structure in the model as imposed by the constructs allows us to define a simple calculus that approximates the effects of mutual exclusion. The use of **P** and **V** operators would necessitate the (compile-time) recognition of **use**-like access templates in order to deduce the work loads in terms of the resources involved. A similar point with respect to procedure-oriented modeling instead of message-oriented modeling is made in Section 3.3. Furthermore, as explained in Section 3.2, providing **wait/signal** operators rather than using semaphores is also motivated by analytic reasons. While a language definition *without* **P/V** operators would yield a too serious limitation with regard to modeling power, one of the underlying theses of this work is that their need in performance modeling of parallel computer systems, and the consequent loss of model analyzability that they introduce, is limited to a small class of problems. This point will be further addressed in Section 3.4.

We end this section by an example that forms a typical demonstration of the modeling approach in PAMELA.

Example 3.4 A classical example in performance modeling is a machine repair model (MRM) [91] in which P clients either spend a mean time τ_l on local processing, or request service from a server s ($s = 1$), with mean service time τ_s, with a total cycle count of N iterations (unlike the steady-state analysis of e.g., queuing systems or Petri nets, in our approach we require models to terminate). Both times are assumed to be exponentially distributed (the implementation through a call to some random generator function is ignored for simplicity). The PAMELA model of the MRM is given by

```
L = par (p = 1, P)
        seq (i = 1, N) {
            delay($\tau_l$);
            use($s, \tau_s$)
        }
```

in which the exclusive service is expressed by the **use** operation applied to the passive resource s that represents the server. Note that the above mathematical expression L is displayed in program format, including the usual indentation. Figure 3.2 shows the task graph as well as the execution trace of the MRM. In the figure deterministic times are assumed. Furthermore, the processes are sorted according to ascending rank (round robin scheduling is assumed). The (**use**) tasks that are mutually exclusive are shaded

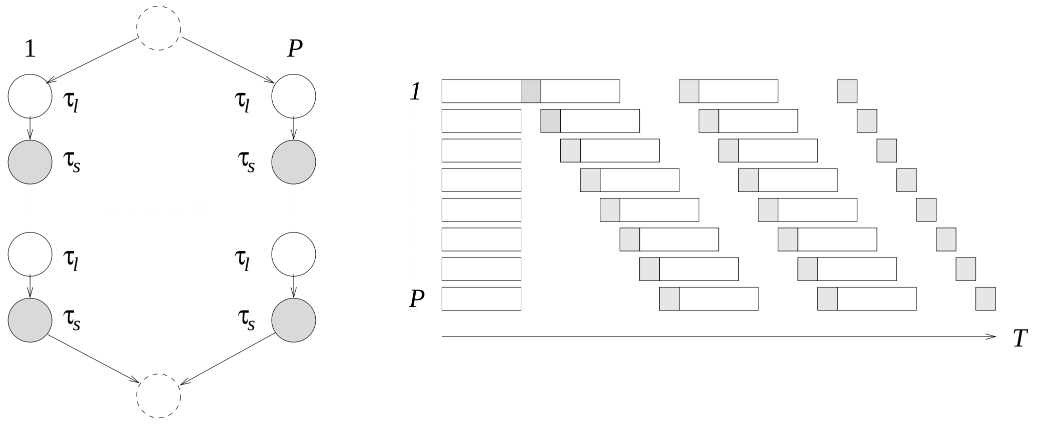


Figure 3.2: MRM graph and execution trace (deterministic time version)

(recall that task graphs cannot express mutual exclusion). Note that T is dominated by the contention for s (in fact, $T = \mathcal{O}(PN)$). The static analysis we present accounts for this effect whereas traditional static analysis only accounts for the critical path due to the task precedences ($T = \mathcal{O}(N)$). □

3.3 Paradigm

In this section we provide a rationale for the choice of a procedure-oriented language paradigm in Pamela. In systems modeling two basic modeling approaches may be distinguished, i.e., *material-oriented* and *machine-oriented* modeling [89]. The terminology originates from modeling and simulation in the industrial environment where material is processed by several machines in sequence according to some manufacturing process. In material-oriented modeling each material is associated a ("client") process (i.e., the manufacturing process) that describes the propagation of the material along the various machines, whereas in machine-oriented modeling each machine is associated a ("server") process, that accepts, processes, and delivers material within a chain formed by all machines. In terms of concurrent programming paradigms, material-oriented modeling has a natural correspondence to *procedure-oriented* programming [8], whereas machine-oriented modeling has a natural correspondence to the *message-oriented* programming paradigm [8]. Note, however, that either concurrent programming paradigm can be used to implement either modeling approach.

In order to enable compile-time analysis, the natural approach to performance modeling chosen in the Pamela methodology is *material-oriented*, hence the choice for a procedure-oriented language definition. This approach is demonstrated in Example 3.4 (MRM), in which the server is modeled as a passive resource S, to be held for duration τ_s by each client process. In a machine-oriented approach, the server would be modeled by a separate process that would communicate with the client processes through message-passing.

The choice between both modeling paradigms touches upon the fundamental issue of ease of *modeling* versus ease of (subsequent) *analysis*. For example consider the MRM. In a machine-oriented paradigm, both clients and server would map to processes that would communicate (and synchronize) using message-passing constructs. Let us assume a message-oriented version of Pamela based on the use of a CSP-like scheme using synchronous **send** and **receive** operators combined with a selective communication statement. Let C_p, $p = 1, \ldots, P$ denote the P client processes and let S denote the server. The MRM is modeled according to

```
$L =$  $S \parallel$ par $(p = 1, P)$ $C_p$
$C_p =$ seq $(i = 1, N)$ {
            delay($\tau_l$);
            send($S$);
            receive($S$)
        }
$S =$  while (true) {
            receive($L_1$)  $\rightarrow$ delay($\tau_s$); send($L_1$) $\Box$
            receive($L_2$)  $\rightarrow$ delay($\tau_s$); send($L_2$) $\Box$
              ...
            receive($L_P$)  $\rightarrow$ delay($\tau_s$); send($L_P$)
        }
```

By the way, note that in this model the mutual exclusion (implicitly) results from the single thread of control within the server while the non-determinism results from the '$\Box$' operator.

From a simulation software engineering point of view it might be advantageous to adopt the machine-oriented paradigm because of its similarity with the object-oriented (machine-oriented) approach taken by most model builders. However, the above approach is less amenable to compile-time analysis, both with respect to the analysis of condition synchronization as well as to the analysis of mutual exclusion. Unlike the material-oriented model the analysis of condition synchronization is complicated because of the non-determinism introduced by the message-passing mechanism (as a result of the mutual exclusion involved). For instance, a critical path analysis technique is impossible as the condition synchronization in above machine-oriented model cannot be expressed in terms of a task graph (it is unknown in which order the tasks actually synchronize). In other words, it may be impossible to deduce the "thread of condition synchronization" that now dynamically passes between *different* processes, whereas in a material-oriented model this thread would coincide with the process's own thread of control, thus enabling a much more simple, symbolic analysis scheme (a striking example of this important aspect is discussed in Section 5.5). In fact, by localizing information (comparable to "information hiding" as proposed in software engineering) the information on the *global* synchronization structure has been lost[2] (demonstrated in Example 3.6 later on).

While the above applies to condition synchronization the material-oriented paradigm also provides better analyzability with regard to mutual exclusion. In fact, unlike message-passing, the **use** construct forces the user to model according to a *structured* (operation-oriented [8]) paradigm that would be characterized by the template

$$\mathbf{use}(s, \tau) = \mathbf{send}(s) \ ; \ \mathbf{receive}(s)$$

where s is given by the above server process. Note that a similar observation has already been made earlier with regard to the basic "asynchronous message-passing" operators **P** and **V**. In some sense the situation is comparable with the use of unstructured **goto**s in sequential languages and the resulting problems with respect to program analyzability.

In summary, the global as well as the structured approach to describing synchronization in the material-oriented paradigm offers the possibility of a symbolic analysis scheme. As it is highly doubtful that recognition of a machine-oriented model in terms of an equivalent material-oriented model (reverse engineering) can be entirely mechanized, we adopt the material-oriented approach, that, at the possible expense of somewhat more modeling effort, retains the possibility of applying an automated mapping scheme yielding a symbolic performance model in the time domain.

Perhaps even more than the MRM, a pipeline is a typical example to demonstrate the merit of the material-oriented modeling approach.

Example 3.5 Consider the pipelined processing of N data sets involving an M unit pipeline (e.g., vector unit, packet-switched communication pipeline, software pipeline). In a machine-oriented paradigm, each unit would map to a process that would synchronously receive a data set, process it, and send it to the next unit. In our material-oriented approach, the entire computational process (involving M stages) is expressed for each data set. The result is a *contention model* in which each data process is executed in parallel and contends for each unit in its course. The PAMELA model is given by

$$L = \mathbf{par}\ (i = 1, N)\ \mathbf{seq}\ (m = 1, M)\ \mathbf{use}(u_m, \tau_m)$$

where u_m denotes the resource that represents unit m, and τ_m denotes the associated processing time. The above model correctly predicts both startup delay as well as the bandwidth of the pipelined system. Note that, although the absolute order in which data is processed is left undetermined, the *performance* prediction is valid.

Note that the material-oriented approach in which the pipeline is expressed in terms of a contention model yields *an SP model* that is amenable to the symbolic analysis method as will be described later on. If a task graph formalism would be used the processing of each data element by each unit would have to be expressed at the expense of a non-SP, $N \times M$ task graph as shown in Fig. 3.3, that, unlike the SP graph, is not amenable to the symbolic mapping process. Although the non-SP model can be mapped to a numeric time domain computation (in PAMELA terms, **wait**/**signal** operators can be

[2]The problem is more or less comparable to the parallelism detection problem with sequential programming languages, that, in general, cannot be solved at compile-time due to the irreversible loss of information.

used instead of **send/receive** operators, see, e.g., Example 5.1), the generated process is still essentially numeric in contrast to the symbolic model that results from the material-oriented approach.

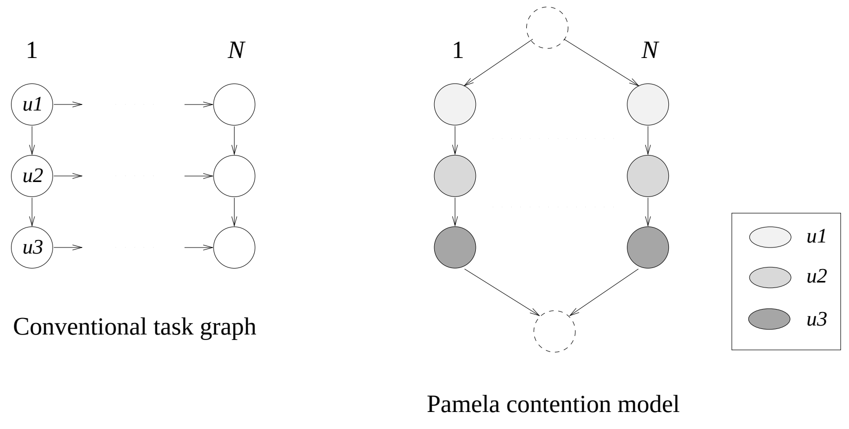


Figure 3.3: Traditional DG of a 3-stage pipeline compared to Pamela graph.

Compared to a machine-oriented solution, a consequence of the material-oriented approach is that more processes may be involved than absolutely necessary as in a typical pipeline $N \gg M$. In fact, contention models express *potential parallelism* (N) instead of the *actual parallelism* $(\min(N, M))$. Note, however, that given our analytical approach the use of a possibly huge number of processes does not induce actual (simulation) costs. For instance, there is no reason not to specify a 10^6-element vector operation using 10^6 "virtual" processes. □

At first glance, the use of the material-oriented paradigm with its structured synchronization operators (which we coin *contention modeling*) may seem to be restricted to systems that perform simple, and highly structured synchronizations. Consequently, a typically "message-oriented" problem such as the producer-consumer problem (cf. Example 3.2) might seem less amenable to a description in terms of a contention model. Yet again, a material-oriented description is possible, as shown in the next example.

Example 3.6 Recall the producer-consumer scheme shown in Example 3.2. In contrast to the machine-oriented flavor of the first solution[3] we now describe the propagation process of the data. In order to obtain a terminating system we define N to be the number of data elements that are processed. If we define the result to be the execution time it takes for the combined system to terminate, it follows

$$\begin{aligned}
L &= \mathbf{par}\ (i = 1, N)\ \{produced\ ;\ buffered\ ;\ consumed\} \\
produced &= \mathbf{use}(producer, \tau_p) \\
consumed &= \mathbf{use}(consumer, \tau_c) \\
buffered &= \mathbf{use}(buffer, \tau_b)
\end{aligned}$$

[3]Of course, a *pure* machine-oriented solution would involve *three* processes, i.e., *producer*, *consumer*, and *buffer* that would communicate using **send/receive** primitives. Nevertheless, although expressed in terms of our procedure-oriented formalism ("**P/V**"), the modeling paradigm was (largely) *machine*-oriented.

where τ_b accounts for the time delay involved with the buffer storage and retrieval of each data element (not specified in the earlier example). □

As in the pipeline example, the contention model accurately accounts for the overall time behavior of the process, while the model can be easily mapped into a symbolic time domain model, unlike the machine-oriented model. Again, the sacrifice is "under-specification", namely the abstraction of the *order* with respect to the actual data being processed, as well as the exact *location* where the data resides. Assuming the bandwidths of the various process stages differ, due to the infinite resource queues, all virtual processes will be queued at the slowest resource, instead of being spread across the total system due to the bounded storage capacity of each stage as in reality (e.g., the bounded buffer). Thus, contention modeling is a good example of the trade-off between obtaining precise knowledge of the timing and location of each individual element with the associated analysis cost, versus obtaining global, system-level timing information only, yet at a much lower expense.

3.4 Analysis

3.4.1 Introduction

In this section we present a basic calculus that enables us to reason about the temporal behavior of Pamela models. As mentioned earlier, the approach towards the analysis of Pamela models is based on the application of critical path analysis, extended with a bounding analysis to approximate the effects of mutual exclusion. As the latter approach is based on identifying potential serialization of contending model components, the analysis has been coined serialization analysis.

Apart from providing transformation rules from the Pamela domain to the time domain, the calculus enables model optimizations based on equivalence relations between models in the Pamela domain that have the same time solution. Some very simple examples have already been presented during the description of the Pamela language.

Recall the model

$$L = \mathbf{delay}(\tau_1) \; ; \; \mathbf{delay}(\tau_2)$$

where τ_1 and τ_2 are deterministic variables. Clearly, it follows $T = \tau_1 + \tau_2$. On the other hand, it also holds

$$\mathbf{delay}(\tau_1) \; ; \; \mathbf{delay}(\tau_2) = \mathbf{delay}(\tau_1 + \tau_2)$$

of which the right hand side also yields $T = \tau_1 + \tau_2$. Likewise the model

$$L = \mathbf{delay}(\tau_1) \parallel \mathbf{delay}(\tau_2)$$

immediately yields $T = \max(\tau_1, \tau_2)$ due to the implicit barrier synchronization. Consequently,

$$\mathbf{delay}(\tau_1) \parallel \mathbf{delay}(\tau_2) = \mathbf{delay}(\tau_1 \max \tau_2)$$

Consequently, the calculus comprises a mixture of mapping descriptions from the Pamela domain to the time domain as well as transformations within the Pamela domain. As

PAMELA models, based on **delay**s only, can be expressed in terms of task graphs, a critical path analysis scheme can be defined that maps a so-called *contention-free* (**delay**) model into a deterministic time domain computation. This will be described in the next section.

As an introduction to the mechanics of contention analysis, consider the PAMELA model

$$L = \mathbf{use}(r, \tau_1) \parallel \mathbf{use}(r, \tau_2)$$

where r is FCFS-type (initially, $r = 1$). In contrast to the above parallel composition this model yields $T = \tau_1 + \tau_2$ due to the *serialization* of both **use** statements. Even though the outcome of the above model can still be expressed in terms of a single solution, the introduction of mutual exclusion next to condition synchronization entails a new analysis problem due to the inherent non-determinism of the conflict arbitration. For instance, the analysis of the model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r, \tau_1) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \tau_3) \ ; \ \mathbf{delay}(\tau_4) \end{aligned}$$

is much less trivial. Depending on which process is given priority the outcome is either $T = \tau_1 + \max(\tau_3 + \tau_4, \tau_2)$ or $T = \tau_3 + \max(\tau_1 + \tau_2, \tau_4)$. The non-deterministic *direction* of the implicit precedence arc between both **use** statements introduces the uncertainty in the prediction. Recall that we assume for every FCFS resource that the service time equals the service demand argument of each **use** operation unless stated otherwise. Hence, the variance of T for the above model can be quite large (up to a factor 2 as will be discussed in Chapter 6). On the other extreme, for PS-type resources (service time $\rightarrow 0$) the variance of the result would go to zero in which case a deterministic result is obtained. (It holds $T = 2\min(\tau_1, \tau_3) + \max(\tau_2 + \tau_1 - \tau_3, \tau_2, \tau_4 + \tau_3 - \tau_1, \tau_4)$ as explained in Section 5.4.) In general, however, the result is essentially non-deterministic. The analysis of PS-type resource usage is discussed in Section 3.6. While this analysis involves additional transformations that apply to PS-type resources only, the basic calculus we present in this chapter covers any scheduling discipline. Hence, in the following we tacitly assume FCFS-type resources unless specifically noted.

In general, large models comprise many **use** statements involving many resources. Even worse, for aggregate and possibly dynamic models, the exact relative *order* of the various **use** statements is typically unknown at compile-time. Clearly, unlike critical path analysis, there is no simple mechanical process through which L can be mapped onto T other than through enumeration of all possible critical paths, depending on the actual τ values and priority schemes. In general, the execution time T of a PAMELA model may be anywhere between a lower bound T^l and some upper bound T^u. Note that this uncertainty is entirely due to the presence (or potential) of contention. At this point we do not (yet) consider conditional control flow which, of course, forms an additional complication.

The characteristic approach in probabilistic models like queuing networks, Petri nets, or process algebras is to analyze the entire (state) distribution or, for efficiency reasons, merely to consider the *mean* value of T. The approach we will take is to select T^l as an estimate for T. The reason for this choice is as follows. First, the analysis of a (tight) lower bound is trivial and, equally importantly, it is amenable to a mechanical, symbolic, scheme like conventional critical path analysis. Second, as will be shown in Section 6.2, the analysis of an upper bound that is *tight* (i.e., has any practical value) is extremely complex. Third, as will be shown in Section 6.3, for the vast majority of systems with random resource usage (i.e., a typical system) the mean value of T is much closer to T^l than T^u. The upper bound corresponds to extremely unlikely schedules (unfair resource conflict arbitrations) whereas the vast majority of schedules entails execution times close to T^l. Hence, the lower bound is a much better estimate of T than T^u.

Our general approach towards the analysis of PAMELA models is characterized by Figure 3.4. For the subset of contention-free models T can be computed based on the

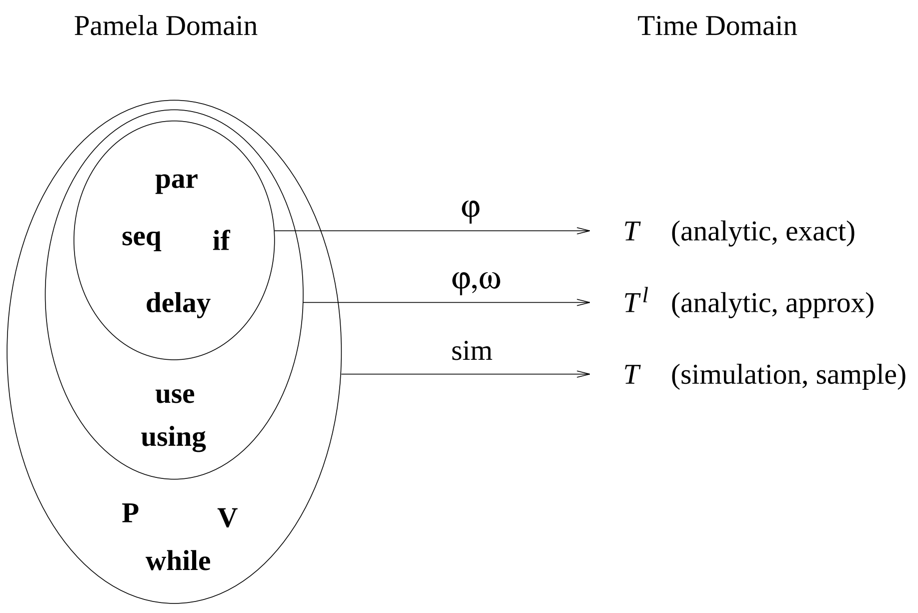


Figure 3.4: PAMELA analysis approach

isomorphism φ. For a subset of contention models (i.e., those having structured ME and no data dependencies) the lower bound analysis technique applies (T^l). For the superset simulation is the only solution, be it that parts of the model that are within the above subsets can be replaced by reduced versions using the appropriate analytic technique. Hence, significant savings in simulation time can be achieved at small (or without) sacrificing prediction accuracy. With respect to the **if** operator within the smallest set, note that this only applies to simple conditions that are static. For many applications, however, the **if** operator should be thought as being in one of the supersets. This point will be discussed later on.

In the following we first describe the critical path component of the calculus that applies for models that are contention-free. After that we will describe our approach to the analysis of mutual exclusion and how it is integrated within the critical path analysis.

3.4.2 Critical Path Analysis

In the following we formalize critical path analysis in terms of PAMELA. First we consider models without conditional control flow. As described in Chapter 2, for task graphs, critical path analysis is based on the isomorphism[4] between the task graph in the original computation domain and the task graph representing the time domain computation. As contention-free Pamela models can be represented by task graphs the isomorphism also applies to Pamela models. Let φ denote the isomorphism between a computation L in the Pamela domain and the corresponding (execution) computation T in the time domain. Then $T = \varphi(L)$ denotes the estimated execution time of L. Based on the task graph isomorphism described in Chapter 2, T is constructed by applying the following transformations.

Let g_i denote a basic Pamela statement, i.e., a **delay** a **wait** or a **signal** statement. Let $g_{p(i)}$ denote its predecessor statement (except for the very first statement). The following time computations are generated:

- identity:

$$\textbf{delay}(\tau) \rightarrow r_i = r_{p(i)} + \tau \tag{3.1}$$

- condition synchronization:

$$\textbf{signal}(\{c_1, \dots, c_N\}) \rightarrow r_i = r_{p(i)} \quad , \forall i = 1 \dots N : r'_{ci} = r_i \tag{3.2}$$

$$\textbf{wait}(\{c_1, \dots, c_N\}) \rightarrow r_i = \max(r_{p(i)}, \max_{i=1 \dots N} r'_{ci})) \tag{3.3}$$

Note that each variable is assigned exactly once. Instead of implementing the **wait**/**signal** time synchronization in terms of the **wait** task variable r_i, a condition-specific variable r'_c is used. The reason for using this modified scheme is its practical value in that a computable result is still achieved in cases where the name of the predecessor task is not available in explicit form (see Example 5.1 that involves the critical path analysis of conditional message-passing code).

The transformation of a **delay** statement can also be expressed in terms of φ according to

$$\varphi(\textbf{delay}(\tau)) = \tau \tag{3.4}$$

The transformation of the sequential and parallel composition operators are given by

- sequentialism:

$$\varphi(L_1 \; ; \; L_2) = \varphi(L_1) + \varphi(L_2) \tag{3.5}$$

- parallelism:

$$\varphi(L_1 \parallel L_2) = \varphi(L_1) \max \varphi(L_2) \tag{3.6}$$

[4] Although all static analysis techniques are based on this isomorphism a comparable algebraic description has only been explicitly introduced in [106].

These transformations generalize to the following reductions

$$\varphi(\textbf{seq}\ (i = 1, N)\ L_i) = \sum_{i=1}^{N} \varphi(L_i) \tag{3.7}$$

$$\varphi(\textbf{par}\ (i = 1, N)\ L_i) = \max_{i=1 \ldots N} \varphi(L_i) \tag{3.8}$$

For example, consider the Pamela model of the parallelized polynomial computation according to Example 3.1. Each equation of the Pamela model is compiled into the following computations (after some simplifying local substitutions)

$$\begin{aligned}
T &= \max(r_1, r_2, \ldots, r_8) \\
r_1 &= 0 + \tau_m \quad , \ r'_{13} = r_1 \quad , \ r'_{14} = r_1 \\
r_2 &= 0 + \tau_m \quad , \ r'_{25} = r_2 \\
r_3 &= \max(0, r'_{13}) + \tau_m \quad , \ r'_{36} = r_3 \\
r_4 &= \max(0, r'_{14}) + \tau_m \quad , \ r'_{47} = r_4 \\
r_5 &= \max(0, r'_{25}) + \tau_a \quad , \ r'_{57} = r_5 \\
r_6 &= \max(0, r'_{36}) + \tau_m \quad , \ r'_{68} = r_6 \\
r_7 &= \max(0, r'_{47}, r'_{57}) + \tau_a \quad , \ r'_{78} = r_7 \\
r_8 &= \max(0, r'_{68}, r'_{78}) + \tau_a
\end{aligned}$$

Due to the static precedence relations, the computation of T is straightforward. The order in which each equation is to be evaluated on a (presumably) sequential system can be determined using compile-time dependence analysis. For models in which the task precedence relations are dynamic the above compilation scheme also applies. In that case, the above equations must be embedded within an additional iterative loop structure that evaluates the data dependencies at run-time[5].

Due to the isomorphism, by Eq. (3.4) through (3.6) the following transformations are defined as well

$$\begin{aligned}
\textbf{delay}(\tau_1)\ ;\ \textbf{delay}(\tau_2) &= \textbf{delay}(\tau_1 + \tau_2) \\
\textbf{delay}(\tau_1)\ \|\ \textbf{delay}(\tau_2) &= \textbf{delay}(\tau_1 \max \tau_2)
\end{aligned}$$

that form the basis for SP reduction. For example, consider the following SP model

$$\begin{aligned}
L &= L_1 \parallel L_2 \\
L_1 &= \textbf{delay}(\tau_1)\ ;\ \textbf{delay}(\tau_2) \\
L_2 &= \textbf{delay}(\tau_3)\ ;\ \textbf{delay}(\tau_4)
\end{aligned}$$

Application of φ (Eqs. (3.4),through (3.6)) results in

$$\begin{aligned}
T &= T_1 \max T_2 \\
T_1 &= \tau_1 + \tau_2 \\
T_2 &= \tau_3 + \tau_4
\end{aligned}$$

[5]See Example 5.1. Note that this is similar to maintaining a discrete event list in a simulation approach. The relation between critical path analysis and simulation is elaborated in Section 5.4.

that yields $T = ((\tau_1 + \tau_2) \max(\tau_3 + \tau_4))$. On the other hand, SP reduction yields the same result, be it in the PAMELA domain according to the derivation

$$\begin{aligned} L_1 &= \textbf{delay}(\tau_1 + \tau_2) \\ L_2 &= \textbf{delay}(\tau_3 + \tau_4) \\ L &= L_1 \parallel L_2 = \textbf{delay}((\tau_1 + \tau_2) \max(\tau_3 + \tau_4)) \end{aligned}$$

As discussed in Chapter 2, conditional control flow is usually handled by weighting the workloads in the model by the (combined) branching probability associated with the branch in which the statement resides. Formally, conditional statements can be incorporated in the compilation scheme by simply transferring the condition to the time domain according to

$$\varphi(\textbf{if } (c) \; L) = [c]\varphi(L) \tag{3.9}$$

where $[\ldots] : \{true, false\} \to \{0, 1\}$ denotes Iverson's operator [75] defined by

$$[c] = \begin{cases} 1, & \text{c is true;} \\ 0, & \text{c is false.} \end{cases}$$

Hence, the following transformation holds

$$\textbf{if } (c) \; \textbf{delay}(\tau) = \textbf{delay}([c]\tau)$$

Clearly, the conditional construct(s) will eventually have to be reduced in order to avoid a functional simulation. For some parameters of interest, however, Eq. (3.9) provides a means of retaining them within the resulting performance model. For instance, consider

$$L = \textbf{seq } (i = 1, N) \; \textbf{if } (i \bmod S) \; \textbf{delay}(\tau)$$

is compiled into

$$T = \sum_{i=1}^{N} [i \bmod S]\tau$$

in which the value of S is clearly of interest to the model. Retaining parameters may also have a favorable effect on accuracy. In the above case, subsequent reduction of the $\sum$ - $[\ldots]$ pair yields

$$T = \lceil \frac{N}{S} \rceil \tau$$

However, in many cases, probabilistic reductions are used, which, in terms of our calculus is expressed as the following mean value expression

$$\mathrm{E}(\sum_{i=1}^{N} [c_i] T) = pT$$

where c_i denotes some, possibly i-dependent condition (e.g., resulting from a branch in a loop), and p denotes the average truth probability of $c_1, \ldots c_N$ (e.g., the average branching probability).

3.4.3 Lower Bound Analysis

In this section we introduce the approximate analysis of mutual exclusion and describe its integration within the critical path compilation scheme as described earlier. The analysis is restricted to simple **use** models, i.e., models in terms of $\mathbf{use}(U, \tau)$ where $|U| = 1$. The analysis of simultaneous resource possession will be discussed in Section 3.6.
Recall the model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r, \tau_1) \ ; \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \tau_3) \ ; \mathbf{delay}(\tau_4) \end{aligned}$$

where $r = 1$. As discussed before, in general the prediction for T is only known to lie between a lower bound T^l and an upper bound T^u due to the non-determinism involved with the potential of resource contention. In the following we describe the analysis of the lower bound.

On the one hand, the lower bound is determined by the fact that T cannot be less than the execution time of the same model L' with $r = \rho$ according to

$$\begin{aligned} L'_1 &= \mathbf{delay}(\tau_1) \ ; \mathbf{delay}(\tau_2) \\ L'_2 &= \mathbf{delay}(\tau_3) \ ; \mathbf{delay}(\tau_4) \end{aligned}$$

Consequently, $T \geq \varphi(L')$ which yields

$$T \geq max(\tau_1 + \tau_2, \tau_3 + \tau_4)$$

as shown earlier.

On the other hand, T cannot be less than the aggregate service demand $\tau_1 + \tau_3$ on the resource r corresponding to the fact that the **use** statements cannot overlap (i.e., are serialized). Consequently

$$T^l = max(\tau_1 + \tau_2, \tau_3 + \tau_4, \tau_1 + \tau_3)$$

When more resources are involved the serialization argument simply applies to each resource separately. Clearly, the above serialization analysis is amenable to a mechanized, symbolic scheme.

We now present the analysis algorithm. Let φ now be defined for *any* PAMELA model including **use** statements according to Eqs. (3.1) through (3.9) including

$$\varphi(\mathbf{use}(r, \tau)) = \tau \tag{3.10}$$

The effect of mutual exclusion is approximated by the following scheme. Let $\underline{\delta}(L) = (\delta_1, \ldots, \delta_M)$ denote the total service demand vector of L where M is the total number of resources involved and δ_m denotes the total service demand on resource r_m. For convenience we will write $\delta_m(L)$ to denote the m-th element of $\underline{\delta}(L)$. Clearly, the aggregate work load on each resource is given by

$$\underline{\delta}(L) = \begin{cases} \underline{\delta}(L_1) + \ldots + \underline{\delta}(L_N), & L = L_1 \ ; \ldots ; \ L_N \text{ or } L = L_1 \parallel \ldots \parallel L_N; \\ \tau \underline{e}^m, & L = \mathbf{use}(r_m, \tau). \end{cases} \tag{3.11}$$

where $\underline{e}^m = (0, \ldots, 0, 1, 0, \ldots, 0)$ is the M-dimensional unit vector in the m direction, and addition and multiplication are defined element-wise. Let ω denote the lower bound on the execution time of L due to the fact that each access to a resource is at least serialized. Assuming that the amount of request parallelism is larger than the appropriate resource multiplicity, for sufficiently large visit counts (e.g., PS-type resources) it follows

$$\omega(L) = \max_{m=1\ldots M} \frac{\delta_m(L)}{r_m} \tag{3.12}$$

Eq. (3.12) follows from the fact that the execution time can never less than the maximum of the total time delay at each resource (i.e., the total service demand divided by the server multiplicity). For cases where the number of visit counts is very low, when the basic service time τ_m is known, a somewhat tighter bound is established by

$$\omega(L) = \max_{m=1\ldots M} \left\lceil \frac{\nu_m(L)}{r_m} \right\rceil \tau_m, \quad \nu_m(L) = \frac{\delta_m(L)}{\tau_m} \tag{3.13}$$

where ν_m denotes the aggregate visit count of resource m. For instance, when the total service demand only entails $\nu = 3$ visits involving a resource with multiplicity $r = 2$, the minimum delay corresponds to 2 units service ($2\tau_m$), rather than 1.5 units service as would be computed by Eq. (3.12). When the ratio between the number of visits and resource multiplicity is large, Eq. (3.13) approaches Eq. (3.12).

Combining the lower bound due to mutual exclusion (ω) with the lower bound due to condition synchronization (φ), it follows that the lower bound on T is predicted by

$$T^l(L) = \max(\varphi(L), \omega(L)) \tag{3.14}$$

While Eq. (3.14) applies to basic parallel sections, for general models the following recursion provides a sharper bound as will be illustrated in Example 3.10.

$$T^l(L) = \begin{cases} T^l(L_1) + \ldots + T^l(L_N), & L = L_1 \;; \ldots ;\; L_N; \\ T^l(L_1) \max\ \ldots\ \max\ T^l(L_N) \max\ \omega(L), & L = L_1 \parallel \ldots \parallel L_N; \\ \max(\varphi(L), \omega(L)), & \text{otherwise.} \end{cases} \tag{3.15}$$

Note that conventional compile-time analysis disregards ω while conventional queuing analysis (partially) disregards φ. Serialization analysis combines both terms in an approximation that sustains a minimum accuracy where the aforementioned approaches fail. Like conventional critical path analysis, serialization analysis has a quadratic complexity in the (symbolic) size of the model, while for SP models the complexity is linear.

Similar to the critical path analysis, described earlier, the lower bound analysis can also be expressed in terms of the PAMELA domain, rather than the time domain. Unlike contention-free models, however, some transformations are approximations, rather than exact. This even applies to simple models. Consider a submodel

$$\mathbf{use}(r, \tau_1) \;;\; \mathbf{use}(r, \tau_2)$$

where $r = 1$. For FCFS-type resources the reduction

$$\mathbf{use}(r, \tau_1) \;;\; \mathbf{use}(r, \tau_2) = \mathbf{use}(r, \tau_1 + \tau_2)$$

is not entirely correct (for PS-type resources the reduction holds). Although, in terms of T^l both cases are equivalent, in terms of T a different distribution will result due to the fact that the state space has changed (in the original submodel a process can be preempted after τ_1 time units). Similarly, the reduction

$$\textbf{use}(r, \tau_1) \parallel \textbf{use}(r, \tau_2) = \textbf{use}(r, \tau_1 + \tau_2)$$

entails the same problem. Nevertheless, when the lower bound analysis is applied the above reductions hold. Even when T is considered, the errors that are introduced are typically small except for small models with coarse grain FCFS resource usage. The related accuracy aspects are extensively discussed in Chapter 6.

As the above reductions are correct within the lower bound analysis, we now present the remaining transformations needed to apply serialization analysis within the Pamela domain. Like in the case of contention-free models, conditional control flow is formally transferred to the time domain according to

$$\textbf{if}\ (c)\ \textbf{use}(r, \tau) = \textbf{use}(r, [c]\tau)$$

After all conditional constructs are rewritten accordingly, a static model results that is amenable to the application of Eq. (3.15). In the Pamela domain, Eq. (3.14) corresponds to the following transformation, denoted F.

Transformation 3.1 *Let*

$$L(r) = \textbf{par}\ (p = 1 \ldots P)\ L_p(r)$$

model a parallel section where $L_p(r)$ involves mutual exclusive access to some resource r up to a total service demand τ_p. Then its lower bound transformation *$F(L)$ is given by*

$$F(L) = \textbf{use}(r, \omega) \parallel L(\rho)\ ,\quad \omega = \frac{\sum_{p=1}^{P} \tau_p}{r_m}$$

□

Note that, instead of the isomorphism with respect to contention-free models, this transformation does *not* imply an equality. It denotes an approximation. In case of multiple resources the lower bound transformation successively applies to each resource, based on the fact that the contributions to T^l are independent.

The transformation factors out the serialization part from within the parallel section. The **use** statements within the original section are subsequently replaced with **delay**s (hence the ρ term) to avoid the introduction of redundant contention terms. For instance consider the example

$$L = \{\textbf{use}(r, \tau_1)\ ;\ \textbf{delay}(\tau_2)\} \parallel \{\textbf{use}(r, \tau_3)\ ;\ \textbf{delay}(\tau_4)\}$$

Application of Transformation 3.1 yields

$$L' = \{\textbf{use}(r, \tau_1)\ ;\ \textbf{use}(r, \tau_3)\} \parallel \{\textbf{delay}(\tau_1)\ ;\ \textbf{delay}(\tau_2)\} \parallel \{\textbf{delay}(\tau_3)\ ;\ \textbf{delay}(\tau_4)\}$$

that introduces the extra term in the lower bound formula derived earlier.

Corresponding to the recursive process of Eq. (3.15) the above transformation is embedded in the following algorithm.

Algorithm 3.1 *Let L be a contention model. The following* serialization algorithm A *transforms L into a contention-free model $L' = A(L)$ while preserving the lower bound on execution time as defined by serialization analysis.*

```
$A(L)$
{
    let $L = \mathbf{seq}\ (s = 1, S)\ L_s$
    if $(S > 1)$
        for $(s = 1 \ldots S)$
            $A(L_s)$
    let $L = \mathbf{par}\ (p = 1, P)\ L_p$
    if $(P > 1)$ {
        for $(p = 1 \ldots P)$
            $A(L_p)$
        $F(L)$
    }
}
```

□

3.5 Examples

In this section we present a number of examples that demonstrate how the analysis technique is applied.

Example 3.7 In this example we derive T^l for the MRM and compare it with results obtained through queuing theory. Recall the MRM in Example 3.4. It follows

$$\begin{aligned}\varphi &= \max_{p=1\ldots P} \sum_{i=1}^{N} (\tau_l + \tau_s) = N(\tau_l + \tau_s) \\ \omega &= \sum_{p=1}^{P} \sum_{i=1}^{N} \tau_s = PN\tau_s\end{aligned}$$

Hence, by Eq. (3.14) (or Eq. (3.15)) it follows $T^l = N \max(\tau_l + \tau_s, P\tau_s)$. Unlike conventional compile-time analysis T^l accounts for the additional queuing delay when s is saturated. The above analysis yields the same result as asymptotic bound analysis in queuing theory [93]. Let R denote the response time and let $Z = \tau_l$ denote the think time. Then the mean cycle time $R + Z$ equals φ/N for $P \ll P^*$ and ω/N for $P \gg P^*$, where the saturation point $P^* = (\tau_s + \tau_l)/\tau_s$ denotes the crossover between the asymptotes.

For deterministic time delays T^l provides a good estimation. However, even for stochastic time delays T^l serves as a reasonable estimator. For example, assume that τ_l and τ_s are exponentially distributed. While the result for T^l remains the same, the accurate (simulation) result T can be computed analytically. Since the MRM maps to a separable queuing network [15], Mean Value Analysis (MVA) [129] may be applied that yields (for large N)

$$T = N(R(P) + \tau_l)$$

where the response time $R(n)$ of the server for n clients in the closed system is given by the MVA recursion [93]

$$R(0) = 0\ ,\ \ R(n+1) = \left[1 + \frac{PR(n)}{\tau_l + R(n)}\right]\tau_s$$

Figure 3.5 compares the predictions T^l (dotted line) and T (straight line) as a function of the number of clients P. The figure shows that the lower bound essentially forms the

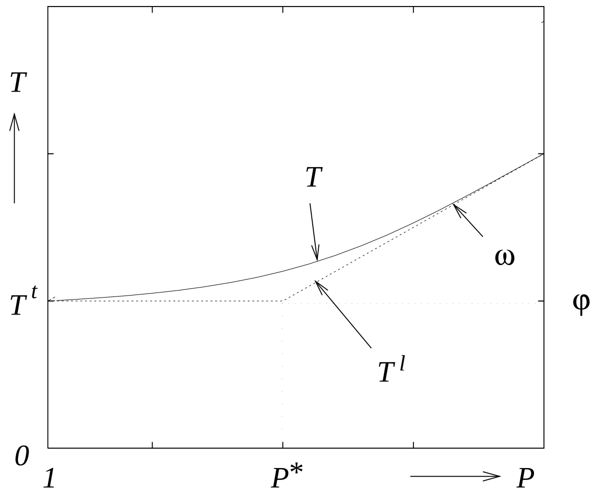


Figure 3.5: Lower bound T^l compared to MVA prediction T

asymptotes of the probabilistic prediction, with a limited deviation occurring at $P = P^*$ due to the small amount of contention that occurs in the stochastic system. Notice the dramatic error that traditional compile-time analysis ($T^t = \varphi$) entails for large P. The utility of T^l as an estimator for T is further discussed in Chapter 6. □

Example 3.8 In this example we derive T^l for the pipeline model and compare it to T. Consider the pipeline model discussed in Example 3.5 given by

$$L = \textbf{par}\ (i = 1, N)\ \textbf{seq}\ (m = 1, M)\ \textbf{use}(u_m, \tau_m)$$

Let a pipeline unit take τ_a on average, and let c denote the slowest pipeline unit. It follows

$$\varphi = M\tau_a,\quad \omega = N\tau_c$$

Thus Eq. (3.14) yields $T^l = \max(M\tau_a, N\tau_c)$. Indeed, T^l is a lower bound as the correct pipeline model is given by $T = M\tau_a + (N-1)\tau_c$. However, the relative deviation is negligible for cases where either φ dominates (startup term, $M \gg N$) or where ω dominates (bandwidth term, $N \gg M$). The maximum deviation occurs for a *balanced* system (i.e., equal resource demands, $\tau_c = \tau_a = \tau$) when $N = M$. In this case, $T^l = N\tau$ whereas $T = (2N-1)\tau$. Hence, the worst case relative deviation is a factor 2. Note, however, that this can only occur for balanced systems that are precisely in the cut-off region between no saturation and full saturation. Notice how the contention model and its associated analysis account for both pipeline startup (critical path term φ) and bandwidth (contention term ω). □

Both the MRM and the pipeline are examples in which T^l and T differ (up to a factor 2) due to the fact that the φ-ω analysis does not account for the transient *skewing* effect in which the processes incur a one-only delay as a result of the initial resource conflicts (see Fig. 3.2). An extension of the lower bound analysis method to accurately account for this general phenomenon will be described in Section 3.6. Due to the extension a correct result for the pipeline model can be achieved.

Example 3.9 As an example of the recursive operation of Algorithm 3.1, interleaved with SP reduction, consider the following SP model

$$\begin{aligned} L &= t_1 \;;\; (L_1 \parallel L_2) \;;\; t_{16} \\ L_1 &= t_2 \;;\; ((t_4 \;;\; t_7) \parallel (t_5 \;;\; t_8)) \;;\; t_{11} \;;\; t_{14} \\ L_2 &= t_3 \;;\; t_6 \;;\; ((t_9 \;;\; t_{12}) \parallel (t_{10} \;;\; t_{13})) \;;\; t_{15} \end{aligned}$$

where

$$t_{\{1,5,6,7,9,11,13,16\}} = \textbf{delay}(1),\ t_{\{3,14\}} = \textbf{delay}(2)$$

are delay tasks and

$$t_2 = \textbf{use}(r_1, 1),\ t_{\{10,12\}} = \textbf{use}(r_1, 5),\ t_{15} = \textbf{use}(r_2, 1),\ t_{\{4,8\}} = \textbf{use}(r_2, 5)$$

contend for resource r_1 or r_2 ($r_i = 1$). Figure 3.6 shows the initial task graph (annotated

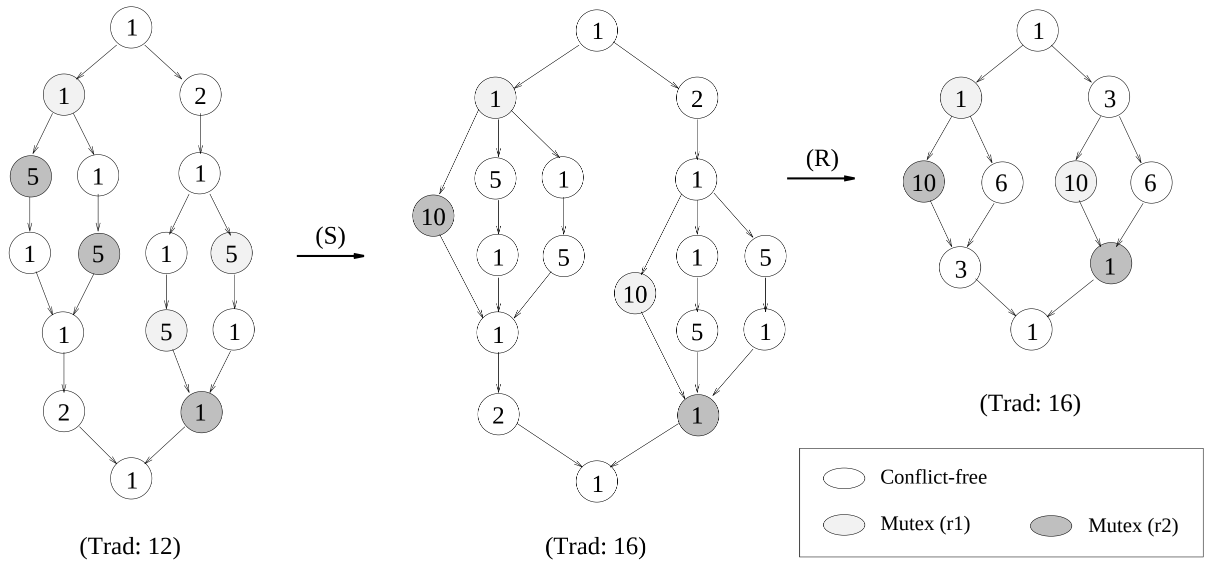


Figure 3.6: Serialization process at recursion depth 1

with the individual workloads, the tasks are numbered row-wise), the graph after the first two applications of Transformation 3.1 (S: serialization) at recursion depth 1, and after partial reduction (R: reduction). Figure 3.7 shows the graph after application of Transformation 3.1 (S) at the top level, after replacing **use** operations by **delay**s as there is no more potential for contention (ρ), and after final reduction (R). The numbers below

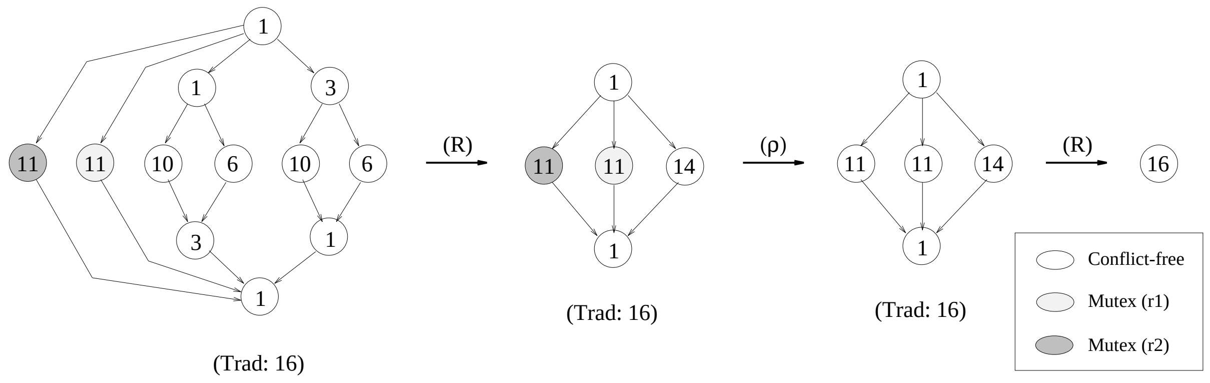


Figure 3.7: Serialization process at recursion depth 0

each graph denote the result of traditional analysis (φ). The example demonstrates that the complete recursion needs to be performed, rather than simply applying Transformation 3.1 just once, incorporating all **par** levels simultaneously. The latter would still yield $T^l = 12$, not accounting for the serialization effects that occur at **par** level 1. □

Example 3.10 In order to demonstrate the vital importance of recursively applying Eq. (3.14) (i.e., Eq. (3.15)), consider the following model, i.e.,

$$L = \textbf{seq}\ (i = 1, N)\ \textbf{par}\ (p = 1, P)\ \textbf{use}(r_i, \tau)$$

in which resource usage is non-uniformly distributed over the length of the entire computation (see Fig. 3.8). While Eq. (3.14) yields $T^l = \max(P\tau, N\tau)$, Eq. (3.15) yields

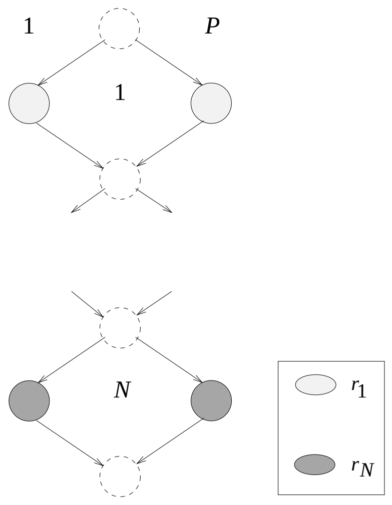


Figure 3.8: Nonuniform resource access

$T^l = \sum_{i=1}^{N} \max(P\tau, \tau) = NP\tau$. Thus applying Eq. (3.15) (i.e., applying Eq. (3.14) to *each* parallel section instead of only once) improves the bound by as much as a factor N. □

To conclude this section we present an example showing how (simple) algorithms are analyzed. For the sake of modeling we must now assume the presence of a *machine*. However, we will not consider machine modeling in more detail than necessary as this is the subject of Chapter 4. Furthermore, the example shows how conditional control flow is modeled.

Example 3.11 Consider the following SPMD [82] program that scales a sparse N element vector $\underline{v}$ using a simple (abstract) P-node scalar shared-memory machine, i.e.,

```
for i = 0 .. B(p)-1
    if v[f(p,i)] != 0
        v[f(p,i)] = v[f(p,i)] * alpha;
```

where $p = 0 \ldots P-1$ denotes the processor index and $B(p)$ and $f(p,i)$ denote local loop bound and index function, respectively (based on the specific partitioning scheme used). Let the machine (interface) be modeled in terms of the instructions `move` (shared memory load/store) and `flop` (local floating point operation, including register traffic). Note the similarity to the abstract parallel machine introduced in Chapter 1. Given this machine interface, the PAMELA model of the parallel program is given by the expression

```
$L = \textbf{par}\ (p = 0, P-1)$
        $\textbf{seq}\ (i = 0, B(p)-1)\ \{$
            $move(v + f(p,i));$
            $\textbf{if}\ (v[f(p,i)] \neq 0)\ \{$
                $flop;$
                $move(v + f(p,i))$
            $\}$
        $\}$
```

where the **par** section accounts for the SPMD execution. Many details like multiprocessing overhead are ignored for simplicity (`alpha` is assumed to be already loaded in register).

For this example we only consider an abstract machine model where a floating point is represented by a delay and shared-memory access is mutually exclusive according to

$$\begin{aligned} flop &= \textbf{delay}(\tau_f) \\ move(a) &= \textbf{use}(m, \tau_m) \end{aligned}$$

where τ_f denotes the effective floating point instruction time, m denotes the shared memory resource, and τ_m denotes the effective memory load/store time. Note that in this abstract architecture the actual memory address (a) is irrelevant (unlike many of the architectures we will discuss in the next chapter). Consequently, it follows

```
$L = \textbf{par}\ (p = 0, P-1)$
        $\textbf{seq}\ (i = 0, B(p)-1)\ \{$
            $\textbf{use}(\tau_m);$
            $\textbf{if}\ (c(p,i))\ \{$
                $\textbf{delay}(\tau_f);$
                $\textbf{use}(\tau_m)$
            $\}$
        $\}$
```

P	T^l/T	φ/T
5	0.96	0.96
10	0.87	0.87
15	0.89	0.72
20	0.92	0.59
25	0.95	0.46

Table 3.1: T^l without reduction.

P	T^l/T	φ/T
5	0.89	0.89
10	0.73	0.73
15	0.89	0.62
20	0.92	0.46
25	0.95	0.38

Table 3.2: T^l (reduction to $\mathcal{O}(1)$).

where $c(p,i) = (v[f(p,i)] \neq 0)$ denotes the data dependency.

Serialization analysis yields

$$\begin{aligned}\varphi &= \max_{p=0\ldots P-1} \sum_{i=0}^{B(p)-1} (\tau_m + [c(p,i)](\tau_m + \tau_f)) \\ \omega &= \sum_{p=0}^{P-1} \sum_{i=0}^{B(p)-1} (1 + [c(p,i)])\tau_m\end{aligned}$$

which yields the prediction $T^l = \max(\varphi, \omega)$.

For vectors with $N = 10^4$ and uniform density $d = 0.1$ Table 3.1 shows some results for different numbers of processors ($\tau_f = 100\tau_m$). As in the other examples T^l is reasonably accurate for cases with relatively high φ or ω. For $P \geq 15$ contention starts to dominate as can be seen by the values of φ (second column).

The above prediction has an $\mathcal{O}(N)$ complexity. In the interest of efficiency we consider two possible reductions of T^l. First, let the density of $\underline{v}$ be uniform and measured to be given by d. As a result of the reduction

$$\sum_{i=0}^{B(p)-1} [c(p,i)] = dB(p)$$

it follows

$$\begin{aligned}\varphi &= \max_{p=0\ldots P-1} B(p)(\tau_m + d(\tau_m + \tau_f)) \\ \omega &= PB(1+d)\tau_m = N(1+d)\tau_m\end{aligned}$$

which reduces the complexity of T^l from $\mathcal{O}(N)$ to $\mathcal{O}(P)$. Note that φ and ω now denote mean values. Furthermore, under the assumption of a standard block or cyclic partitioning scheme an $\mathcal{O}(1)$ complexity model results, i.e.,

$$T^l = \max(\left\lceil \frac{N}{P} \right\rceil (\tau_m + d(\tau_m + \tau_f)), N(1+d)\tau_m)$$

Note that the first reduction of φ to a mean value introduces an error as the real mean value be slightly higher due to the actual variance in the number of non-zeros that are processed by each node (the mean value offset is logarithmically in P [90]). This phenomenon is reflected by the lower φ/T values in Table 3.2 compared to Table 3.1 (again, for parameter valuses $N = 10^4, d = 0.1, \tau_f = 100\tau_m$). However, the error is limited due the fact that for larger P the ω term will dominate (a correct mean value), which yields the correct asymptote as illustrated by Table 3.2. □

3.6 Extensions

In this section we describe two simple extensions to the lower bound analysis method. The first extension applies to the introduction of a third term (next to φ and ω) that accounts for the transient skewing that occurs in a parallel system. The second extension deals with the approximation of the effects of simultaneous resource possession.

3.6.1 Skewing Effect

Inherent to our methodology, the models that are subject to our analysis method are transient, i.e., they eventually terminate (as does the corresponding time domain model). Consequently, there are cases in which performance is dominated by an initial (and final) transient phase, rather than the steady state phase. This transient phenomenon specifically occurs when multiple processes (initially) execute the *same* resource access sequence that gives rise to a large number of initial conflicts. As a result the processes are *skewed* during their subsequent steady state phase. Example models are the MRM (Example 3.4 and 3.7, especially for small N, also see Fig. 3.2), and, most notably, the pipeline model (Example 3.5 and 3.8). This one-only phenomenon is not accounted for by φ as it is a contention effect, nor is it accounted for by ω since this initial serialization phenomenon is independent of the possible steady state serialization. In the following we will account for the skewing effect when we derive an exact solution for a simple model called the "generalized MRM" that generalizes over models like the MRM and the pipeline. The result represents an optimization over the general bound

$$T^l = \max(\varphi, \omega)$$

Let the generalized MRM be given by

$$L = \mathbf{par}\ (p = 1, P)\ \mathbf{seq}\ (i = 1, N)\ \{\mathbf{delay}(\tau_l)\ ;\ \mathbf{seq}\ (m = 1, M)\ \mathbf{use}(u_m, \tau_m)\}$$

For $M = 1$ the model reduces to the MRM while for $N = 1$ and $\tau_l = 0$ the model reduces to a pipeline. After each process has finished its first iteration $i = 1$, the system enters a steady state in which each thread is delayed (skewed) with respect to its immediate resource access predecessor by an amount equal to the largest resource access delay (see Fig. 3.9). It holds

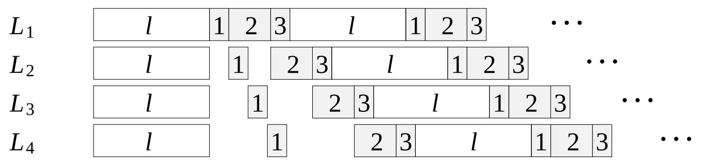


Figure 3.9: Trace of a 3-stage generalized MRM ($P = 4$, τ_m deterministic)

$$\varphi = N\left(\tau_l + \sum_{m=1}^{M} \tau_m\right), \quad \omega = PN \max_{m=1\ldots M} \tau_m$$

Let

$$\tau^* = \max_{m=1\ldots M} \tau_m$$

denote the largest resource access delay ($\tau^* = \tau_2$ in the example). From Fig. 3.9 it is easily seen that when contention only entails skewing it holds $T = \varphi + (P-1)\tau^*$ (critical path plus skewing). When contention does dominate the entire process it holds $T = \omega + \varphi/N - \tau^*$ (contention chain embedded within slightly longer trace). Consequently, for generalized MRMs the optimization of Eq. (3.14) is given by

$$T = \max(\varphi + P\tau^*, \omega + \varphi/N) - \tau^* \tag{3.16}$$

Note that for deterministic time delays the above result is exact, i.e., $T^l = T$ as is easily seen from the example in Fig. 3.9.

For the standard pipeline Eq. (3.16) reduces to

$$T = \varphi + \omega - \tau^*$$

For equal stages (i.e., balanced pipeline) it follows

$$T = (M + P - 1)\tau_m$$

which is the well-known linear startup-bandwidth model [72]. Note that for large P and M it holds

$$T \approx \varphi + \omega$$

instead of

$$T^l = \max(\varphi, \omega)$$

that explains the worst case deviation of a factor two between T^l and T as mentioned in Example 3.8.

3.6.2 Simultaneous Resource Possession

The analysis presented thus far has been restricted to models with single resource possession, i.e., to **use**(U, τ) operations where $|U| = 1$. In this section we briefly discuss the analysis of simultaneous resource possession. Note that this analysis is significantly harder than that for single resource usage as exemplified by the difference in complexity between, e.g., the analysis of queuing networks (single resource possession) and Petri nets (multiple resource possession) [117]. Consider the following model

$$\begin{array}{rcl} L & = & \textbf{par}\ (p = 1, P)\ L_p \\ L_p & = & \textbf{seq}\ (i = 1, N)\ \{\ldots\ ;\ t_1\ ;\ \ldots\ ;\ t_2\ ;\ \ldots\ ;\ t_3\ ;\ \ldots\} \\ t_1 & = & \textbf{use}(r_1, \tau_1) \\ t_2 & = & \textbf{use}(r_2, \tau_2) \\ t_3 & = & \textbf{use}(\{r_1, r_2\}, \tau_3) \end{array}$$

where r_1, r_2 are FCFS-type resources and the $\ldots$ regions represent a finite number of arbitrary statements. Instead of only two types of mutual exclusion, i.e., the r_1 accesses and the r_2 accesses, there are five serialization mechanisms to be considered:

- t_1 conflicts with t_1
- t_2 conflicts with t_2
- t_3 conflicts with t_3
- t_3 conflicts with t_1
- t_3 conflicts with t_2

Let the set $\{r_1, r_2\}$ be represented by the "simultaneous resource" r_{12} and let $\tau_{12} = \tau_3$. In terms of r_{12} it follows that the ω contribution of t_1, t_2, and t_3 to the lower bound is given by

$$\omega = NP \max(\tau_1, \tau_2, \tau_{12}, \tau_1 + \tau_{12}, \tau_2 + \tau_{12}) = NP(\tau_{12} + \max(\tau_1, \tau_2))$$

In general, the number of potential conflicts (i.e., intersections between sets of simultaneous resources $U_1, U_2, \ldots$) that need to be considered grows with the number and cardinality of the sets.

Like the approximate transformations, discussed earlier, that hold in terms of the lower bound analysis, we introduce the following transformation

$$\textbf{use}(\{r_1, \ldots, r_M\}, \tau) \rightarrow \textbf{par}\ (i = 1, M)\ \textbf{use}(r_i, \tau) \tag{3.17}$$

Thus, the potential conflicts between the various resource sets are preserved. Consequently, in terms of the lower bound, the above transformation yields the same solution as can easily be seen in the above example (if follows $\omega = NP \max(\tau_1 + \tau_3, \tau_2 + \tau_3)$). In terms of T, however, the transformation introduces an error since the original synchronization requirement is not present in the **par** statement. For example, consider the following process (taken from the case study in Section 5.3)

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \textbf{use}(\{f, x\}, \tau_x)\ ;\ \textbf{use}(f, \tau_y) \\ L_2 &= \textbf{use}(x, \tau_x) \end{aligned}$$

where f, x are FCFS-type resources (see Fig. 3.10). When L_1 is scheduled before L_2 it follows $T = T^l = \tau_x + \max(\tau_x, \tau_y)$. When L_2 is scheduled before L_1 it follows $T = T^u = \tau_x + \tau_x + \tau_y$. However, transformation (3.17) yields

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \{\textbf{use}(f, \tau_x) \parallel \textbf{use}(x, \tau_x)\}\ ;\ \textbf{use}(f, \tau_y) \\ L_2 &= \textbf{use}(x, \tau_x) \end{aligned}$$

which results in $T^u = T^l = \tau_x + \max(\tau_x, \tau_y)$. In this case, there are no synchronization restrictions with regard to the f resource, hence, the decrease in T^u. Note, however, that T^l is equal for both models.

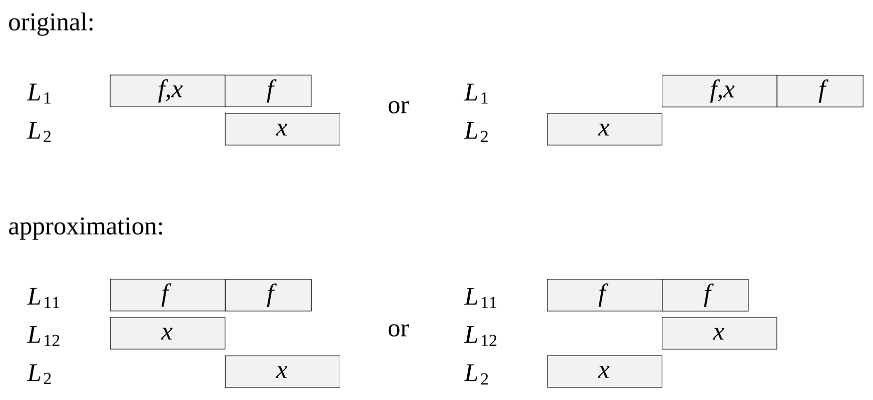


Figure 3.10: The effect of applying Eq. (3.17)

Example 3.12 In this example we demonstrate the application of the above approximation. Recall the dining philosophers problem in Example 3.3. Application of transformation (3.17) yields

$$\begin{aligned} L &= \textbf{par}\ (p=1,P)\ \mathit{philosopher}(p) \\ \mathit{philosopher}(p) &= \textbf{seq}\ (i=1,N)\ \{\mathit{think}(p)\ ;\ \mathit{eat}(p)\} \\ \mathit{think}(p) &= \textbf{delay}(\tau_t) \\ \mathit{eat}(p) &= \textbf{use}(c_p,\tau_e)\ \|\ \textbf{use}(c_{(p+1)\mathrm{mod}P},\tau_e) \end{aligned}$$

in which the problem has been generalized to P philosophers and where the original infinite loop has been replaced by a finite loop in order to apply our transient analysis method. Lower bound analysis yields

$$\begin{aligned} \varphi &= N(\tau_t+\tau_e) \\ \omega &= NP(\tau_e+\tau_e) = 2NP\tau_e \end{aligned}$$

Assuming τ_e is sufficiently large it follows $T^l = 2NP\tau_e$ which (correctly) implies that at most $P/2$ philosophers can be eating simultaneously. □

The above transformation can also be used for the lower bound analysis of models with *nested* resource usage. For example, consider the process

$$\textbf{using}\ (s_1)\ \{\textbf{delay}(\tau_s)\ ;\ \textbf{using}\ (s_2)\ \{\textbf{delay}(\tau_s)\ ;\ \textbf{delay}(\tau_x)\}\}$$

that models circuit-switched communication involving 2 intermediate switches s_1, s_2 (τ_s is setup time, τ_x is the data transfer time, see Section 4.3). First, the usage nest is approximated in terms of multiple resource possession, according to the transformation

$$\textbf{using}\ (r)\ L_1\ ;\ \textbf{use}(U)\ ;\ L_2\ \rightarrow \textbf{using}\ (r)\ L_1\ ;\ \textbf{use}(\{r\}\cup U,\tau)\ ;\ \textbf{using}\ (r)\ L_2 \quad (3.18)$$

where L_1, L_2 are arbitrary submodels and U is a resource multiset as defined earlier. This yields

$$\textbf{use}(s_1,\tau_s)\ ;\ \textbf{use}(\{s_1,s_2\},\tau_s)\ ;\ \textbf{use}(\{s_1,s_2\},\tau_x)$$

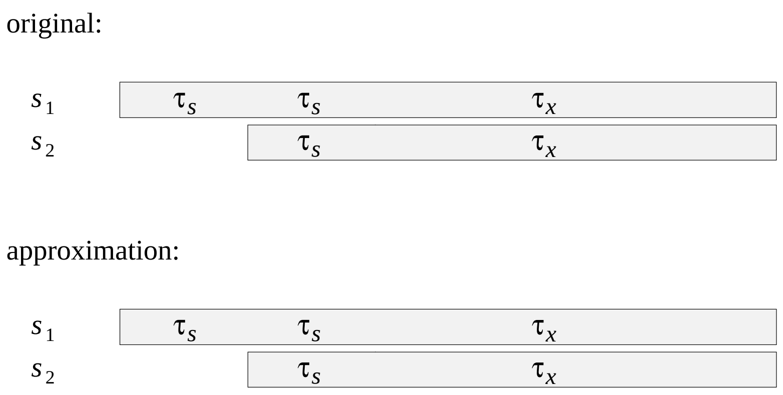


Figure 3.11: The effect of applying Eq. (3.18) and Eq. (3.17)

which, in turn, is approximated by Eq. (3.17) to

$$\textbf{use}(s_1, \tau_s) \ ; \ \{\textbf{use}(s_1, \tau_s) \parallel \textbf{use}(s_2, \tau_s)\} \ ; \ \{\textbf{use}(s_1, \tau_x) \parallel \textbf{use}(s_2, \tau_x)\}$$

The approximation process is depicted in Fig. 3.11. Again, note that the lower bound is preserved.

3.7 Summary

In this chapter we have introduced Pamela as well as a basic calculus that yields symbolic lower bound predictions for Pamela models that feature a structured use of resources. A key concept is the use of a procedure-oriented performance simulation formalism with structured synchronization operators that enables a material-oriented modeling approach coined contention modeling. As a result, many systems can be expressed in terms of an SP model that, unlike in a machine-oriented modeling approach, enables the application of an automatic compilation scheme yielding analytical performance models. From the point of view of typical mutual exclusion analysis techniques, such as conventional queuing theory or complexity theory, the critical path analysis prevents the usual errors when parallel system performance is dominated by task precedence relations. On the other hand, from the viewpoint of critical path analysis techniques, the (approximate) analysis of mutual exclusion prevents the large errors that occur when system performance is dominated by contention parameters. The integration of critical path analysis and contention analysis provides the basic prediction robustness needed in view of the large parameter space that is inherently covered by the analytic model.

As the language has been defined to provide sufficient modeling power in order to avoid *a priori* loss of accuracy at the modeling stage, the subsequent analysis method essentially breaks down in a number of categories as illustrated in Fig. 3.12. In terms of this taxonomy the specific contribution of the Pamela methodology pertains to the subset of "**use**" models. The underlying thesis is that a large part of the parallel computer systems can be adequately expressed in terms of structured mutual exclusion. Next to the examples discussed in this chapter, including inherently "message-oriented" problems (Example 3.6), the next chapter will substantiate this claim. Note that the appropriate

model category	*model structure*	*analysis method*
contention-free	SP ("**par/seq**")	symbolic expression (φ)
	non-SP ("**wait/signal**")	symbolic SE (φ)
contention	SP ("**use**")	symbolic expression (φ, ω)
	non-SP ("**use**")	symbolic SE (φ, ω)
	SP/non-SP ("**P/V**")	simulation

Figure 3.12: Performance modeling classification

analysis technique can be easily deduced given any Pamela model. The situation for Petri nets or process algebras (next comparable in terms of modeling power) is not as attractive because of the lack of structured mutual exclusion operators. Of course, restricted Petri nets include syntactic classes such as state machines, and marked graphs that are deducible at the transition or place level [117]. However, these restrictions yield a modeling power that is much less compared to the "**use**" subset of Pamela models.

Apart from the related work in terms of other representation formalisms[6] (as surveyed in Chapter 2), our language approach and associated symbolic analysis has clearly been influenced by much existing work in the language area. Language approaches to performance modeling in conjunction with an associated analysis method include the work of Lester (PEL [95]), and Qin (TCAS [125] that compiles symbolic model output). While symbolic model compilation originated in the sequential domain [44, 66, 157], in the parallel processing domain, symbolic compilation approaches have been recently described by Atapattu and Gannon [10], Clement and Quinn [29], Mendes, Yang and Reed [102], Sahner and Trivedi [134] (for stochastic graphs), and Wang [156]. As discussed in Chapter 2, however, none of the above approaches integrate contention analysis within the symbolic compilation scheme.

The lower bound approach to mutual exclusion analysis in static prediction techniques for parallel systems is not entirely new, be it that the underlying resource model is typically restricted to work conserving processor pools, unlike our general approach (at this point we do not discuss bounding analysis techniques for, e.g., queuing networks as we specifically focus on lower bound techniques for parallel systems including condition synchronizations). As mentioned in Chapter 2, a lower bound comparable to Eq. 3.14 has been used by Allen *et al.* [7] to account for the limited number of (multi)processing resources in compile-time prediction of dynamically scheduled task graphs. Recently, a technique has been described by Jain and Rajaraman [76] to predict the lower bound for optimal multiprocessor schedules that is tighter than the bounds obtained thus far in the multiprocessing domain. Comparable to our recursive approach (Eq. 3.15) they improve the sharpness of the basic bound given by Eq. 3.14 by applying the analysis to separate task graph layers (cf. Example 3.10). However, both approaches apply to *dynamic* scheduling of M equal processors only (i.e., one resource cpu with $cpu = M$ in terms of Pamela), whereas our generalized approach applies without any constraints on resource types or demands. In particular, our approach does *not* assume a work conserving scheduling dis-

[6]The attractiveness of a structured approach towards mutual exclusion, terms of both modeling and analysis is an important reason for the success of queuing networks.

cipline. In fact, when considering all resources in a *real* system, be it physical (processors, switches, memories) or logical (i.e., software services, critical sections), *unforced idleness* is quite common.

The choice to account for contention in terms of a lower bound model is based on two reasons. In contrast to the computation of a tight upper bound (discussed in Chapter 6), a tight lower bound can be computed at the same cost as conventional compile-time techniques. Moreover, as illustrated by many of the examples, in the limit, the lower bound model correctly predicts the average execution time of many systems that are either contention-free or fully saturated. While the efficiency of the presented analysis is optimal (linear for SP models), an important question is the accuracy of T^l compared to T. An elaborate discussion of this issue is presented in Chapter 6.

Being an experimental formalism, the definition of PAMELA may still be subject to (minor) changes. Especially the development of a compiler (currently under way) is expected to yield valuable feedback on the language level as well as on the calculus. Currently all experiments have been conducted using a discrete-event simulation kernel (see Appendix E) that is to be used as the compiler run-time environment. Although experimentation using this "PAMELA-like" library interface has been invaluable in the development of PAMELA and its associated analysis, a real language interface is expected to accelerate this process.

Chapter 4

Modeling Technique

4.1 Introduction

Thus far, we have discussed the application of PAMELA using problems relating to concurrency in a general sense. In this chapter we discuss how parallel computer systems are modeled (and subsequently analyzed). On the one hand, the discussion shows how PAMELA can be applied. On the other hand, it shows how the entire domain of (von Neumann) parallel system architecture (multiprocessing, multicomputing, vector processing) can be characterized in terms of one simple formalism. As our main purpose is to discuss the technique that we use in parallel systems modeling, we will only address the *principles* involved with modeling parallel machines without going into much detail with respect to contemporary parallel computer architecture.

In the previous chapter we have shown the material-oriented way in which parallel computations are modeled. In this paradigm parallel programs are mapped to "active" parallel processes while parallel machines are mapped to a set of concurrently operating "re-active" subroutines. The formalization of this concept in terms of PAMELA is by specifying a subroutine model for each instruction. For example, consider the scalar floating point computation $y = x_1 x_2$. Let the machine *program* be given by the process

```
move(x1,r1); move(x2,r2); mult(r1,r2,r3); move(r3,y)
```

thus assuming a traditional register architecture. Assuming that the register number does not affect performance, for the purpose of modeling we define the following abstract instruction set $mult$, $load(a)$, and $store(a)$ where a is the memory address (thus we distinguish memory loads and stores). Then the PAMELA *program model* becomes

$$L = load(x_1) \ ; \ load(x_2) \ ; \ mult \ ; \ store(y)$$

where the instructions now represent *models* of the original instructions. The set of these models is referred to as the *machine model*. In the most simple performance model each instruction maps to a simple time delay according to

$$\begin{aligned} mult &= \mathbf{delay}(\tau_m) \\ load(a) &= \mathbf{delay}(\tau_l) \\ store(a) &= \mathbf{delay}(\tau_s) \end{aligned}$$

As discussed before, the total performance model L is given by the combination of all four equations. By substitution it follows

$$L = \mathbf{delay}(\tau_l)\ ;\ \mathbf{delay}(\tau_l)\ ;\ \mathbf{delay}(\tau_m)\ ;\ \mathbf{delay}(\tau_s)$$

Consequently

$$T = 2\tau_l + \tau_m + \tau_s$$

Thus a key step in performance modeling of parallel computer systems is its mathematical formalization in terms of a program model and a machine model, both generally expressed as a set of PAMELA equations. Note, that modeling practice does not necessarily involve modeling applications at the instruction level. As shown by the above reduction of the four **delay** statements effectively into one **delay**$(2\tau_l + \tau_m + \tau_s)$ model, in many cases the modeling level can be increased to a macro instruction level (e.g., subroutine level) where the parameters of the aggregate model are determined in terms of the constituent submodels (as shown) and/or calibrated by measurements.

In practice, the subject of parallel systems modeling mainly concerns the way how *machines* are modeled rather than the programs. Being a simulation language, modeling imperative programs and algorithms in PAMELA is relatively straightforward. In view of the material-oriented modeling approach, this is particularly true in case of a procedure-oriented ("shared-memory") source, that is typical for intermediate code generated by compilers for high-level, global name-space programming languages such as the data parallel languages (e.g., HPF [86], FORTRAN-D [68], VIENNA-FORTRAN [161], BOOSTER [113], and KALI [87]), the language restructuring systems (e.g., Faust [59], Paraphrase 2 [123], ParaScope [25], PTRAN [7]), as well as the explicit parallel language interfaces such as VM/EPEX [34], the Force [80], and other dialects [83]. Even in the case of programs expressed in terms of an explicit parallel message-passing interface (e.g., PVM [148], MPI [154], or comparable interfaces [100]), the mapping to a PAMELA model is simple as will be shown by some of the examples in this chapter as well as in the succeeding chapter.

The simple nature of the above correspondence implies that (PAMELA) performance models can be automatically extracted from a program source description as in most parallel program performance prediction approaches discussed in Section 2.6 (e.g., [13, 29, 41, 102]). Thus, the prediction technique can be integrated within a more general parallel application engineering environment based on only one source specification. A good example of the advantages of integrating performance prediction within the software engineering process is shown by the N-MAP environment [43].

Although, the transformation of a source program to a PAMELA image can be assumed trivial in many aspects, there remains the fundamental problem of undecidability due to (data-dependent) program parameters, such as branching conditions and loop bounds when the analytic prediction technique is chosen rather than simulation. As discussed earlier, out of three common solution techniques are assumed to be implemented, i.e., automatic compile-time reduction, or, if not compile-time deducible, symbolic parameterization (in which the problem parameters are carried over in the time domain), or the use of probabilistic abstractions based on either symbolic values, numeric defaults or numeric profile data. Especially in the latter case, the underlying premise is that the program's branching behavior largely depends on the given data set (problem size, input data, etc.). Thus the profile obtained is also valid for a different parameter setting (e.g., different timing parameters or different number of processors[1]. Evidence suggesting this is presented in, e.g., [41]). As mentioned in Chapter 1, the issues involved in parameter calibration is outside the scope of our research.

To conclude the discussion on program modeling, we present a simple program modeling example (more examples can be found in Chapter 5). Many approaches in performance modeling are based on the notion that the program describes the *processes* with their parallelism, only limited by inherent algorithmic properties such as sequential fractions (condition synchronization), whereas the machine represents the collection of *resources*, and, consequently, is the system part responsible for the main performance degradation due to contention. Hybrid queuing networks are an example of a representation formalism that is conceived according to this model. Although this is often a realistic assumption (e.g., when only a limited number of processors are considered), the notion of resource limitations (i.e., queuing) do play a role at *program* level as well, as is shown in the next example.

Example 4.1 In this example we describe a case where mutual exclusion at the program level already dominates overall performance. Consider an SPMD shared-memory program that computes the sum of a global N-element floating point vector $\underline{v}$. Instead of applying a recursive doubling scheme, in this example the vector is simply block-wise partitioned, i.e., N/P consecutive elements are assigned to the same processor (for simplicity we assume $P|N$). The SPMD program is given by the pseudo code

```
for i = 0 .. (N/P)-1
    local = local + v[p*(N/P)+i];
set(lock);
sum = sum + local;
reset(lock);
```

where $p = 0 \ldots P-1$ denotes the processor index, `local` denotes a local summation variable (e.g., register), `sum` is the global result, its exclusive access ensured by the `lock` semaphore.

Let the machine model interface be given by only two instructions models,

- *flop* that models the floating point addition instruction including all local register traffic
- *move*(a) that models the global data transfer to or from shared memory where a denotes the address

Given the above instruction interface, the corresponding PAMELA model is given by

[1]Note that this only applies to branches and loop bounds that are not directly involved in the parallelization itself. Clearly, there exist many (data parallel) loops that are inherently affected. However, these branching and loop bound dependencies are already accounted for in the model as they are explicitly present in the (compiled) program.

```
$L =$ **par** $(p = 0, P-1)$ {
        **seq** $(i = 0, (N/P)-1)$ {
            $move(v + p(N/P) + i)$;
            $flop$
        } ;
        **using** $(lock)$ {
            $move(sum)$;
            $flop$;
            $move(sum)$
        }
    }
```

where the **par** construct models the SPMD parallelism. For simplicity, multiprocessing overhead is ignored. The FCFS-type resource *lock* ($lock = 1$) implements the critical section (this is an example of a *software* resource, rather than a hardware resource). Note that the mapping between the program code and the Pamela program model is straightforward.

As we intend to illustrate the dominating influence of program-level contention, let, as a first-order approximation, the machine model be given by the contention-free model

$$\begin{aligned} flop &= \mathbf{delay}(\tau_f) \\ move(a) &= \mathbf{delay}(\tau_m) \end{aligned}$$

It follows

```
$L =$ **par** $(p = 0, P-1)$ {
        **seq** $(i = 0, (N/P)-1)$ {
            **delay**$(\tau_m)$;
            **delay**$(\tau_f)$
        } ;
        **using** $(lock)$ {
            **delay**$(\tau_m)$;
            **delay**$(\tau_f)$;
            **delay**$(\tau_m)$
        }
    }
```

Application of the basic calculus for **delay** and **using** operations yields the following reduction

```
$L =$ **par** $(p = 0, P-1)$ {
        **delay**$((N/P)(\tau_m + \tau_f))$;
        **use**$(lock, 2\tau_m + \tau_f)$
    }
```

Thus, the summation algorithm (or any arithmetic reduction) has a similar behavior as the MRM. Simulation as well as our symbolic technique (Eq. (3.16)) correctly predict the initial speedup for small P as well as the eventual slow down for large P when lock contention dominates. $\Box$

Indeed many applications have three behavioral phases with respect to their scalability, i.e., initial speedup ($\mathcal{O}(P^{-1})$), a maximum bound ($\mathcal{O}(1)$) due to some sequential fraction, and an eventual slow down ($\mathcal{O}(P)$) due to service demand that is inherently proportional to the system size (P). In this example it is a numerical service. Another typical example is the data transfer part that is proportional to P (e.g., some shared I/O service).

4.2 Machine Modeling

4.2.1 Principles

Thus far, we have seen a few small examples of how machines are modeled in the material-oriented paradigm. Usually, machine models have been considered in which the instructions are simply modeled (abstracted) by single **delay** statements, or, in some cases, by a simple **use** statement in order to account for memory contention (Example 3.11). In general, however, machine models (especially those associated with data transfer instructions) may become rather complicated subroutines, involving synchronization delays at various levels, which (like at program level) can easily dominate performance.

While in program modeling, the level of detail is limited by the size of the program description itself, machines can be modeled to any level of detail, in principle up to, say, a gate-level hardware description. With respect to the performance prediction of parallel applications however, such a high level of detail does not necessarily add to the prediction accuracy (although it certainly does add to the prediction costs). Consequently, an important issue is the maximum level of aggregation that is acceptable without sacrificing predictive power (due to loss of vital information concerning the internals of the aggregate component). Essentially, the criterion regarding what to model or not to model explicitly is the question whether the time behavior of a potential component (aggregate submodel) depends on its access history. A basic example is a cache where the access delay experienced by a calling process depends on previous accesses. The mechanism involved must be explicitly modeled (in terms of conditional control flow) as we will see later on. While this form of context dependency also arises in purely sequential systems, in parallel systems there exists the additional potential for *interference* between *different* processes while accessing machine components due to *synchronization*. The primary form of synchronization between services at machine level is mutual exclusion where a service call of one program thread experiences (additional) queuing delay as a result of some other, possibly non-related thread of control (condition synchronization in message-passing systems will be discussed separately)[2]. If there is no potential for interference outside the range

[2]Note that in this respect where to attribute the actual "history" as mentioned earlier depends on the paradigm used. In a machine-oriented approach, the history indeed associates with the *component* in terms of the current state of its mutual exclusion mechanism (e.g., request buffer state). In the material-oriented approach, however, we tend to associate the concept of history with the current state of the concurrently calling *process*.

of the current caller, a component (and all within it) can simply be modeled by a single **delay** that has a fixed duration, independent of the state of other program threads[3]. For example, local arithmetic operations within a CPU may, at the register level, involve a large amount of internal concurrency. However, as the range of the operation is local to the processor there is no potential of interference between the instruction and instructions executed at other processors. Hence, from the viewpoint of the invoking program thread the operation (and therefore the entire subsystem involved with its execution) can be accounted for by a simple **delay** (note that we assume a synchronous instruction architecture where each instruction blocks until completion).

While the same argument holds for local data transfers (i.e., register to register), *global* data transfers may involve contention *between* CPUs for switch links and (global) memories. Consequently, from the CPU point of view, machines need only be modeled as far as global data transfer operations are concerned. Hence, all machine instructions involving local resources used in the context of a single CPU process are simply modeled by **delay**s, whereas global `load` and `store` instructions refer to more complex machine models.

Thus, from a parallel *process* point of view machines can be modeled as a collection of processors providing CPU service to the process threads at program level, each CPU being connected to a *shared memory subsystem* comprising switches, and possibly caches, all of which provide memory (`load` and `store`) services based on a bottom layer of basic memory devices as illustrated in Fig. 4.1. As a result of the above criterion by which

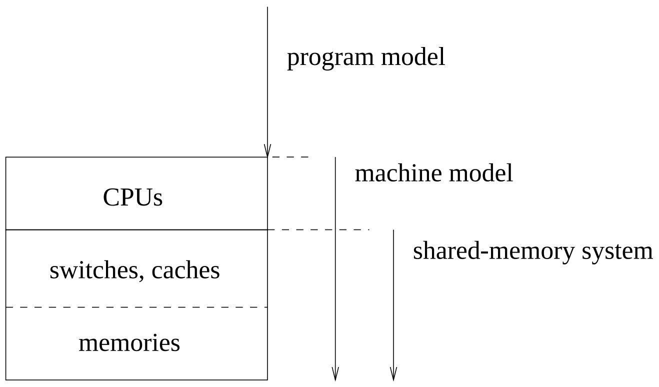


Figure 4.1: Machine modeling hierarchy

we only focus on the components that may be responsible for interference experienced at process level, the level of modeling detail is effectively limited to the *processor-memory-switch* (PMS) level. In the following we first consider the computation service layer (i.e., the CPU resources). The main part of this chapter, however, is devoted to modeling the memory subsystem as this part is by far characteristic for the machine architecture.

[3]Note that this also applies for caching in which case the **delay** expression represents a mean value being the result of a probabilistic reduction based on an average hit ratio.

4.2.2 Processor Modeling

In the SPMD examples presented so far, it has been assumed that each process executes on a unique processor. This implies that the effect of a limited number of processors P has directly been translated into the workload description (e.g., the size of the index space per process) through some static partitioning scheme. In the interest of generality, however, in machine modeling we must assume a more general scheme in terms of which the earlier model is just a specific instance. While the other resources in the PMS scheme will be treated in detail later on, at this stage we explicitly consider the CPU as just another resource capable of servicing multiple program-level processes through a (processor sharing) scheduler. Examples include a uniprocessor (e.g., workstation or mainframe) scheduler as well as a multiprocessor kernel or a node microkernel on a multicomputer. For instance consider the following parallel program model

$$\textbf{par}\ (i = 1, N)\ L_i$$

where L_i denotes some sequence of instructions local to the CPU (e.g., $flop$) with a total duration of τ_i. Consider a P processor parallel machine capable of running multiple processes per CPU according to a static process mapping. Let μ denote the mapping of the N process parallel program onto the P processor parallel machine, such that $\mu(i)$ denotes the processor resource onto which L_i is mapped. Then the resulting Pamela model is given by

$$\textbf{par}\ (i = 1, N)\ \textbf{use}(cpu_{\mu(i)}, \tau_i)$$

where the resource *cpu* ($cpu = 1$, PS-type) represents the CPU service (note that at the instruction level, *cpu* is FCFS-type). When multiple processes are mapped onto the same CPU, serialization will occur corresponding to the fact that each process receives less computation service (bandwidth). This is accounted for by the **use** construct (even to a high degree of accuracy as will be shown in Section 6.3).

According to this general modeling approach each local instruction such as *flop* is formally modeled

$$flop = \textbf{use}(cpu_{\mu(i)}, \tau_f)$$

thus signifying the resource responsible for its execution (note that formally, each operation is associated with some resource). Thus, by explicitly accounting for the CPU as a resource (i.e., modeling all local instructions as **use** models rather than **delay** models) the effect of process mappings is naturally expressed in Pamela through functions (like μ) that map logical resources to physical resources. In case of the SPMD examples discussed earlier, each process maps onto a unique CPU. As this implies that there is no potential interference between any of the $flop$ operations, the $flop$ model reduces to a simple **delay**.

The above approach corresponds to a static process placement. While this is appropriate for distributed-memory systems, in shared-memory systems, dynamic scheduling is often used because of the absence of large data transfer overhead when successor tasks are scheduled on different processors. Again, a processor sharing model can be used. Consider a P processor machine running N processes. When the scheduler itself is of no interest, a first-order approach is based on modeling the multiprocessor as a resource pool *cpu* (i.e., one resource instead of P separate resources), however, with $cpu = P$. Again local CPU instructions are modeled like the *flop* model

$$flop = \mathbf{use}(cpu, \tau_f)$$

Only when the amount of machine parallelism suffices ($P \geq N$ in the example) the $flop$ model reduces to a **delay**.

Also with respect to the processor layer note that the modeling approach is material-oriented in which the processor is modeled as a *passive* device, contended for by the program threads. Of course, this is in contrast to the typical implementation in which processor scheduling is often performed by an operating system (process) on interrupt (e.g., I/O or full time slice), rather than based on atomic process tasks voluntarily relinquishing control as in our instruction-level FCFS model. While the exact scheduling process at this level of detail may be of great interest for the evaluation of operating system aspects (e.g., fairness), especially in our approximate analysis the precise slicing granularity and scheduling order in which the processor resource is used is of no consequence as long as the total workload is the same. In Section 5.3 we will present measurements that show that a simple, coarse grain sharing model provides an accurate prediction.

In the following we discuss the memory modeling technique in which we characterize the machine architecture that is external with respect to the processor. Although from methodological point of view there is no difference between modeling shared-memory machines and distributed-memory machines in practice the architectures have different features as they reflect different design optimizations. Hence, we will discuss each architecture separately.

4.3 Shared-Memory Systems

4.3.1 Introduction

As mentioned earlier, apart from the computational services, an architectural description in Pamela is effectively a specification of the global memory resources and how they are made available (i.e., "seen" by the CPU) through intermediate components like caches and switching networks. (Note that the methodology equally applies to other forms of storage media like disks and intermediate software components like file servers including buffer caches, etc. For simplicity, however, we will limit the discussion to central memory systems.)

Due to the material-oriented paradigm, we characterize the memory architecture in terms of its access interface, i.e., the (global) `load` and `store` operations. Except where it is essential, in the following we will simply consider a *move* model that generalizes the data movement process we are basically interested in. Thus the general model of a shared-memory subsystem is

$$move(m, a)$$

where m denotes the memory system (or device) that is addressed while a denotes the memory address. Note, that the actual data transfers associated with loading and storing are quite different. For example, loading implies a return transfer next to sending the request. However, from our abstract point of view we assume that all latencies and queuing involved in both directions are accounted for in terms of the initial processor-to-memory path (typically one does not need to consider both the forward and return network separately).

Clearly, the terminal device of a shared-memory system is the basic (single ported) memory module shown in Fig. 4.2. The arrow in the figure denotes the interaction between

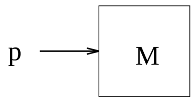


Figure 4.2: Basic memory module

the *active* process (denoted p) and the *reactive* memory component. In a basic memory module (single port, no internal caching) the externally measurable timing behavior will only depend on one internal state that is determined by previous requests, being the memory busy state. In contrast to the simple **delay** model often used at program-level analysis, in general, a memory resource m is modeled according to

$$move(m, a) = \mathbf{use}(m, \tau_a) \tag{4.1}$$

where τ_a denotes memory access time, that is defined to be the total time between starting the (load or store) transaction at the memory and the moment a new transaction can be started. In this general-purpose model, exclusive memory service is modeled by the resource m. Despite the fact that the basic memory unit is single-ported, this general model does account for the fact that multiple processes may gain access within a memory access time slot due to intermediate switching units (as we will discuss later on). If a **delay** model were used instead, this would allow multiple requests to be serviced simultaneously. Only when one process is to gain access, the above contention model may be reduced to a **delay** model as a result of the fact that the sequential access is synchronous (i.e., the memory will always be ready to service the next call). Note that conflict arbitration does not need to be considered since simultaneous accesses (from different clients) cannot occur (single port). Hence, a simple **P** operation suffices.

As discussed earlier, a memory system can be viewed as a layer of basic memory units with on top a layer that maps the basic memory service m' into more sophisticated, shared, service m (e.g., faster, more parallelism) to the processor layer. In terms of the material-oriented approach the system is expressed as

$$move(m, a) = \ldots\ ;\ move(m', a')\ ;\ \ldots$$

where the $move(m, a)$ model calls upon a lower-level $move$ model. In the following we will discuss the basic intermediate components that enhance the service provided by the basic memory unit, i.e., the *cache* ("temporal" enhancement through faster access time), and the *switch* that provides concurrent access ("spatial" enhancement through multi-port access). Especially switches will be discussed in more detail due to the prominent role played by contention as well as the fact that interconnection networks are largely responsible for the machine contribution to application performance. This fact is evidenced by the large body of work on the performance modeling of (shared-memory) interconnection networks (e.g., see [17]).

4.3.2 Cache

A cache model simply maps its move call $move(c, a)$ to its successor memory unit, which may be a basic memory unit as illustrated in Fig. 4.3, or may be some complex memory subsystem on itself. For the purpose of merely illustrating the general modeling technique

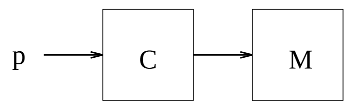


Figure 4.3: Cached memory system

we only discuss a simple load model for a request-buffered cache.

```
$move(c, a)$ = **using** $(c)$ {
                    **if** $(hit(a))$
                        **delay**$(\tau_c)$
                    **else** {
                        $move(m, a)$;
                        $update(a)$
                    }
               }
```

Again, the mutual exclusion construct ensures access serialization. In case of a cache hit (abstracted through the data-dependent *hit* function), a relatively small cache access time τ_c is charged. In case of a cache miss the lower-level memory unit m is invoked entailing a call latency that is much larger than τ_c (data is returned to caller and cache in parallel). The cache directory update is modeled by the *update* function. Both *hit* and *update* do not involve simulated delays. Note that this simple scalar model ignores the fact that usually a cache *line* is loaded from memory. Also the possible existence of cache coherence logic [145] has been ignored. The above example will be discussed further in Example 4.3.

4.3.3 Switch

While caches upgrade memory service from a *temporal* point of view, the main purpose of (shared) memory systems is to provide the memory service to *more* than one client (i.e., a service upgrade from a *spatial* point of view) through the use of switches. Again in this section we only touch upon the main principles as far as performance modeling is concerned.

The basic component that maps (single-port) memory service onto multiple clients is the n-to-1 *switch* illustrated by Fig. 4.4). As connecting multiple clients to a single succes-

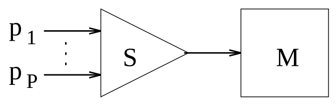


Figure 4.4: Multiported memory system

sor unit inherently involves contention, the switch's basic function is to implement mutual exclusion between multiple requesters and providing some sort of arbitration in case multiple requests arrive at exactly the same moment (i.e., at the same clock tick). Basically, two switching protocols can be distinguished, i.e., circuit-switching and packet-switching. Unlike the above, a more detailed *move* model is needed in order to describe both switching protocols (note, that our only purpose is to illustrate the modeling technique used, not to model all possible protocols in great detail).

In principle, a *move* model (or any service model) comprises the following protocol (in our material-oriented approach to be executed by the caller)

- service request (acquiring necessary resources)
- the actual service (data transfer)
- end notification (releasing resources)

Similar to typical service terminology, we will adopt the names *open* and *close* to model the first and last phase, while we will denote the actual data transfer by *xfer*. For simple models like a memory module it holds

$$\begin{aligned} move(m,a) &= open(m)\ ;\ xfer(m,a)\ ;\ close(m) \\ open(m) &= \mathbf{P}(m) \\ xfer(m,a) &= \mathbf{delay}(\tau_a) \\ close(m) &= \mathbf{V}(m) \end{aligned}$$

that reduces to the **use** model described earlier.

Let s denote a switch. Let m denote the downstream memory system to which s provides concurrent access. For circuit-switching, the basic switch model is given by the following model

$$\begin{aligned} move(s,a) &= open(s)\ ;\ xfer(s,a)\ ;\ close(s) \\ open(s) &= \mathbf{P}(s)\ ;\ \mathbf{delay}(\tau_s)\ ;\ open(m) \\ xfer(s,a) &= \mathbf{delay}(\tau_x)\ ;\ xfer(m,a) \\ close(s) &= close(m)\ ;\ \mathbf{V}(s) \end{aligned}$$

where τ_s denotes the startup time needed to setup the path through the switch, and τ_x denotes the time needed to propagate the datum through the switch. Thus, first all resources in the transfer path are acquired[4] before the actual transfer is executed.

[4]Apart from this so-called *hold* protocol, a second, so-called *drop* protocol exists in which a request is simply dropped (discarded) if during circuit establishment a contest is lost. The modeling of this protocol is discussed in Section 4.5.

Note, that because there is no intermediate buffering (like in packet-switching) τ_s and τ_x are compared to the downstream *open* and *xfer* parameters (especially for large data transfers). In the above model conflict arbitration is left undetermined for simplicity. In more detailed models more elaborate arbitration schemes are possible by inserting "user-defined" operators with explicit queue management using the basic **P**/**V** scheme as explained in Chapter 3.

For packet-switching, the basic switch model is given by the following model

$$\begin{aligned} move(s,a) &= open(s)\ ;\ xfer(s,a)\ ;\ close(s) \\ open(s) &= \mathbf{P}(s) \\ xfer(s,a) &= \mathbf{delay}(\tau_x) \\ close(s) &= \mathbf{V}(s)\ ;\ open(m)\ ;\ xfer(m,a)\ ;\ close(m) \end{aligned}$$

where any local setup time is accounted for by τ_x. In contrast to circuit-switching the switch provides transfer service as soon as its output link is available, after which the process moves to the next device. Note that this simple model assumes an infinite (packet buffer) queue for requests (packets) that are temporary blocked (resource queues are infinite). In terms of modeling the problem is comparable to the abstract, material-oriented solution to the producer-consumer problem (Example 3.2). In many practical situations, however, the fact that the actual "location" of each propagating process is left indeterminate does not degrade the overall model accuracy. In a detailed study the use of more elaborate synchronization models may, again, be necessary in order to account for the inter-switch handshaking involved. As discussed before, however, the two-way (message-passing) synchronization that arises with the use of bounded buffers is not amenable to our analysis technique.

Also note that in the case where a memory is connected through a packet-switched network, the mutual exclusion construct in the memory model of Eq. (4.1) is indeed necessary in order to prevent multiple processes from (almost simultaneously) accessing the module (assuming $\tau_x < \tau_a$). As mentioned earlier, the resource m symbolizes the queue at the memory module that buffers access requests passed on by the switch.

While the above models allow a reasonable description of circuit-switched and packet-switched communication, the use of the individual protocol phases introduces the explicit use of **P**/**V** operators which is unsuited for our calculus. Hence, we present versions of both models in terms of the structured templates we have introduced. Similar to the example of the memory module, for packet-switching the above model can be simply written as

$$move(s,a) = \mathbf{use}(s,\tau_x);\ move(m,a)$$

where each *move* service induces temporary resource usage followed by a service call down stream. While for packet-switching the result is equivalent to the original model, for circuit-switching the solution is slightly different because the switching workload is divided across two phases. The following expression

$$move(s,a) = \mathbf{using}\ (s)\ \{\mathbf{delay}(\tau_s)\ ;\ move(m,a)\ ;\ \mathbf{delay}(\tau_x)\}$$

adequately models the process although, compared to the earlier model, the transfer delays τ_x are accounted for after the ultimate *move* call, rather than prior to the call. However, in terms of performance (resource usage) both models are identical.

4.3.4 Networks

While the basic n-to-1 switch, as introduced thus far, captures two important aspects of switching networks, i.e., delay and contention, practical interconnection systems switch n sources to m destinations, rather than providing concurrent access to just one resource. In order to describe interconnection networks, we therefore generalize the above switching component by distinguishing m addressable output links. Consequently, the request address a determines the *routing* inside the switch (assuming decentralized routing control [17]). With the introduction of this decoding functionality, the n-to-m switch is simply referred to as an (n-to-m) *crossbar* of which a 2×2 version is shown in Fig. 4.5 in terms of its basic decoding and switching functionality (denoted D and S, respectively in the figure). Let $rout(a)$ denote the address decoding and routing function that determines

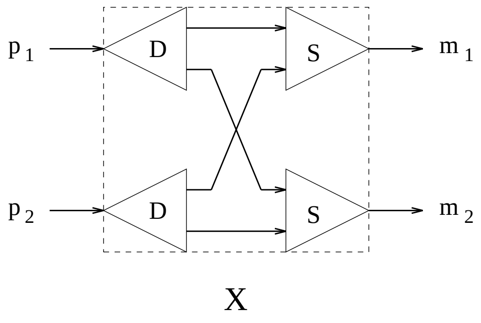


Figure 4.5: Crossbar switch model (2×2)

the output link index $i = 1, \ldots m$ and the address remainder a', used as address for the next stage[5]. Let d_i be the memory device at the output i of s. The crossbar model (again, ignoring arbitration, infinite buffers) is given by

$$move(s, a) = \{(i, a') = rout(a);\ \textbf{using}\ (s_i)\ move(d_i, a');\}$$

for a circuit-switched protocol (setup propagation delay ignored), and by

$$move(s, a) = \{(i, a') = rout(a);\ \textbf{use}(s_i, \tau_s);\ move(d_i, a');\}$$

for a packet-switched protocol. For $m = 1$ the decoding and routing function *rout* is immaterial ($i = 1$, $a' = a$). Thus, the above model generalizes over the former switch model.

As discussed earlier, an interconnection network provides a connection between multiple process clients and multiple memory servers, through a number of switching components. The most obvious IN is simply the above crossbar component that provides full interconnectability between any input and output without any intermediate blockage. (Of course, contention may still occur at each output i but this is inherent to any n-to-m IN.) However, the absence of intermediate blocking comes at the expense of a quadratic complexity in terms of internal logic. As a result of the trade-off between network cost

[5]Destination tag routing is typically given by the low-order address interleaving function $(i, a') = (a \bmod m, a/m)$.

and network performance, a large variety of network topologies exist of which we will only discuss the most common classes[6]. In essence, all these networks are based on the use of multiple, crossbar-like switches with a much smaller capacity than the total network capacity. Hence, we will model interconnection networks in terms of the above crossbar model, which we will adopt as a basic building block. Although this amounts to a graphical model representation, note that each crossbar represents the above Pamela model.

An example on the opposite extreme of the interconnection network spectrum is the (single) bus, that is modeled according to Fig. 4.6. While the second crossbar merely rep-

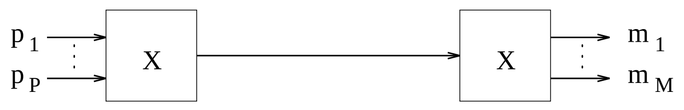


Figure 4.6: Bus system)

resents the decoding functionality of each of the M connected memory modules, the first P-to-1 crossbar (the output link resource s) accounts for the bus contention that occurs when multiple processors access memory. In order to support the full memory bandwidth of the M parallel memory modules, the bus is typically packet-switched ("pended" [158], see Example 4.4).

In terms of our generic crossbar model, a multiple-bus comprising B busses has the same representation. However, $s = B$ instead of $s = 1$. Hence, the effective bus bandwidth increases by a factor B which directly follows from Eq. (3.12).

Note that the configuration modeled according to Fig. 4.7. represents another solution

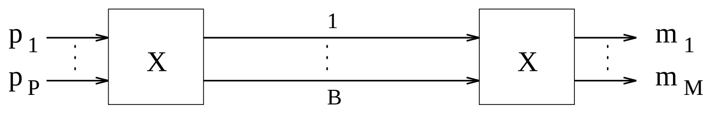


Figure 4.7: Multiple bus (static bus selection)

because of the static routing assumption of the crossbar model as defined earlier (B single servers vs. 1 multiple server).

Given the basic crossbar model, the expression of multistage interconnection networks is straightforward (see [42]) as illustrated by the example Omega network [92], shown in Fig. 4.8. Note that the resulting Pamela model accurately accounts for the possibility of internal contention at each intermediate switch level[7].

[6]An extensive survey of multiprocessor interconnection techniques appears in [17]. A survey of the performance aspects of some well-known multiprocessor architectures appears in [145].

[7]For instance, consider the two conflicts that may arise when p_0, p_1, and p_5 address m_0, m_3, and m_2, respectively.

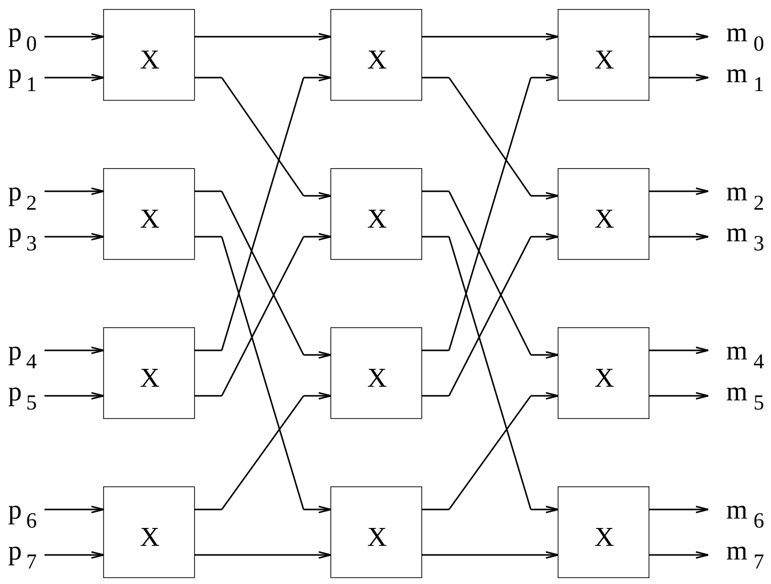


Figure 4.8: Omega network

4.3.5 Examples

In this section we describe a number of examples in which some of the machine modeling principles discussed so far are applied.

Example 4.2 In this example we demonstrate the modeling of memory contention as well as CPU contention in the case of the polynomial computation. Recall the parallelization of the polynomial computation for $P = 2$ in which we assume the machine model including the load/store overhead and memory contention (cf. Fig. 1.6 and 1.8). The PAMELA model is given by

$$\begin{aligned}
L &= L_1 \parallel L_2 \\
L_1 &= t_1 \;;\; \mathbf{signal}(c_{14}) \;;\; t_3 \;;\; \mathbf{signal}(c_{36}) \;;\; \mathbf{wait}(c_{25}) \;; \\
&\quad\; t_5 \;;\; \mathbf{wait}(c_{47}) \;;\; t_7 \;;\; \mathbf{signal}(c_{78}) \\
L_2 &= t_2 \;;\; \mathbf{signal}(c_{25}) \;;\; \mathbf{wait}(c_{14}) \;;\; t_4 \;;\; \mathbf{signal}(c_{47}) \;;\; \mathbf{wait}(c_{36}) \;;\; t_6 \;;\; t_8
\end{aligned}$$

where

$$\begin{aligned}
t_1 &= load \;;\; flop \;;\; store \\
t_2 &= load \;;\; load \;;\; flop \;;\; store \\
t_3 &= flop \;;\; store \\
t_4 &= load \;;\; load \;;\; flop \;;\; store \\
t_5 &= load \;;\; load \;;\; flop \\
t_6 &= load \;;\; load \;;\; flop \\
t_7 &= load \;;\; load \;;\; flop \;;\; store \\
t_8 &= load \;;\; flop \;;\; store
\end{aligned}$$

The abstract machine model is given by

$$\begin{aligned} flop &= \textbf{delay}(1) \\ load &= \textbf{use}(m, 0.5) \\ store &= \textbf{use}(m, 0.5) \end{aligned}$$

It follows

$$\varphi = 9.5 \quad , \; \omega = 8.5$$

which yields $T^l = 9.5$. From Fig. 1.8 it follows that $T = 11.5$.

In the following we demonstrate the use of CPU contention modeling in order to assess the performance of an implementation in which the actual order of tasks on each processor is scheduled dynamically. Under the same task mapping for $P = 2$, a dynamic task scheduling implementation is expressed by the original, fully parallel task model (cf. Example 3.1), i.e.,

$$\begin{aligned} L &= \textbf{par}\ (i = 1, 8)\ L_i \\ L_1 &= t_1\ ;\ \textbf{signal}(\{c_{13}, c_{14}\}) \\ L_2 &= t_2\ ;\ \textbf{signal}(c_{25}) \\ L_3 &= \textbf{wait}(c_{13})\ ;\ t_3\ ;\ \textbf{signal}(c_{36}) \\ L_4 &= \textbf{wait}(c_{14})\ ;\ t_4\ ;\ \textbf{signal}(c_{47}) \\ L_5 &= \textbf{wait}(c_{25})\ ;\ t_5\ ;\ \textbf{signal}(c_{57}) \\ L_6 &= \textbf{wait}(c_{36})\ ;\ t_6\ ;\ \textbf{signal}(c_{68}) \\ L_7 &= \textbf{wait}(\{c_{47}, c_{57}\})\ ;\ t_7\ ;\ \textbf{signal}(c_{78}) \\ L_8 &= \textbf{wait}(\{c_{68}, c_{78}\})\ ;\ t_8 \end{aligned}$$

however, where t_i is given as before in terms of *load*, *flop*, and *store*. However, now it holds

$$flop = \textbf{use}(cpu_{i \bmod 2})$$

where i denotes the task index $(1, \ldots, 8)$. In this model each task "contends" for its (statically assigned) processor, one of the possible task schedules being the one according to the earlier, static model. It follows

$$\varphi = 5 \quad , \; \omega = \max(7.5, 9, 8.5) = 9$$

where the max applies to cpu_1, cpu_2, and m, respectively. Consequently, $T^l = 9$, which is not far from the actual execution time corresponding to the static schedule ($T = 11.5$). □

Example 4.3 In this example we illustrate how the cache load model discussed earlier may be reduced to a performance model that is amenable to our analytic technique. Consider the simple cached memory such as depicted in Fig. 4.3. Based on the memory and cache models discussed earlier, the model equals

```
$move(c, a)$ = using $(c)$ {
                    if $(hit(a))$
                       delay$(\tau_c)$
                    else {
                       use$(m, \tau_a)$;
                       $update(a)$
                    }
               }
```

A number of model reductions are possible. First of all, as the memory module can not be accessed concurrently (cache provides mutual exclusion) the **use** model reduces to a **delay**. Furthermore, let the hit ratio be known to equal the fraction h. As the *hit* and *update* models do not involve time delays it follows

```
$move(c, a)$ = using $(c)$ {
                    if $(r < h)$
                       delay$(\tau_c)$
                    else
                       delay$(\tau_a)$
               }
```

which reduces to

$$move(c, a) = \mathbf{use}(c, h\tau_c + (1-h)\tau_a)$$

□

Example 4.4 In this example we demonstrate a number of model reductions by discussing the machine model of a bus-based multiprocessor with M (low-order) interleaved memory banks. As in earlier examples, we consider a simple instruction interface $(flop, move)$. For the *flop* model it holds

$$flop = \mathbf{use}(cpu_{\mu(p)}, \tau_f)$$

where μ denotes the process mapping (see Section 4.2). With respect to the *move* model, we first consider a conventional bus that is blocked during memory access (cf. circuit-switching). Hence, the *move* model is given by

$$move(a) = \mathbf{using}\ (b)\ \{\mathbf{delay}(\tau_b)\ ;\ \mathbf{use}(m_{a \bmod M}, \tau_a)\}$$

where b denotes the bus resource and m_i denotes the memory bank addressed according to the interleaved scheme (τ_a is the memory access time). For simplicity we assume all the bus delay (τ_b) to be accounted for by the single **delay** statement. Due to the fact that m_i is always accessed under mutual exclusion due to b, the model can be reduced to

$$\begin{aligned} move(a) &= \mathbf{using}\ (b)\ \{\mathbf{delay}(\tau_b)\ ;\ \mathbf{delay}(\tau_a)\} \\ &= \mathbf{using}\ (b)\ \mathbf{delay}(\tau_b + \tau_a) \\ &= \mathbf{use}(b, \tau_b + \tau_a) \\ &= \mathbf{use}(b, \tau_m) \end{aligned}$$

where $\tau_m = \tau_b + \tau_a$ denotes the effective memory access time (b is of type FCFS). Note that the actual address a is of no importance as the blocking bus prohibits any form of memory bank concurrency. The resulting model is exactly the same as the abstract machine model in Example 3.11.

If multiple processes were to run on a CPU (recall that the above **delay** is in fact a **use**$(cpu_{\mu(p)}, \tau_f)$), the above model assumes that the CPU is not occupied during memory system access. Consequently, during memory access another process at p will have the possibility to do computations. (Note that queuing approaches use the same assumption by including both a CPU server as well as a memory server.) When memory access can *not* be overlapped with computations, i.e., the access call is *synchronous*, the *move* model would be given by

$$move(a) = \textbf{using}\ (cpu_{\mu(p)})\ \textbf{using}(b)\ \{\textbf{delay}(\tau_b)\ ;\ \textbf{use}(m_{a \bmod M}, \tau_a)\}$$

which is an example of nested resource possession. Note that at higher software level (e.g., operating systems) the access call would be based on *simultaneous* resource possession in order to avoid wasting CPU cycles waiting for service (e.g., disk I/O). In most CPU thread kernels, however, instruction execution is non-preemptive. Hence, when a thread executes a memory instruction that blocks, the entire CPU becomes blocked which is essentially captured by the above nesting. Again, since $m_{a \bmod M}$ is always accessed under mutual exclusion (due to b, note that this does not hold for $cpu_{\mu(p)}$) the model reduces to

$$move(a) = \textbf{using}\ (cpu_{\mu(p)})\ \textbf{using}(b)\ \{\textbf{delay}(\tau_b + \tau_a)\} = \textbf{using}\ (cpu_{\mu(p)})\ \textbf{use}(b, \tau_m)$$

Note that this model can not be reduced further because of the need to account for bus contention as well as for sharing $cpu_{\mu(p)}$ with another process simultaneously doing computations.

Next we consider a *pended* bus that does not block during memory access (cf. packet-switching). The *move* model is given by

$$move(a) = \textbf{use}(b, \tau_b)\ ;\ \textbf{use}(m_{a \bmod M}, \tau_a)$$

Thus, bus access and memory access occur concurrently instead of sequentially which allows the effect of the M memory banks to become manifest. Serialization analysis quickly reveals that depending on τ_b, τ_m and M, for large P the bottleneck is either the bus of the memory bank system (see next example). Again, when the move call is synchronous, for multiple threads per CPU the model becomes

$$move(a) = \textbf{using}\ (cpu_{\mu(p)})\ \{\textbf{use}(b, \tau_b)\ ;\ \textbf{use}(m_{a \bmod M}, \tau_a)\}$$

□

Example 4.5 In this example we derive some familiar performance models of a multi-banked memory system for vectorized access. Consider a memory bank system comprising M memory banks organized according to a low-order interleaved addressing scheme. A memory vector access is based on a sequence of individual requests issued by an access port with a cycle time τ_c. Let each memory bank have an access time given by τ_m. The vector access pattern is given by the address sequence $f(1), \ldots, f(N)$. A simple PAMELA model of the vector move *vmove* is given by

```
$vmove$ = par $(p = 1, N)$ {
        use$(port, \tau_c)$;
        use$(m_{f(i) \mathrm{mod} M}, \tau_m)$
    }
```

where $m_{f(i)\mathrm{mod}M}$ models the memory bank. Again, note the material-oriented approach in which the port, that issues the addresses, is modeled as a passive resource that effectively serializes N parallel requests at the port request rate $1/\tau_c$ (cf. pipeline model). The requests are passed asynchronously, i.e., the port is only occupied during τ_c after which the request is processed (and perhaps queued) at the memory bank. Note that in a real system, the port may *block* until a memory bank is able to process another request (i.e., the memory banks have no request queue). However, as far as the performance of the *entire* vector operation is concerned, the above performance model is equivalent (again note the discussion in Section 3.3 on material-oriented modeling).

When M is sufficiently large (depending on the access pattern $f(i)$), memory contention will not occur and the above model reduces to the familiar "memory pipeline" characterized by startup time τ_m and bandwidth $1/\tau_c$ (see, e.g., [71, 94]). In practice, however, memory bandwidth is often determined by the limited number of memory banks M, that, under ideal circumstances boosts memory system bandwidth by a factor M compared to a single memory bank. However, in situations where the access pattern is such that the bank reference rate exceeds the memory bank service time, performance decreases sharply due to the occurrence of memory contention. The situation becomes even worse when multiple vector access streams occur in parallel. For regular vector accesses (i.e., f is affine), the effect of simultaneous access, start address, and stride on the effective memory bandwidth has been subject of much work [12, 21, 20, 24, 28, 40, 112]. In the following we apply our analysis method to derive the effect of the access stride on the effective memory bandwidth. Serialization analysis on the above *move* model yields

$$T = \max(\tau_c + \tau_m, \omega)$$

where

$$\omega = \max(N\tau_c, \max_{m=1\ldots M} \sum_{i=1}^{N} [f(i) \bmod M = m]\tau_m)$$

Let $f(i) = Si$ where S denotes the access stride. Since the following reduction holds (Definition 5.2, discussed in Section 5.2)

$$\sum_{i=1}^{N} [(Si) \bmod M = m] = \frac{N}{\gcd(M, S)}$$

it follows

$$T = \max(\tau_c + \tau_m, N\tau_c, \frac{N}{\gcd(M, S)}\tau_m)$$

When performance is limited by memory, the effective memory bandwidth B is given by

$$B = \frac{\gcd(M, S)}{\tau_m}$$

which is similar to the results mentioned in the work cited above. □

Example 4.6 As a final demonstration of our approach to machine modeling, we describe a model of the Cray X/MP memory system. The memory system has been extensively described in, e.g., [23, 28, 112]. However, these models usually reflect a hardware view of the underlying system rather than being based on a methodological performance modeling point of view. For instance, the memory bank sections are usually expressed in terms of the multiported memory banks as shown in Fig. 4.9. Although correct, the way the

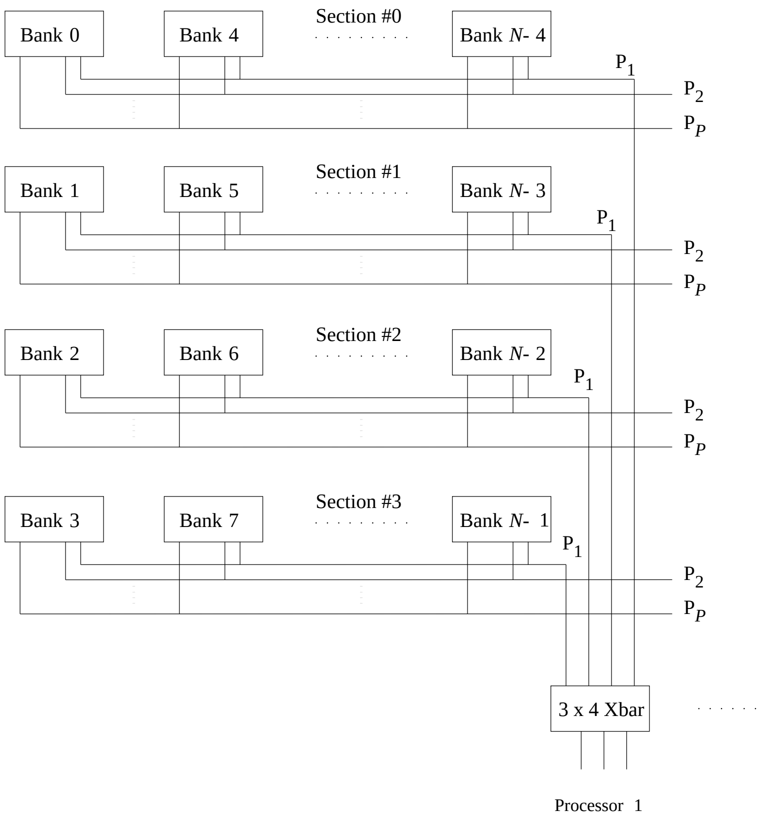


Figure 4.9: Cray X/MP memory system (taken from [23]).

architecture is modeled is not entirely conducive to revealing the actual locations where contention may occur (i.e., line conflicts, simultaneous bank conflicts, and bank busy conflicts). A PAMELA model of the memory system, on the other hand, turns out to be simple while providing more insight at the same time. Let B denote the number of banks per section, S denote the number of sections, T the number of vector ports per processor, and P the number of processors. Expressed in terms of our crossbar representation, the complete system is given by the multistage representation in Fig. 4.10. In the figure X and M denote crossbar and memory device, respectively. Both crossbar stages are interconnected through a simple permutation (a 2-D transposition). The multiport capability of the section memory banks is explicitly modeled by the second crossbar stage $X_1 \ldots X_S$, that is typically omitted in traditional diagrams. From the model the three potential contention points are now easily identified (i.e., each crossbar output link and each memory bank, respectively corresponding to the three conflict types mentioned earlier). □

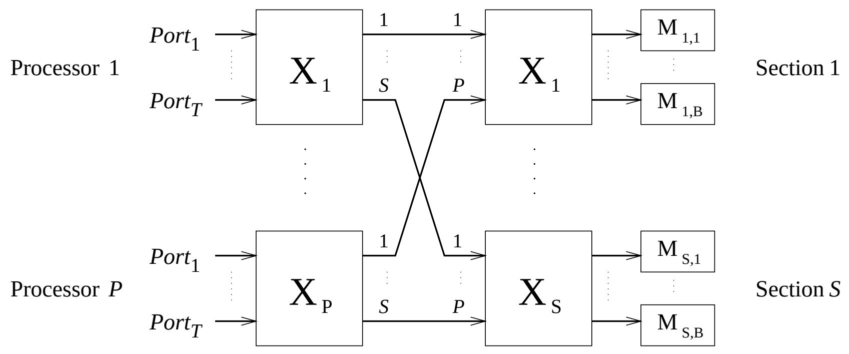


Figure 4.10: Pamela model of the Cray X/MP memory system.

4.4 Distributed-Memory Systems

4.4.1 Introduction

The global memory modeling approach taken in Pamela in which processes use processing and/or memory services, in effect, implies a unification with respect to parallel machine modeling. While most machine description taxonomies make a distinction between shared-memory and distributed-memory architectures (e.g., dynamic vs. static point-to-point topologies [42]), in the Pamela approach, a distributed-memory machine is basically just another global memory system but with a message-passing interface on top. The corresponding modeling hierarchy is shown in Fig. 4.11. To illustrate the principle, consider

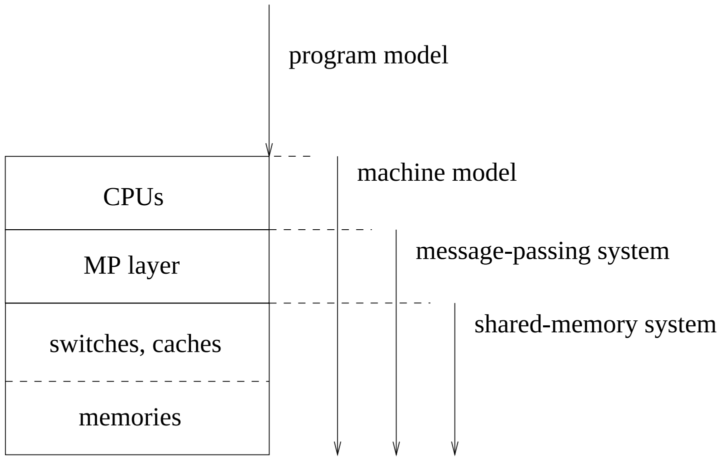


Figure 4.11: Modeling hierarchy for message-passing machines.

the shared-memory architecture depicted in Fig. 4.12 based on a 2-crossbar network. Despite its appearance however, the architecture corresponds to a 2-node *distributed-memory* machine with the PEs communicating through what is now usually referred to as a bi-directional link. (In a multiprocessor architecture a single, symmetric crossbar would, of course, suffice[8].) Note that the network invites the exploitation of memory locality due to its form (multiple hops for non-local loads and stores) and different timing parameters (the much longer links between PEs require much slower access speeds). The message-

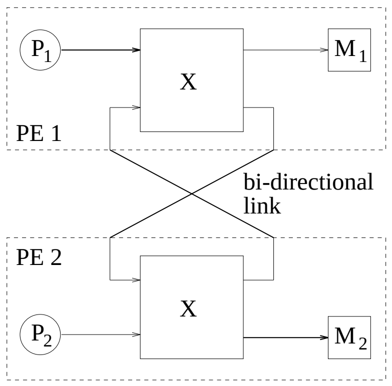


Figure 4.12: Shared memory hardware of distributed-memory machine.

passing layer on top of the above memory layer simply maps the `send/recv` calls onto the `load/store` (move) functions of the underlying global memory, incorporating some condition synchronization protocol. For instance, in the above architecture a simple (scalar) memory access model is given by

$$load(a) = store(a) = \{q = f(a);\ \mathbf{delay}([p \neq q]\tau_s);\ \mathbf{use}(m_q)\}$$

where m denotes the memory (and associated switch link), p denotes the processor index of the caller, and q denotes the processor index to which the target memory module is "local" ($f(a)$, determined by memory address a). The **delay** term accounts for the additional switching delay τ_s for a non-local transfer (in fact, a reduced model since at an outbound switch link no contention will occur). Note that in general τ_s is large which is why the model penalizes non-local memory access. A simple message-passing interface, organized around a scalar buffer at address `a`, would be modeled by the producer-consumer scheme (cf. Example 3.2, $room = 1$, $data = 0$)

$$\begin{aligned} send(a) &= \mathbf{P}(room);\ store(a)\ ;\ \mathbf{V}(data) \\ recv(a) &= \mathbf{P}(data)\ ;\ load(a)\ ;\ \mathbf{V}(room) \end{aligned}$$

where a is either in m_1 or m_2. Thus, all performance aspects of the message-passing layer (i.e., condition synchronization protocol) and underlying network layer (i.e., latencies, bandwidth, mutual exclusion on links and buffers, etc.) are accounted for (for simplicity, protocol stack overhead has been ignored in the example). Note that the physical implementation of the message-passing layer (handshaking traffic) is completely abstracted. Only the condition synchronization, i.e., its effective result, is modeled. The overhead of the synchronization protocol is assumed to be accounted for by the startup cost of the message-passing calls (discussed later on).

[8]Note that a multicomputer topology like the hypercube is, in fact, related to the Omega type multistage network known in the multiprocessing domain [114]. For busses, the similarity is obvious.

4.4.2 Basic Communication

In the above example, the actual interprocessor communication could be through a memory location at either sender or receiver. In typical message-passing systems, however, the target (or intermediate buffer) address resides at the receiver which implies that the actual interprocessor data transfer is initiated by the *store* operation, rather than the *load* operation. Furthermore, typical interfaces explicitly refer to a source, destination, as well as intermediate data location involved in the data transfer. An example in which the addressing scheme is associated with both the processors involved are the calls `send(r,x)` that stores local (user) address x to some remote address designated by the receiver r, and `recv(s,y)` that loads data from sender s from some intermediate buffer into (user) address y.

As in general a data transfer involves much more than just a scalar operation, we will introduce the block move model $bmove(a, b, l)$ to denote the actual global transfer of l (contiguous) data elements between address a and b. Consequently, a synchronous message-passing model is modeled as

$$\begin{aligned} send(r, x, l) &= \mathbf{P}(room) \ ; \ bmove(x, b, l) \ ; \ \mathbf{V}(data) \\ recv(s, y, l) &= \mathbf{V}(room) \ ; \ \mathbf{P}(data) \ ; \ bmove(b, y, l) \end{aligned}$$

where b denotes the local buffer address for data from s, with $room = 0$ and $data = 0$ (in fact, a "rotated" version of the producer-consumer model in Example 3.2 with regard to the receiver).

The asynchronous message-passing scheme can be conveniently expressed in terms of the same model with $room = C$ where C denote the initial capacity of the buffer (mailbox). Note that typical implementations of asynchronous sends will never block which is modeled by $C = \infty$. As in practice C is certainly not infinite this implies that overflowing messages may have to be discarded. When infinite buffers are assumed the above model reduces to

$$\begin{aligned} send(r, x, l) &= bmove(x, b, l) \ ; \ \mathbf{V}(data) \\ recv(s, y, l) &= \mathbf{P}(data) \ ; \ bmove(b, y, l) \end{aligned}$$

Another consequence is that when the pair of tasks that are to communicate is determined at application level (e.g., in terms of some task index i and j), the model can be written in terms of

$$\begin{aligned} send(r, x, l) &= bmove(x, b, l) \ ; \ \mathbf{signal}(c_{ij}) \\ recv(s, y, l) &= \mathbf{wait}(c_{ij}) \ ; \ bmove(b, y, l) \end{aligned}$$

as a result of the fact that the memory property of the $\mathbf{P}/\mathbf{V}$ operators is not used. Clearly, the implications for the analyzability of message-passing programs are considerable (see Example 4.7 and also Example 5.1).

With respect to the *move* model we use the same modeling principles as described for interconnection networks for shared-memory systems. For example, the interconnection network of the iPSC/2 hypercube [111] (P nodes) can be modeled by P $(\log_2 P + 1) \times (\log_2 P + 1)$ crossbars (like the earlier example, an extra link is needed to connect with

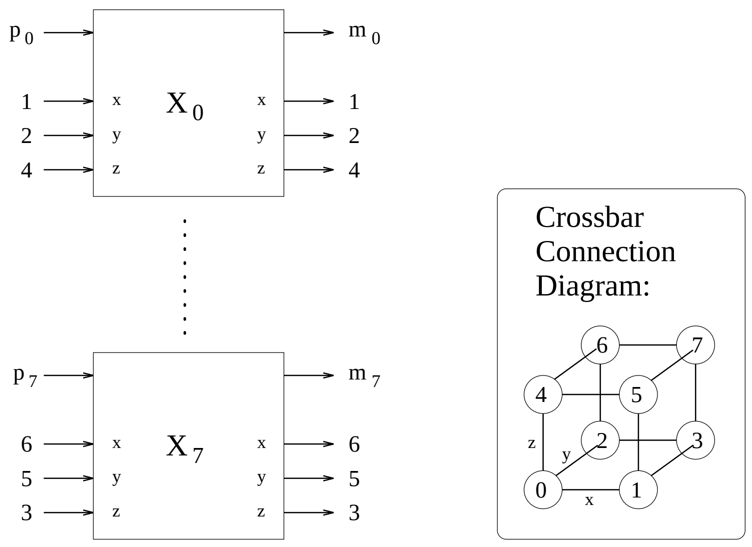


Figure 4.13: PAMELA model of Intel iPSC/2 distributed memory system.

local traffic, see Fig. 4.13). Compared to the hypercube (omega) network discussed earlier this network is clearly optimized for memory access locality (0 hops for local access, up to logarithmic number of hops for remote access). Let $n_k = s \dots r$ denote the index of the K nodes involved in the path between processors s and r where $n_1 = s$ and $n_K = r$. In the following we will ignore the forwarding delay through the pipeline of switches. Let c_i denote the switch link (channel) responsible for transferring data from n_i to n_{i+1}. For circuit-switched systems (like the iPSC/2) the *bmove* model can be roughly expressed by (ignoring circuit setup overhead)

$$bmove(x,b,l) = \textbf{using}\ (c_1)\ \{\textbf{using}\ (c_2)\ \{\dots \textbf{using}\ \{(c_{K-1})\ xfer(x,b,l)\}\dots\}\ \}$$

and the data transfer is modeled by

$$xfer(x,b,l) = \textbf{seq}\ (i=1,l)\ \textbf{delay}(\tau_c) = \textbf{delay}(l\tau_c)$$

where τ_c denotes the transfer time per unit data. Note that the initial link resource c_1 would always seem available as the outbound link is used only by local senders. However, in order to account for multiple threads sending concurrently (in non-blocking send mode), the c_s term must be included.

Also for packet-switched interconnection systems the situation is comparable to the shared-memory model. (In Example 4.9 a message-passing model will be described for a mesh topology.) The fact that in distributed-memory networks message *vectors* are transferred instead of unit data implies that a pipelining model must be used. However, due to the material-oriented approach, the basic propagation model stays the same. Consider a vector of l data elements. Let the packet size be W. For simplicity we assume padding such that the last packet has the same length W (although l may not be a multiple of W). Again the forwarding nodes are denoted by channels $c_1 \dots c_{K-1}$. Note that in this respect a channel resource may represent a node's CPU (older generation systems) as well as dedicated switching hardware (current generation systems). For one packet a coarse model for *bmove* is given by

$$bmove(x, b, W) = \textbf{seq}\ (k = 1, K-1)\ \textbf{using}\ (c_k)\ xfer(b_k, b_{k+1}, W)$$

where b_k denotes the address of the intermediate packet buffers. Unlike the shared-memory switching model, however, the multiple packet transfer is pipelined which is modeled as

$$bmove(x, b, l) = \textbf{par}\ (i = 1, \lceil l/W \rceil)\ \textbf{seq}\ (k = 1, K-1)\ \textbf{using}\ (c_k)\ xfer(b_k, b_{k+1}, W)$$

according to the material-oriented approach (cf. pipelining example). Note that the above model accounts for startup time, bandwidth, as well as the effect of link contention with simultaneous communications that use mutual channels (forwarding services). This is a significant improvement over conventional models (the linear **delay** model accounting for startup and bandwidth [9, 18, 70]) that only predict communication performance for *isolated* point-to-point communications. The above model will be used to describe the message-passing interface of a transputer mesh (Example 4.9).

4.4.3 Non-blocking Communication

The basic message-passing systems discussed thus far are blocking in the sense that the actual message transfer (the *bmove* call) is *synchronous* with respect to the sending process. Especially when dedicated transfer hardware is installed (e.g., c_i is implemented by switches or DMA devices), even an asynchronous send will entail many lost cycles for the calling process. In order to provide additional scheduling freedom current interfaces feature a non-blocking version of the (a)synchronous send and receive call. The typical use of such a call is to exploit the additional concurrency for overlapping communication and computation, or to execute concurrent communications under the assumption that the underlying system actually supports this form of parallelism.

The implementation of a non-blocking call typically involves a separate, dedicated process that takes care of the actual (blocking) call. Consider a non-blocking send call `nbsend`. The call communicates the actual send request (by process s) to a dedicated (kernel) process S according to the (blocking) scheme

$$nbsend(r, x, l) = send(S, a, 2)$$

in which both parameters $a = (r, x)$ are passed to S (hence the parameter '2' in the above model). The *send* and *recv* models are defined as earlier and S executes the service

$$\textbf{while}\ (true)\ \{recv(s, a, 2);\ \ send(r, x, l)\}$$

Thus, the intermediate process insulates the program-level process from the blocking call. Note that the intermediate communication may involve an asynchronous request buffering scheme.

Like the earlier model, the above model is essentially a message-passing model. Although the model is a legal Pamela model, the message-passing paradigm used to model the call (featuring the kernel process) makes it hard to apply serialization analysis (as discussed in Chapter 3). In order to allow the call to be modeled in terms of a *material-oriented* PAMELA model one must consider the call in combination with its formal counterpart, i.e., the `nbtest`, that tests if the transfer has actually finished. (This is especially appropriate for `nbrecv`, however, note that a non-blocking scheme that never blocks must always include a test function or must eventually block when out of resources.) In fact, this problem is an example of the general idea that the fork-join structuredness of PAMELA's '$\|$' operator somehow forces the system under study to have structure as well (thus enabling compile-time analyzability). Assuming that `nbsend` is always used in conjunction with the `nbtest`. The semantics of the blocking `nbtest` are such that the following equality holds

$$send(r, x, l) = nbsend(r, x, l)\ ;\ nbtest(r)$$

In modeling *nbsend* we use the following (structured) template

$$nbsend(r, x, l)\ ;\ L\ ;\ nbtest(r)$$

where L is program code that is executed between `nbsend` and `nbtest`. Note that the above template is a fork-join template. A procedure-oriented model of this template is given by

$$nbsend(r, x, l)\ ;\ L\ ;\ nbtest(r) = L \parallel send(r, x, l)$$

This model is amenable to serialization analysis. Even when the `nbtest` is not present (in the program) at modeling-time always an appropriate location can be found at which point the processes are joined (e.g., the end of the parallel section of which L is part of). Depending on the nature of the forwarding hardware, the communication proceeds either fully or partially in parallel (see Example 4.8).

4.4.4 Examples

In this section we describe a number of examples that show some of the principles underlying the modeling of distributed-memory systems.

Example 4.7 In this example we demonstrate the (performance) equality between simple message-passing implementations and shared-memory implementations. Consider an implementation of the polynomial computation statically scheduled for $P = 2$ on a distributed-memory machine with asynchronous *send*/*recv* instructions. The PAMELA model would be given by (ignoring the actual data)

$$\begin{aligned}
L &= L_1 \parallel L_2 \\
L_1 &= t_1\ ;\ send(cpu_2)\ ;\ t_3\ ;\ send(cpu_2)\ ;\ recv(cpu_2)\ ; \\
&\quad\ t_5\ ;\ recv(cpu_2)\ ;\ t_7\ ;\ send(cpu_2) \\
L_2 &= t_2\ ;\ send(cpu_1)\ ;\ recv(cpu_1)\ ;\ t_4\ ;\ send(cpu_1)\ ;\ recv(cpu_1)\ ;\ t_6\ ;\ t_8
\end{aligned}$$

where each task t_i is given by

$$\begin{aligned} t_1 &= load \ ; \ flop \\ t_2 &= load \ ; \ load \ ; \ flop \\ t_3 &= flop \\ t_4 &= load \ ; \ flop \\ t_5 &= flop \\ t_6 &= load \ ; \ flop \\ t_7 &= flop \\ t_8 &= flop \end{aligned}$$

Note that the local *load* instructions account for the initial loading of variables x, $a_0, \ldots a_3$ in order to achieve a comparable situation with respect to the original shared-memory implementation. When we consider a simple (memory-less) implementation of the message-passing interface according to

$$\begin{aligned} send(r) &= store \ ; \ \mathbf{signal}(c_{ij}) \\ recv(s) &= \mathbf{wait}(c_{ij}) \ ; \ load \end{aligned}$$

the model automatically reduces to the original shared-memory model described in Example 4.2. Thus, in terms of the PAMELA models, the actual difference between applications programmed in a shared-memory paradigm or in a (simple) distributed-memory paradigm is merely a matter of choice between instruction interface abstraction levels. □

Example 4.8 In this example we analyze the effective performance yield of non-blocking communication. In earlier systems, the effective overlap of computation and communication can be disappointing. For example, consider a non-blocking nearest-neighbor communication of l data units in which *cpu* acts as sender. For simplicity assume that the receiver has already posted its `recv` call such that the effect of condition synchronization can be ignored. Let the blocking send be modeled by

$$send(l) = \mathbf{use}(cpu, \tau_s) \ ; \ \mathbf{par} \ (i = 1, \lceil l/W \rceil) \ \mathbf{using} \ (cpu) \ xfer(W)$$

where τ_s represents software (call) overhead. For simplicity in this model we only consider the l parameter. Due to the absence of (multi-hop) pipelining (nearest-neighbor communication), the model simply reduces to

$$\begin{aligned} send(l) &= \mathbf{use}(cpu, \tau_s) \ ; \ \mathbf{seq} \ (i = 1, \lceil l/W \rceil) \ \mathbf{using} \ (cpu) \ xfer(W) \\ &= \mathbf{use}(cpu, \tau_s) \ ; \ \mathbf{use}(cpu, \lceil l/W \rceil W \tau_c) \end{aligned}$$

as a result of serialization, where τ_c is the transfer time per data element. Thus all the communication workload is attributed to the CPU.

Let L represent local computation modeled by

$$L = \mathbf{use}(cpu, \tau_f)$$

In this case, attempting to overlap the computation with communication yields no speedup at all as

$$\begin{aligned} nbsend(l)\ ;\ L\ ;\ nbtest &= L \parallel send(l) \\ &= \mathbf{use}(cpu, \tau_f) \parallel send \\ &= \mathbf{use}(cpu, \tau_f) \parallel \mathbf{use}(cpu, \tau_s)\ ;\ \mathbf{use}(cpu, \lceil l/W \rceil W \tau_c) \\ &= \mathbf{use}(cpu, \tau_f + \tau_s + \lceil l/W \rceil W \tau_c) \end{aligned}$$

Note that *cpu* is PS type.

On the other hand, consider a transputer architecture where the actual data transfer is performed through DMA. The *send* model is given by

$$send(l) = \mathbf{use}(cpu, \tau_s)\ ;\ \mathbf{use}(dma, \lceil l/W \rceil W \tau_c)$$

where *dma* represents the link DMA device ($dma = 1$, PS type at multipacket level). It follows

$$\begin{aligned} nbsend(l)\ ;\ L\ ;\ nbtest &= \mathbf{use}(cpu, \tau_f) \parallel \mathbf{use}(cpu, \tau_s)\ ;\ \mathbf{use}(dma, \lceil l/W \rceil W \tau_c) \\ &= \mathbf{use}(cpu, \tau_f + \tau_s) \parallel \mathbf{use}(dma, \lceil l/W \rceil W \tau_c) \end{aligned}$$

Serialization analysis yields

$$W = \max(\tau_f + \tau_s, \lceil l/W \rceil \tau_c)$$

Provided software startup overhead is small, computation can indeed be overlapped with communication. □

Example 4.9 In this example we develop a communication bandwidth model for a Parsytec T800 transputer mesh[9]. In contrast to the earlier examples this study involves modeling and measurement of actual machine hardware in which most of the principles described earlier are demonstrated. The model provides a good first-order performance approximation of large data transfers in the presence of *simultaneous* communications. The bandwidth model will be used in a case study in which the execution times of a number of applications running on the T800 mesh are compared with our predictions. Although the model is based on a number of (simplifying) assumptions the predictions prove to be quite accurate as is shown later on. The message-passing interface considered is based on the "virtual link" service, that provides a dedicated logical channel between a sender and receiver task. Since the virtual link topology needed to connect senders and receivers is setup in the prologue of the actual application discussed above, link setup times need not to be considered. The communication mode selected is "asynchronous" in terms of the machine interface, which, in reality however, implies a non-blocking mode in our modeling terminology (sender unblocks before transfer has completed). The communication mode selected does not involve buffer copying at the sender, nor at the receiver (direct copy from user address x to y as described previously). In terms of the message-passing interface the communication functions correspond to `ARecv`, and `ASend` calls,

[9]Kindly made available by the Interdisciplinary Center for Computer-based Complex systems research Amsterdam (IC^{3}A).

respectively [115]. In contrast to the address parameters used earlier, in the following we will consider $bmove(s, r, l)$ where s and r denote sender and receiver *node*, respectively.

The virtual link service of the transputer system is based on a multiplexing scheme in which each 120 bytes of the message is packetized. Each packet is statically routed through the mesh in a pipelined fashion. Unlike the T9000, the virtual link service is still emulated by the T800 node kernel. Consequently, each transfer not only induces workload at all the hardware links in the circuit but also at software level on each forwarding node. Traditional communication models only characterize communication performance in terms of a point-to-point model that typically accounts for latency and bandwidth only [9, 18, 70]. In terms of PAMELA this implies a linear **delay** model according to

$$bmove(s, r, l) = \mathbf{delay}(\tau_s + l\tau_c)$$

where τ_s denotes the startup time (typically including a dependency on the number of hops between s and r) and τ_c denotes the inverse bandwidth. For a 4×4 mesh partition Fig. 4.14 shows the execution time as measured on the mesh. Note, that in the above linear model the effects of packetization are ignored (the small increments per 120 bytes shown in the figure).

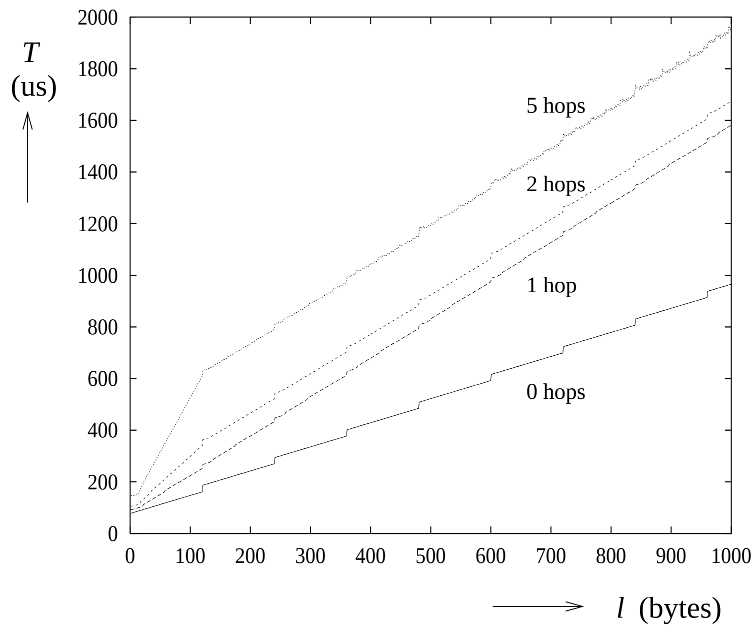


Figure 4.14: Point-to-point communication time of the T800 transputer mesh

In contrast to this **delay** construct we express the additional knowledge concerning the potential contention delay in terms of **use** constructs referring to the limited number of services available. With each physical link between neighboring transputers we will associate a *service complex* comprising a subsystem of physical (e.g., DMAs at both link ends) and/or semi-logical (software servers at both ends) resources. Without any loss of generality, we project the service complex at the receiving node of each link, as shown in Fig. 4.15. In the following we consider the communication system at the packet level which is the smallest level of granularity with respect to resource sharing. Although the service complex at each link comprises several software/hardware components, it can be modeled as to provide two services at the packet level that are subsequently denoted e and f (see Fig. 4.15).

The first service e represents the r*e*ception service at the packet destination involving the exclusive transfer of one packet across the link, including the software overhead at both ends (e.g., moving, handshaking). The second service f represents the *f*orwarding service (including intermediate storage and protocol overhead) required for a packet destined for a different node. Consequently, compared to e, f includes additional routing/forwarding workload. In general, a packet transfer from node $s = n_1$ to $r = n_K$ will require forwarding at $n_2 \ldots n_{K-1}$ (according to static x-y routing[10]) and one reception service at n_K. Both services are based on an underlying service, represented by the resource x that represents the basic link service that has to be shared. Consequently, e and f are *logical* resources (kernel servers) sharing the underlying link service. Apart from sharing link transfer service x supplied by the sending and receiving DMA resources for each channel, e and f are mapped onto the CPU resources as well since receiving a packet, and, most notably, forwarding a packet involves a number of CPU cycles. However, the effective CPU bandwidth as available for regular computation is hardly affected by the occurrence of data transfers (up to approximately 20 % CPU bandwidth degradation when all four reception or forwarding services are simultaneously active). Furthermore, communication bandwidth is also not affected by the number of links that are simultaneously active. Consequently, in our approximate model we simply ignore the impact of e and f on the CPU performance (and therefore on neighboring e and f services as well[11]). Thus, the model is expressed in terms of the logical services e and f (as if supported by independent physical servers) and a physical link (DMA) service x. Typical for the PAMELA method-

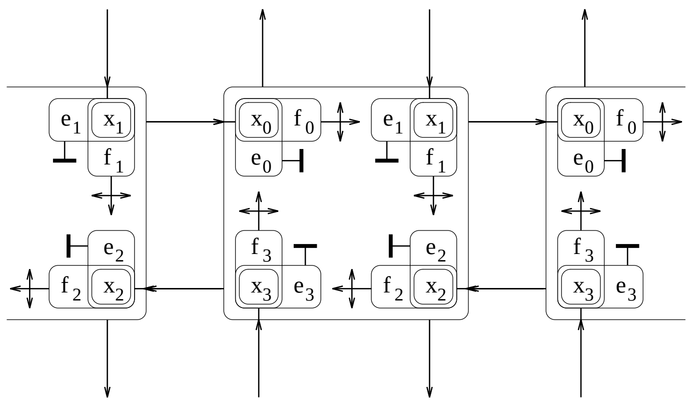


Figure 4.15: Message-passing service model of the T800 transputer mesh

[10]First, $|(s/4) - (r/4)|$ nodes in the sender column are traversed after which $|(s \bmod 4) - (r \bmod 4)|$ nodes in the receiver row are traversed (in terms of row-major node numbers).

[11]A possible explanation for this relative absence of CPU interference is that the (high-priority) CPU kernel involved with supporting e and f is based on a (busy waiting) scheme such that a number of cycles are spent anyway, regardless of the presence of (useful) service activity (e and f). The effective CPU bandwidth as experienced by user threads is therefore a relatively constant fraction of the total CPU power available. As no information on the kernel was available the exact reason could not be investigated.

$\underline{s}$	$\underline{r}$	T^m	T	T^t
(0)	(1)	0.9	0.9	0.9
(0)	(2)	1.5	1.5	1.5
(0,0)	(1,1)	1.8	1.8	0.9
(0,0)	(2,2)	3.0	3.0	1.5
(0,0)	(1,2)	1.8	1.8	1.5
(0,0,0)	(1,1,2)	2.7	2.7	1.5
(0,0,0)	(1,2,2)	3.3	3.3	1.5
(0,0,0,0,0,0)	(1,1,1,2,2,2)	6.0	5.4	1.5

Table 4.1: Results for 10^6 byte concurrent communications (s)

ology, we use a material-oriented approach to model packet propagation, as this approach is conducive to our analytic method. In the following we model the transfers on a packet level rather than on a byte level. However, the error is negligible in view of the large data transfers we will consider in the case study.

In order to model the pipelined packet propagation we model the entire transfer as a parallel operation (cf. Example 3.5.). Let $n_k = s \ldots r$ denote the index of the K nodes involved in the pipeline route. Then the PAMELA model is given by

```
$bmove(s, r, l)$ = par $(i = 1, l/120)$ {
                       seq $(k = 2, K-1)$ {
                           use($\{f_{n_k}, x_{n_k}\}, \tau_x$);
                           use($f_{n_k}, \tau_y$)
                       } ;
                       use($\{e_r, x_r\}, \tau_x$)
                   }
```

Note that this model ignores startup delay and the fact that $l \bmod 120$ represents a half packet on average. However, for large data communications this model suffices to accurately capture the effective bandwidth degradation when many virtual links are simultaneously active.

The contention model has been validated for many types of concurrent communications (equal message lengths) as well as random patterns (as discussed later on). From the point-to-point measurements as well as from the bandwidth measurements for concurrent communications it follows that $\tau_x = 108$ μs (link transfer) and $\tau_y = 73$ μs (intermediate forwarding). Table 4.1 shows a few typical results for (10^6 byte) data transfers involving only the first row of the mesh (nodes 0, 1, 2, and 3). In the table only the most significant digits are displayed for the ease of interpretation. The nodes that are simultaneously sending are expressed by the $\underline{s}$ vector, while the receivers are expressed by the $\underline{r}$ vector. Each pair (s_n, r_n) corresponds to one communication. Apart from the measured value T^m and the simulation result T the traditional static prediction T^t is listed to demonstrate the prediction errors that may occur. The results for T show that the *bmove* model is quite accurate for a first-order approximation. Only in a very few situations a limited deviation is measured (cf. last row). This optimistic prediction is due to the precise packet scheduling that is left undetermined in the Pamela model. In contrast to practice, this non-determinism sometimes leads to assuming a more efficient schedule than the actual implementation.

As mentioned earlier, the above model is merely intended as a first-order approximation of realistic communication bandwidth. Hence, many phenomena are not accounted for, such as the influence of concurrent communication in reverse direction on the (duplex) link performance, as well as the communication overhead on the CPU. Both phenomena may introduce errors in the order of 20 % as shown by measurements. The second phenomenon influences the *computation* performance, rather than communication performance (communication tasks by the CPU are run at high priority while computation threads have low priority). The first phenomenon, however, directly relates to *communication* performance. Duplex communication essentially degrades communication performance. Each data transfer on a link induces acknowledgement traffic on the reverse link (2 bits per 11 bit datum [74]). Consequently, when data transfers are performed concurrently in both directions, effective link bandwidth drops with approximately 18 %. For instance, a concurrent communication $(0,1) \rightarrow (1,0)$ of 10^6 bytes is indeed measured to take about 1.06 s instead of 0.9 s. □

4.5 Summary

In this chapter we have presented the approach to modeling parallel computer systems using Pamela. A key concept is the use of a material-oriented paradigm in combination with structured mutual exclusion operators because of the advantages in analytical sense as explained in the previous chapter. The constraints imposed by this modeling approach with respect to the ability to accurately model parallel computer systems are quite acceptable, a fact that may be illustrated by the longstanding use of queuing networks (i.e., structured mutual exclusion) in this area. As shown by the examples for most instruction models this approach does not introduce inherent limitations as long as the resource possession across the system components is perfectly nested and the synchronization protocols between the components can be expressed in terms of one thread of control. While the first requirement is typically met, the second one requires that components (i.e., different processes acquiring different components) cannot engage in a two-way synchronization as required when finite buffers are involved. The problem has already been introduced in the pipeline and producer-consumer example (Section 3.3) and has been discussed in the context of modeling switches with finite buffers. Of course, a (message-oriented) solution is to use individual **P/V** operators that provides the additional modeling power, at the expense of analyzability (i.e., appropriate for simulation). A solution more in the flavor of the Pamela methodology, however, is to accept the "under-specification" in view of the overall analysis technique (see Section 3.3). As mentioned earlier, only the principles have been outlined. In actual systems more detailed models may be necessary than the simple memory, cache, and switching models discussed thus far. However, our modeling examples do account for the most important behavioral aspects in terms of delay and contention at a level of abstraction as typically found in modeling approaches based on queuing networks. Indeed, Example 4.9 shows that even with simple models good accuracy can be obtained (also when used in actual applications as we shall see in Section 5.3).

The material-oriented approach to parallel systems modeling implies that a parallel program and machine are viewed in the same way in terms of a chain of re-active subroutines that are called by some root process that represents the algorithm. Each subroutine involves the use of resources and, possibly, the use of additional processes in order to express some form of asynchronous behavior (e.g., non-blocking communication). However, similar to resource usage, the use of subordinate processes is structured due to the **par** construct. This approach to machine modeling in terms of service layers (e.g., processors, switches, memories) has lead to the unification of distributed-memory and shared-memory machine modeling. While at software level the distributed-memory architecture is accounted for by an additional message-passing layer, at machine "load/store" level both machines only differ in terms of the processor-memory interconnection network. Note that this implies somewhat a departure from the "traditional" perspective on interconnection networks that distinguishes between shared-memory machine networks and distributed-memory machine networks. A representative example is the terminology due to Feng [42] in which a distinction is made between "dynamic" topologies, i.e., shared-memory machine networks where link paths are dynamic, and "static" topologies, i.e., distributed-memory machine networks where the (point-to-point) links cannot be re-configured for direct connection between other processors. From our perspective, both categories are dynamic, the only difference being that for distributed-memory machines the number of network switches equals the number of processors, the unification being the overall non-uniform memory access (NUMA) model. A framework for the unified description and analysis of machine networks is presented in [19].

In this chapter we have shown how the various networks can be expressed in terms of the same (load/store) modeling concepts. From a taxonomic perspective this material-oriented approach offers an interesting alternative to the description of parallel computer architectures which is traditionally "structure-oriented" (e.g., [35] and the references therein). Consider a cache connected to a memory. In terms of our "behavior-oriented" modeling approach we describe the architecture in terms of its (load/store) behavior. Let C, and M denote the Pamela models. Let M' denote the cached memory system. As shown in this chapter we can characterize the model in terms of some functional description formalism $M' = C(M)$ where C is a higher-order function that maps memory functionality between various layers in the memory hierarchy (same applies to switches). Such a description not only represents the structural link between the components but also describes its behavior. If we tacitly assume this "higher-order Pamela script" language to include operators for replication, selection, etc. (e.g., D^P represents P data processors), a machine could be represented in a concise (hierarchic) format, analogous to the graphical representation technique used in this chapter. This formalism would offer the ability to automatically compile performance models from the description, rather than deriving performance behavior through human-like inference only.

Chapter 5

Case Studies

5.1 Introduction

In contrast to the small examples presented thus far, in this chapter we shall discuss a number of more elaborate case studies in which we touch upon various aspects of the PAMELA methodology. In Section 5.2 we will demonstrate the application of the calculus in automatically compiling PAMELA models into analytic performance models using two case studies. While in these cases the PAMELA models themselves are not based on actual implementations, in Section 5.3 we will describe the modeling of a real application on a distributed-memory machine. in which we show that actual application performance can be modeled with relatively simple PAMELA models. Thus far, we have distinguished simulation modeling (numeric) and analytic modeling (symbolic). In Section 5.4 we explore alternative approaches to numeric performance prediction. Being a performance prediction technique, an important application of PAMELA is system optimization. In Section 5.5 we show how the PAMELA calculus can be used in parallel program synthesis. Finally, in Section 5.6 the main points are summarized.

5.2 Performance Compilation

5.2.1 Introduction

In Chapter 3 we have defined the analysis through which PAMELA models can be transformed into symbolic time domain models. In fact, the transformation process can be perceived as *compiling* time domain models from PAMELA models, which is a purely mechanical process. In order to show that the compilation procedure and the inherent reduction that can be applied in the process is not only feasible for toy problems only, we present two case studies. The first case study involves a matrix factorization on a multiple-bank shared-memory system. The second case study involves a matrix-vector multiplication on a distributed-memory machine. Parts of this work have appeared in [50, 51].

From an automated point of view some of the model reductions that will be applied in the course of the analysis may not be immediately obvious, in particular those that relate to the partitioning of the index space (see the next section). The information needed for the recognition of these few situations, however, can safely be assumed available as part of the knowledge of the (index) partitioning process.

5.2.2 Preliminaries

In many cases we consider a computation that is partitioned over M identical resources $m = 0 \ldots M-1$ (e.g., processors). Let S be a statement called for $i = a \ldots b$. Let $f(i) = ci + d$ be some linear index generated by S that references a resource m according to the partitioning function $m = \pi(i)$. Then the *visit count* on resource m is given by

$$V_m = \sum_{i=a}^{b} [\pi(ci+d) = m]$$

In the following we give reductions for block and cyclic partitioning functions. Some background is given in Appendix B.

Definition 5.1 *Let*

$$\pi(i) = \lfloor \frac{i}{B} \rfloor$$

denote the block partitioning function where

$$B = \lceil \frac{b-a+1}{M} \rceil$$

denotes the block size. Then

$$V_m = \beta_m - \alpha_m + 1$$

where

$$\alpha_m = \max(a, \lceil \frac{Bm-d}{c} \rceil), \quad \beta_m = \min(b, \lceil \frac{B(m+1)-d}{c} \rceil - 1)$$

□

Definition 5.2 *Let*

$$\pi(i) = i \bmod M$$

be a cyclic partitioning function. Then

$$V_m = \begin{cases} 0, & (m-d) \bmod \gcd(c, M) = 0; \\ \beta_m - \alpha_m + 1, & \textit{otherwise.} \end{cases}$$

where

$$\alpha_m = \lceil \frac{a-\iota_m}{\kappa} \rceil, \quad \beta_m = \lceil \frac{b+1-\iota_m}{\kappa} \rceil - 1, \quad \kappa = \frac{M}{\gcd(c, M)}$$

and ι_m is the smallest solution of the diophantine equation $ci + kM = m - d$ such that $\iota_m \geq a$. □

5.2.3 Matrix Factorization

In this case study we consider the factorization of an $N \times N$ matrix (a_{ij}) without pivoting, parallelized for a P processor shared-memory machine. The parallelization is based on a column-wise cyclic partitioning of the column updates over processors $p = 0 \ldots P-1$. The implementation is characterized by the following C-style pseudo code, i.e.,

```
for k = 0 .. N-2 {
    scale pivot column k;
    forall p = 0 .. P-1
        update columns k+1 .. N-1 assigned to p;
}
```

Let the underlying machine interface be defined in terms of just the following two instructions, i.e.,

- *flop*, that symbolizes all floating point operations, including possible local register traffic,
- $move(i,j)$, that represents a global memory load or store of the data word associated with the matrix element a_{ij}.

When (multiprocessing) overhead is ignored, the PAMELA model is given by

```
$L = $ seq $(k = 0, N-2)$ {
        $move(k,k)$; $flop$;
        seq $(i = k+1, N-1)$ {    ! scale
            $move(i,k)$;
            $flop$;
            $move(i,k)$
        } ;
        par $(p = 0, P-1)$
            seq $(t = t_\alpha, t_\beta)$ {    ! update
                $j = p + tP$;
                $move(k,j)$;
                seq $(i = k+1, N-1)$ {
                    $move(i,j)$;
                    $move(i,k)$;
                    $flop$; $flop$;
                    $move(i,j)$
                }
            }
    }
```

where t_α, t_β are given by

$$t_\alpha = \lceil \frac{k+1-p}{P} \rceil, \quad t_\beta = \lceil \frac{N-p}{P} \rceil - 1$$

as a result of Definition 5.2 ($t_\alpha = \alpha_p, t_\beta = \beta_p$). Let the multiprocessor's global memory consist of M interleaved memory banks $b_m, m = 0 \ldots M-1$ ($b_m = 1$). Without loss of generality, the interconnection is assumed ideal[1]. Then the machine model is given by

$$\begin{aligned} flop &= \mathbf{delay}(\tau_f) \\ move(i,j) &= \mathbf{use}(b_m, \tau_m) \end{aligned}$$

where τ_f and τ_m represent delays due to floating point and memory access instructions, respectively, and $m = (i + Nj) \bmod M$, thus taking a column-wise storage scheme into account.

First, we consider the application of Eq. (3.14) to the parallel section which we will denote L_k. With respect to $\varphi(L_k)$ by Eqs. (3.1) through (3.10) it holds

$$\varphi(L_k) = \max_{p=0\ldots P-1} \sum_{t=t_\alpha}^{t_\beta} (\tau_m + \sum_{i=k+1}^{N-1} (3\tau_m + 2\tau_f))$$

From Definition 5.2, it follows that the maximum number of columns assigned to a processor is given by

$$\max_{p=0\ldots P-1} (t_\beta - t_\alpha + 1) = \lceil \frac{n}{P} \rceil$$

in which $n = N - k - 1$. Hence $\varphi(L_k)$ reduces to

$$\varphi(L_k) = \lceil \frac{n}{P} \rceil (\tau_m + n(3\tau_m + 2\tau_f))$$

For the analysis of $\omega(L_k)$ (Eq. (3.12)) we must consider M resources b_m with workload $\delta_m, m = 0 \ldots M-1$ according to the cyclic distribution. For the purpose of explaining the analysis of δ_m, we will treat each *move* statement separately. The work load on memory bank b_m generated by the $move(k,j)$ statement in the t loop is given by

$$\delta_m^{(k,j)} = \sum_{p=0}^{P-1} \sum_{t=t_\alpha}^{t_\beta} [(k + Nj) \bmod M = m] \; \tau_m$$

where $j = p + tP$. By definition of t_α and t_β, this immediately reduces to

$$\delta_m^{(k,j)} = \sum_{j=k+1}^{N-1} [(k + Nj) \bmod M = m] \; \tau_m$$

By Definition 5.2, this form reduces corresponding to the parameters

$$a = k+1,\ b = N-1,\ c = N,\ d = k,\ \kappa = \frac{M}{\gcd(N, M)}$$

to the subtraction

$$\delta_m^{(k,j)} = (\lceil \frac{N - \iota_m}{\kappa} \rceil - \lceil \frac{k + 1 - \iota_m}{\kappa} \rceil + 1)[(r - k) \bmod \gcd(N, M) \neq 0] \; \tau_m$$

[1]The above model characterizes many practical system configurations. The case $M = 1$ also applies to a single (circuit-switched) bus system.

The work load due to both $move(i,j)$ statements is given by

$$\delta_m^{(i,j)} = 2 \sum_{p=0}^{P-1} \sum_{t=t_\alpha}^{t_\beta} \sum_{i=k+1}^{N-1} [(i+Nj) \bmod M = m]\ \tau_m$$

which, by definition of t_α and t_β, immediately reduces to

$$\delta_m^{(i,j)} = 2 \sum_{j=k+1}^{N-1} \sum_{i=k+1}^{N-1} [(i+Nj) \bmod M = m]\ \tau_m$$

By Definition 5.2, the i loop is reduced corresponding to the parameters

$$a = k+1,\ b = N-1,\ c = 1,\ d = Nj,\ \iota_m = m - Nj,\ \kappa = M$$

to the form

$$\delta_m^{(i,j)} = 2 \sum_{j=k+1}^{N-1} (\lceil \frac{N+Nj-m}{M} \rceil - \lceil \frac{k+1+Nj-m}{M} \rceil + 1)\tau_m$$

Similarly, the work load due to the $move(i,k)$ statement is given by

$$\delta_m^{(i,k)} = \sum_{j=k+1}^{N-1} (\lceil \frac{N+Nk-m}{M} \rceil - \lceil \frac{k+1+Nk-m}{M} \rceil + 1)\tau_m$$

Since $b_m = 1$ it follows

$$\omega(L_k) = \max_{m=0\ldots M-1} (\delta_m^{(k,j)} + \delta_m^{(i,j)} + \delta_m^{(i,k)})$$

Including the sequential fraction as well as the outer k loop, by Eq. (3.15) it follows

$$T^l = \sum_{k=0}^{N-2} (\tau_m + \tau_f + \sum_{i=k+1}^{N-1} (2\tau_m + \tau_f) + \max(\varphi(L_k), \omega(L_k)))$$

that reduces to

$$T^l = (N-1)(\tau_m + \tau_f) + \frac{N(N-1)(2\tau_m + \tau_f)}{2} + \sum_{k=0}^{N-2} \max(\varphi(L_k), \omega(L_k))$$

Effectively, the computation complexity of T^l is $\mathcal{O}(N^2 M)$ due to the ω term. In the spirit of the approximative nature of serialization analysis as well as in the interest of efficiency we investigate the quality of a less complicated approximation that results from assuming equal memory load balance due to the interleaving scheme. Formally, this corresponds to redefining the *move* model in terms of one single memory resource b with $b = M$ according to

$$move(i,j) = \mathbf{use}(b, \tau_m)$$

As a result of this simplification it immediately follows

$$\omega(L_k) = \frac{\sum_{p=0}^{P-1} \sum_{t=t_\alpha}^{t_\beta} (\tau_m + \sum_{i=k+1}^{N-1} 3\tau_m)}{M}$$

which, by definition of t_α and t_β immediately reduces to

$$\omega(L_k) = \frac{(n+3n^2)\tau_m}{M}$$

in which $n = N - k - 1$. Hence

$$\begin{aligned} T^l &= (N-1)(\tau_m + \tau_f) + \frac{N(N-1)(2\tau_m + \tau_f)}{2} + \\ & \sum_{n=1}^{N-1} \max(\lceil \frac{n}{P} \rceil \{\tau_m + n(3\tau_m + 2\tau_f)\}, \frac{(n+3n^2)\tau_m}{M}) \end{aligned}$$

Furthermore, if, at this stage, we neglect the dependency on n (i.e., approximating $\sum \max$ by $\max \sum$) we obtain the following expression[2], i.e.,

$$\begin{aligned} T^l &= (N-1)(\tau_m + \tau_f) + \frac{N(N-1)(2\tau_m + \tau_f)}{2} + \\ & \max(\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil \{\tau_m + n(3\tau_m + 2\tau_f)\}, \sum_{n=1}^{N-1} \frac{(n+3n^2)\tau_m}{M}) \end{aligned}$$

Without loss of precision, this $\mathcal{O}(N)$ expression can be further reduced to an $\mathcal{O}(1)$ expression using standard discrete mathematics. The reductions of the expressions

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil \quad \text{and} \quad \sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil n$$

are described in Appendix C.

Figures 5.1 and 5.2 each show three speedup results,

$$S = \frac{T_{P=1}}{T_{P\geq 1}}\ , \quad S^t = \frac{T^t_{P=1}}{T^t_{P\geq 1}}\ , \quad S^l = \frac{T^l_{P=1}}{T^l_{P\geq 1}}$$

based on the predictions T (simulation), T^t (traditional prediction, based on φ only), and our lower bound prediction T^l, respectively. Figure 5.1 shows the speedup for $M = 2$ while Fig. 5.2 shows the speedup for $M = 4$. In both cases $\tau_f = 10\tau_m$. The results clearly illustrate the added value of serialization analysis compared to the traditional approach, that yields far too optimistic predictions for large values of P. The M values have deliberately been chosen small to demonstrate the effects contention may have on performance (for $M = 1$, T^l and T practically coincide). For increasing M values the plots more or less blow up as a result of the simple scaling laws that can be directly derived by our approach to contention analysis. Let P^* denote the saturation point. For $P = P^*$, the total memory work load in the parallel loop equals the traditional execution time. For not too small problems this implies

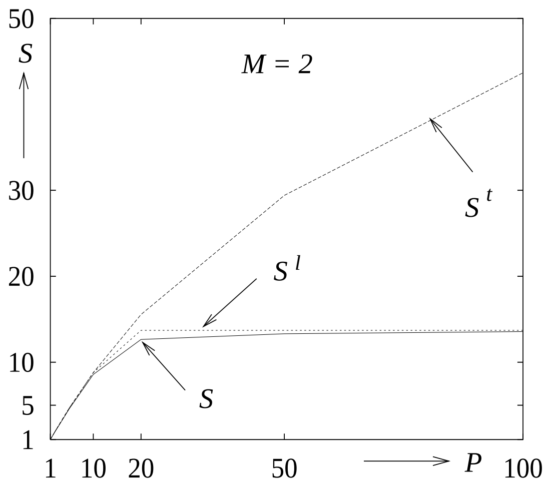
$$\frac{3\tau_m}{M} = \frac{3\tau_m + 2\tau_f}{P^*}$$

Hence,

$$P^* = (\frac{2\tau_f}{3\tau_m} + 1)M \approx 7.7M$$

which implies that (memory) performance degradation is determined by the ratio of P and M as can be seen from the figures.

[2]This establishes a lower bound of somewhat less quality. Results show, however, that the dependence on n is indeed negligible, resulting in a deviation that is less than a few percents over the total range.



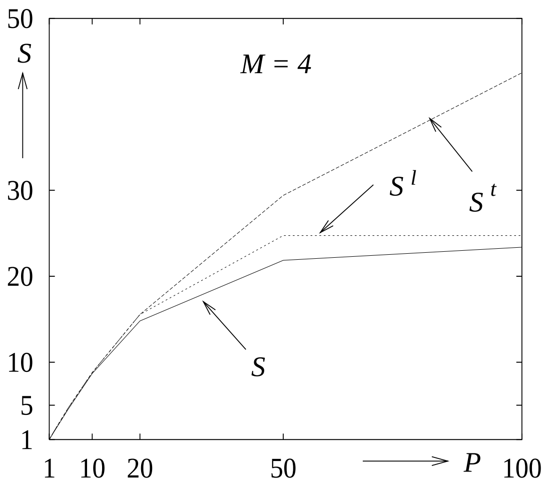


Figure 5.1: Speedup for $N = 100,\ M = 2$ Figure 5.2: Speedup for $N = 100,\ M = 4$

5.2.4 Matrix Multiplication

In this case study we consider an $N \times N$ matrix vector update $y = y + Ax$ on a P node multicomputer according to simple block partitioning in which we assume $P|N$. In order to emphasize the role of interprocessor communication (contention), an intentionally suboptimal *column-wise* partitioning is chosen, while x is evenly distributed over all nodes. Moreover, the implementation is based on a naive application of the "owner-computes" convention [26] to the *result* vector y (note, that less communication-intensive schemes exist for column-wise partitioning but, again, this is not the issue here). The result is characterized by the following C-style SPMD pseudo code (p is the node id), where $b = N/P$, and the indexing is kept in terms of the (original) global data space, for simplicity.

```
for i = 0 .. b*p-1
    for j = b*p .. b*(p+1)-1
        send(i/b,A[i][j]);
for i = b*(p+1) .. N-1
    for j = b*p .. b*(p+1)-1
        send(i/b,A[i][j]);
for i = b*p .. b*(p+1)-1 {
    for j = b*p .. b*(p+1)-1
        y[i] = y[i] + A[i][j] * x[j];
    for j = 0 .. b*p-1
        y[i] = y[i] + recv(j/b) * x[j];
    for j = b*(p+1) .. N-1
        y[i] = y[i] + recv(j/b) * x[j];
}
```

In this code, `send(q)` moves a datum from node p to node q, while `recv(q)` returns a datum sent by node q.

With respect to the communication of data we assume an *asynchronous* model. Also, we only model the `send` statement, assuming that the `recv` operation only accounts for synchronization and local data transfers. Furthermore, we assume that the data transfers are finished at the time the `recv` operations are executed, thus allowing condition synchronization to be ignored. (Note that this simplification is only made for the purpose of this example. Formally, we should take into account that the data transfers may not be finished in time.) Consequently, we can use a *send* model to completely account for the work load associated with the communication. Thus the Pamela model becomes

```
$L$ = par $(p = 0, P-1)$ {
        seq $(i = 0, bp-1)$
            seq $(j = bp, b(p+1)-1)$
                $send(i/b)$;
        seq $(i = b(p+1), N-1)$
            seq $(j = bp, b(p+1)-1)$
                $send(i/b)$;
        seq $(i = bp, b(p+1)-1)$ {
            seq $(j = bp, b(p+1)-1)$ {
                $flop$;
                $flop$
            } ;
            seq $(j = 0, bp-1)$ {
                $flop$;
                $flop$
            } ;
            seq $(j = b(p+1), N-1)$ {
                $flop$;
                $flop$
            }
        }
    }
```

which immediately reduces to

```
$L$ = par $(p = 0, P-1)$ {
        seq $(i = 0, bp-1)$
            seq $(j = bp, b(p+1)-1)$
                $send(i/b)$;
        seq $(i = b(p+1), N-1)$
            seq $(j = bp, b(p+1)-1)$
                $send(i/b)$;
        seq $(i = 0, 2bN)$
            $flop$
    }
```

With respect to the message-passing interface we assume a unidirectional ring of P point-to-point link resources $l_0 \ldots l_{P-1}$. In this arbitrary, simple model each scalar transmission involves a forwarding copy by each link processor in the path between sender and receiver. Hence we assume the following simple machine model:

$$\begin{aligned} flop &= \mathbf{delay}(\tau_f) \\ send(q) &= \mathbf{seq}\ (k = 0, K-1)\ \mathbf{use}(l_{(p+k)\mathrm{mod}P}, \tau_l) \end{aligned}$$

where $K = (P + (q - p))$ mod P denotes the number of links involved in the transmission (in this case study, forwarding costs at CPU level are ignored).

Since the parallel section is located on the outermost loop level we simply apply Eq. (3.14). Critical path analysis yields

$$\varphi = \max_{p=0\ldots P-1} (\sum_{i=0}^{bp-1} \sum_{j=bp}^{b(p+1)-1} \sum_{k=0}^{K-1} \tau_l + \sum_{i=b(p+1)}^{N-1} \sum_{j=bp}^{b(p+1)-1} \sum_{k=0}^{K-1} \tau_l) + \sum_{i=0}^{2bN} \tau_f$$

where

$$K = (P + \lfloor \frac{i}{b} \rfloor - p) \bmod P$$

Since it follows from Definition 5.1 that for any f

$$\sum_{i=0}^{bp-1} f(\lfloor \frac{i}{b} \rfloor) = b \sum_{p'=0}^{p-1} f(p')$$

φ reduces to

$$\varphi = b^2 \max_{p=0\ldots P-1} (\sum_{p'=0}^{p-1} (P + p' - p) \bmod P + \sum_{p'=p+1}^{P-1} (P + p' - p) \bmod P) \tau_l + 2bN\tau_f$$

which equals

$$\varphi = b^2 \max_{p=0\ldots P-1} (\sum_{p'=0}^{P-1} (P + p' - p) \bmod P) \tau_l + 2bN\tau_f$$

Finally, since

$$\sum_{p'=0}^{P-1} (P + p' - p) \bmod P = \sum_{p'=0}^{P-1} p' = \frac{P(P-1)}{2}$$

φ reduces to

$$\varphi = b^2 \frac{P(P-1)}{2} \tau_l + 2bN\tau_f$$

In order to derive ω we compute the work loads on links $l_0 \ldots l_{P-1}$. From the structure of L and *send* it immediately follows

$$\delta_l = \sum_{p=0}^{P-1} \left(\sum_{i=0}^{bp-1} \sum_{j=0}^{b-1} \sum_{k=0}^{K-1} [(p+k) \bmod P = l] + \sum_{i=b(p+1)}^{N-1} \sum_{j=0}^{b-1} \sum_{k=0}^{K-1} [(p+k) \bmod P = l] \right) \tau_l$$

where

$$K = (P + \lfloor \frac{i}{b} \rfloor - p) \bmod P$$

Again, since

$$\sum_{i=0}^{bp-1} f(\lfloor \frac{i}{b} \rfloor) = b \sum_{p'=0}^{p-1} f(p')$$

δ_l eventually reduces to

$$\delta_l = b^2 \sum_{p=0}^{P-1} \sum_{p'=0}^{P-1} \sum_{k=0}^{K'-1} \, [(p+k) \bmod P = l] \; \tau_l$$

where

$$K' = (P + p' - p) \bmod P$$

Given the fact that K' is cyclic with respect to p', it holds

$$\sum_{p'=0}^{P-1} f(K') = \sum_{p'=p}^{P-1+p} f(K')$$

Hence,

$$\delta_l = b^2 \sum_{p=0}^{P-1} \sum_{p'=p}^{P-1+p} \sum_{k=0}^{K'-1} \, [(p+k) \bmod P = l] \; \tau_l$$

Let $p'' = p' - p$. Then

$$K' = (P + p'') \bmod P = p''$$

and

$$\delta_l = b^2 \sum_{p=0}^{P-1} \sum_{p''=0}^{P-1} \sum_{k=0}^{p''-1} \, [(p+k) \bmod P = l] \; \tau_l$$

Now, we exploit the cyclic nature of the term $(p+k) \bmod P = l$ with respect to p. By Definition 5.2,

$$\sum_{p=0}^{P-1} \, [(p+k) \bmod P = l] = \lceil \frac{P-l+k}{P} \rceil - \lceil \frac{0-l+k}{P} \rceil = 1$$

and it follows

$$\delta_l = b^2 \sum_{p''=0}^{P-1} p'' \tau_l = b^2 \frac{P(P-1)}{2} \tau_l$$

As a result

$$\omega = b^2 \frac{P(P-1)}{2} \tau_l$$

which is independent of l. This, of course, agrees with the communication symmetry (note, however, that this knowledge has not been used in the above mechanical derivation). Hence

$$T^l = \max(b^2 \frac{P(P-1)}{2} \tau_l + 2bN\tau_f, b^2 \frac{P(P-1)}{2} \tau_l)$$

which implies

$$T^l = \varphi = N^2 (\frac{P-1}{2P} \tau_l + \frac{2}{P} \tau_f)$$

At first glance, the fact that $T^l = \varphi$ irrespective of P may be surprising. It is explained, however, if one realizes that the number of (link) resources scales linearly with the number of processors, thus maintaining balance.

The two plots in Fig. 5.3 each show two speedup results, i.e.,

$$S = \frac{T_{P=1}}{T_{P \geq 1}} \ , \quad S^l = \frac{T^l_{P=1}}{T^l_{P \geq 1}}$$

where T is the simulation value. The communication-to-computation ratio is parameterized according to

$$\lambda = \frac{\tau_l}{\tau_f}$$

where λ is chosen 0.1 and 0.5 in both figures, respectively. The results show a consider-

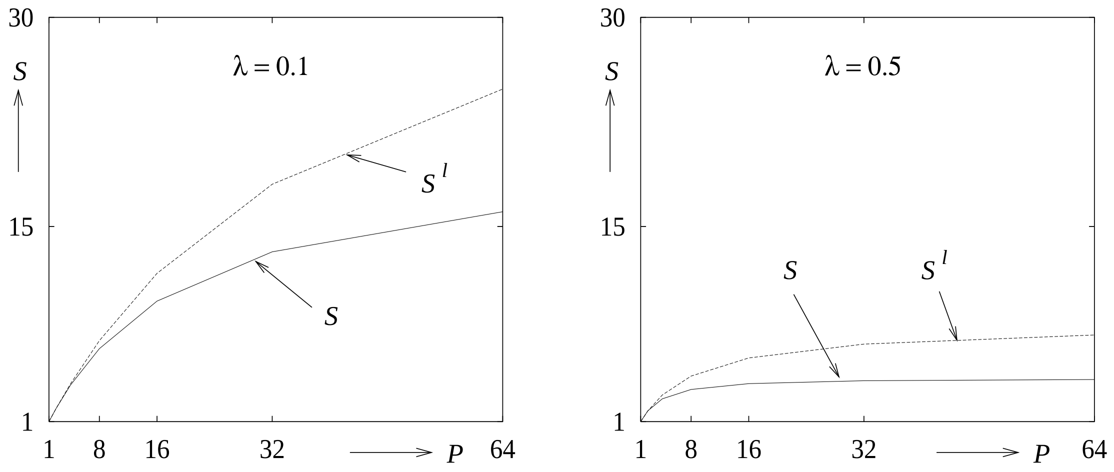


Figure 5.3: Simulation vs. predicted speedup for $N = 64$

able deviation between simulated and predicted speedup. This illustrates the fact that serialization analysis yields a lower bound that may still deviate considerably from T. For small P the deviation is small as the effect of contention is limited. For large P the communication dominates ($\mathcal{O}(1)$ versus $\mathcal{O}(P^{-1})$ computation) which implies $T^l = \varphi = \omega$. As in earlier examples for cases where $\varphi \approx \omega$ the deviation between T and T^l can be significant. This point will be elaborated further in Chapter 6. The fact that serialization analysis does not introduce an extra order term, like in the factorization case ($\mathcal{O}(N^3)$ in addition to the traditional $\mathcal{O}(N^3/P) + \mathcal{O}(N^2)$), indicates the problem's potential to run relatively contention-free on the given machine architecture. Indeed, it turns out that when the communication schedule of the algorithm is modified by simply reversing the direction in which the i loops are executed, yields completely conflict-free execution[3].

Although the analyses in both case studies somewhat reflect a human touch, the initial compilation procedure (i.e., compiling the raw, unreduced T^l model) can be mechanized. While compiling the visit count expressions is straightforward, the subsequent process of *reducing* them involves the repeated application of discrete calculus as shown in the workload analysis of the processor, memory, and link resources. In this reduction phase sometimes a judicious approximation (e.g., memory workload balance) can be helpful in the derivation of cheaper expressions, especially when the emphasis lies on asymptotic analysis. To which extent this reduction process can be mechanized is, of course, a different matter that lies outside the scope of this work. However, it is clear that the need for "reduction engines" in discrete mathematics is as general an issue as the need for reduction tools in standard calculus, something which is illustrated by the rapid developments in mathematical tooling. With respect to the implementation of the PAMELA compilation process this implies that PAMELA provides the tool to derive raw time expressions, intended to be reduced using separate mathematical tools.

5.3 Macro Data Flow Computation

5.3.1 Introduction

In order to validate our methodology in terms of actual measurements we present a case study in which the measured execution times of 15 synthetic programs on a distributed-memory machine are compared with our predictions based on both simulation and our analytic technique. One of the aims of the case study is (once again) to demonstrate the necessity of accounting for contention. It will be shown that traditional static techniques yield severe prediction errors. Parts of this section has appeared in [53].

The programs involve a macro data flow-style execution of random computation task graphs that are mapped on a 4×4 mesh partition of the Parsytec GCel T800 transputer system mentioned earlier in Example 4.9. The computation task graphs, that represent the user application, are SP graphs generated by a random generator that will be described in Chapter 6. For each graph the number of computation tasks is given by $N = 100$. Each task t_i, $i = 1, \ldots, N$ is statically mapped onto a random processor p_i according to a uniform distribution between 1 and $P = 16$ ($\underline{p}$ denotes the task mapping vector). Thus, on average, 100/16 tasks are mapped onto the same processor. Each task is executed by a separate (lightweight) thread that is scheduled dynamically by the node's run-time kernel. In order to enable true data flow execution, after each task has been executed, the (same) produced data set is asynchronously broadcasted (non-blocking communication) to each successor task (thread) except when a successor resides locally. The dynamic resource

[3]In terms of the original pseudo code this implies the sequence $i = bp - 1, \ldots, 0$, etc. Consequently, the successive resource access sequences of the *send* calls do no longer conflict. This phenomenon was first observed by Jonkers [79].

sharing approach is partly inspired by the fact that recent technological developments in the use of lightweight threads makes it increasingly justified[4] to use *dynamic* scheduling at the processor level, both for computation and communication tasks (i.e., data flow, using non-blocking send/recv calls) while the overall mapping is static. Furthermore, the parallel slackness (multiple concurrent tasks per processor, cf. Valiant's BSP model [151]) increases the average utilization of the processing and communication resources. A typical example of the proposed approach is described in [96] where task graphs representing finite element computations are statically mapped (based on a domain decomposition) such that the number of tasks per node is much larger than one.

Due to the dynamic approach towards task computation and communication, the case study (intentionally) provides an excellent example of the added value of serialization analysis compared to conventional static prediction techniques. While static analysis inherently ignores the additional delay incurred by tasks sharing the same processor, our approach naturally accounts for this delay by modeling task execution in terms of "processor contention". Apart from processor contention, the use of non-blocking communication introduces the possibility of link contention as multiple task communications may share the same communication links. Again, conventional static analysis makes no provision to account for the additional queuing delay, that may easily dominate performance (as will be shown, later on). In our aim just to demonstrate the impact contention analysis may have, we simply consider coarse grain task execution where each task entails a large amount of computation ($\mathcal{O}(10^6)$ floating point operations) as well as communication ($\mathcal{O}(10^6)$ byte transfers). As will be shown, without loss of accuracy we can therefore concentrate on computational and communication *bandwidths* rather than startup times (and other sources of overhead), which simplifies the modeling discussion.

In order to allow for the execution of arbitrary task graphs a simple, generic SPMD interpreter program is developed that accepts a task graph description file, executes the task graph, and records the execution time. Thus, the 15 random programs that are modeled is essentially the SPMD program, instantiated with each of the 15 random task graphs. The execution time recorded from the SPMD program is compared to our predictions. The task graph description is based on a simple abstract data type. For each task $i = 1, \ldots, N$ the data type specifies

- `pred(i,k)`: predecessors (`k = 1 .. fanin(i)`)
- `succ(i,k)`: successors (`k = 1 ..  fanout(i)`)
- `node(i)`: processor it is mapped onto (p_i)
- `work(i)`: amount of computation (w_i)
- `size(i)`: amount of data produced (l_i)

After each task has executed, the (same) produced data set is broadcasted to each successor except in case the successor resides locally. In the following, we sketch the global architecture of the SPMD application where `p` denotes the node index.

[4]Note that our only interest is just a reasonably realistic case study, not to prove any point with respect to the interesting issue of task graph mapping or scheduling.

```
spmd(p):
    for i = 1 .. N
        if node(i) = p
            create task(i);

task(i):
    for k = 1 .. fanin(i)
        j = pred(i,k);
        nrecv(j,size(j));
    for k = 1 .. fanin(i)
        j = pred(i,k);
        await(j);
    comp(i);
    for k = 1 .. fanout(i)
        j = succ(i,k);
        nsend(j,size(i));
```

The data flow style implementation assumes a non-blocking message-passing interface based on individual (logical) channels (links) between each task pair (i,j). The `nrecv` loop enables concurrent reception of input data, while the `await` loop implements the task's synchronization barrier. The `nsend` loop yields a concurrent broadcast of the task's output data. Apart from programming convenience this approach retains maximum potential parallelism in the communication structure (whether or not actually supported is machine-dependent). Thus unnecessary sequentialization at the task level is avoided (analogous to the dynamic task scheduling approach). When, as usual, the implementation is based on multiplexing logical links on a smaller number of physical links, the effective bandwidth reduction is naturally accounted for in terms of a link contention model.

The message-passing interface used for the implementation is based on the "virtual link" service, that provides a dedicated logical channel between a sender and receiver task. The virtual link topology needed to connect predecessor and successor tasks is setup in the prologue of the actual program. (Thus link setup times are not measured.) The communication mode selected is "asynchronous" in terms of the machine interface which, in reality, implies a non-blocking mode[5]. The communication mode selected does not involve buffer copying. In terms of the message-passing system interface the above three communication functions are implemented by `ARecv`, `ASync`, and `ASend` calls, respectively [115].

5.3.2 Computation Model

Let G denote the task graph to be executed, consisting of tasks $t_i, i = 1, \ldots, N$. From the SPMD program it follows that the execution of G is modeled by

$$L = \textbf{par}\ (p = 1, P)\ \textbf{par}\ (i = 1, N)\ \textbf{if}\ (p_i = p)\ task(i)$$

in which $task(i)$ models the task thread. The first parallel section is due to the SPMD parallelism while the second parallel section is due to the simultaneous creation of the

[5]The difference between asynchronous and non-blocking sends has been discussed in Chapter 4.

task threads at initialization time. Clearly, we may also write

$$L = \textbf{par}\ (i = 1, N)\ task(i)$$

We do not account for the overhead involved with the creation of threads or virtual links since the measurement only involves the actual task execution times.

Based on the fact that each `nrecv` and `nsend` call is implemented by a separate thread [115] (also see Example 4.8), the task model is given by

```
$task(i) =$ **par** $(k = 1, fanin(i))$ {
                $j = pred(i,k);$
                $recv(j, l_j)$
            } ;
            $comp(p_i);$
            **par** $(k = 1, fanout(i))$ {
                $j = succ(i,k);$
                $send(j, l_i)$
            }
```

where *send* and *recv* represent the actual (synchronous) communication tasks (addressed in terms of task indices), and *comp* denotes the computation model in terms of the appropriate processor. Note that the implicit barrier in the first **par** construct accounts for the explicit barrier in the program code (the `await` loop). The parallel send and recv sections express the concurrency involved in the non-blocking communications. Note that while a non-blocking send essentially involves a separate thread of control, a non-blocking receive actually may just involve a simple subroutine (e.g., some initialization for the future communication). Consequently, it would seem that the above parallel model for the `nrecv` loop might not always reflect the actual implementation. In reality, however, the approach does cover the complete spectrum of thread-based as well as subroutine-based implementations. In case of a subroutine, the work involved is simply charged to a single resource (processor). Consequently, the parallel loop will automatically be serialized (both in terms of simulation *and* serialization analysis) as if the calls were made in sequence. Hence, the performance result is essentially the same.

Because all the receives are already posted during task initialization, the data transfers initiated by the send calls effectively execute asynchronously (i.e., do not incur any additional condition synchronization delay), corresponding to the philosophy of macro data flow. Hence, the communication interface model is effectively given by

$$\begin{aligned} send(j,l) &= bmove(p_i, p_j, l)\ ;\ \textbf{signal}(c_{ij}) \\ recv(j,l) &= \textbf{wait}(c_{ij}) \end{aligned}$$

that corresponds to the model for one-only (unbounded buffer) communication (Section 4.3) in which c_{ij} corresponds to the specific communication channel ("virtual link") between task i and j. The *bmove* model represents the actual data transfer activity. Consequently, the task model is given by

```
$task(i) =$ par $(k = 1, fanin(i))$ {
        $j = pred(i, k)$;
        wait$(c_{ij})$
    } ;
    $comp(p_i)$;
    par $(k = 1, fanout(i))$ {
        $j = succ(i, k)$;
        $bmove(p_i, p_j, l_j)$;
        signal$(c_{ij})$
    }
```

Due to the message-passing synchronization, at the task level, L is topologically similar to G. (This, of course, corresponds to the fact that the SPMD program correctly executes G.) The only difference is that it is expressed in a message-oriented style rather than a procedure-oriented style, as described earlier. For the purpose of analysis, however, it is more attractive to use G as the basis for a model for the SPMD message-passing program[6] rather than the message-passing version of L. For instance, consider the simple task graph $G = t_1\ ;\ t_2$. According to the message-passing approach it follows

$$L = \{comp(p_1)\ ;\ bmove(p_1, p_2, l_1)\ ;\ \mathbf{signal}(c_{12})\} \parallel \{\mathbf{wait}(c_{12});\ comp(p_2)\}$$

However, as G is an SP graph, a material-oriented approach simply yields

$$L = comp(p_1)\ ;\ bmove(p_1, p_2, l_1)\ ;\ comp(p_2)$$

Thus in the procedure-oriented approach L is constructed by simply expanding every arc in G with a *bmove* model. Consequently, L can be directly compiled into an analytic performance model based on the application of Eq. (3.15).

The *comp* model represents the actual task computation. For the purpose of the case study each task just executes a meaningless computation according to to

```
for i = 1 .. work(i)
    r = 1.0 * A[i mod 1000];
```

that generates integer and floating point computations as well as memory access, the total work load parameterized by w_i. In our aim to evaluate the prediction accuracy in the face of processor and network contention, we refrain from modeling the above code in detail and simply measure it as a whole. For the amount of work we consider ($10^4 \ldots 10^6$ loops) the execution time increases linearly with w_i according to 6.1 μs per iteration[7]. The execution time including queuing delay due to processor sharing is accurately expressed by the following simple model (expressed in μs)

$$comp(p_i) = \mathbf{use}(cpu_{p_i}, 6.1 w_i)$$

[6]Note that it might seem obvious to use G for the performance model in the first place. Formally, however, we must deal with the fact, that L must represent the SPMD *program* of which the performance is measured, not its data *input*. Hence, we must adopt this line of reasoning.

[7]Without any form of (compile-time) optimization. In the coefficient, the (small) effect of multithreading overhead is automatically accounted for since during the measurement the above code is run as a thread.

where cpu_{p_i} is of PS-type. As an example, consider the following 6-tasks graph

$$G = t_1 \ ; \ \textbf{par} \ (i = 2, 5) \ t_i \ ; \ t_6$$

with a task mapping given by $\underline{p} = (p_1, \ldots, p_6)$. According to the procedure-oriented modeling procedure, the Pamela model of the SPMD program is given by

```
L = $comp(p_1)$;
    par $(i = 2, 5)$ {
        $bmove(p_1, p_i, l_1)$;
        $comp(p_i, \tau_i)$;
        $bmove(p_i, p_6, l_i)$
    } ;
    $comp(p_6, \tau_6)$
```

Note that any SP graph G always maps to an SP model L. Let $\underline{p} = (0, 0, 1, 0, 1, 0)$. When the *bmove* model is ignored (discussed in the next section) Eq. (3.15) yields

$$T^l = 6.1(w_1 + \max(w_2, w_3, w_4, w_5, w_2 + w_4, w_3 + w_5) + w_6)$$

Indeed, for large computations and zero communication, the above prediction not only equals T (i.e., the simulation result) but also closely matches the actual execution time measured (within a few percents, as shown later on). Note that even for this simple example conventional static analysis may already yield an error up to 100 %.

5.3.3 Communication Model

As discussed in Example 4.9 the *bmove* model is given by (expressed in μs)

```
$bmove(s, r, l)$ = par $(i = 1, l/120)$ {
                       seq $(k = 2, K - 1)$ {
                           use$(\{f_{n_k}, x_{n_k}\}, 108)$;
                           use$(f_{n_k}, 73)$
                       } ;
                       use$(\{e_r, x_r\}, 108)$
                   }
```

where s and r denote sender and receiver, respectively.

Because of the simultaneous resource usage, the above model (and hence L) is not amenable to the application of Eq. (3.15). Thus, we consider an approximation using transformation Eq. (3.17) discussed in Section 3.6 (see Fig. 3.10). It follows

$$\textbf{use}(\{f_{n_k}, x_{n_k}\}, 108) \ ; \ \textbf{use}(f_{n_k}, 73) \rightarrow \textbf{use}(f_{n_k}, 108) \parallel \textbf{use}(x_{n_k}, 108) \ ; \ \textbf{use}(f_{n_k}, 73)$$

while the second **use** term immediately reduces to

$$\textbf{use}(\{e_r, x_r\}, 108) \rightarrow \textbf{use}(x_r, 108)$$

$\underline{s}$	$\underline{r}$	T	T^l	T^t
(0)	(1)	0.9	0.9	0.9
(0)	(2)	1.5	1.5	1.5
(0,0)	(1,1)	1.8	1.8	0.9
(0,0)	(2,2)	3.0	3.0	1.5
(0,0)	(1,2)	1.8	1.8	1.5
(0,0,0)	(1,1,2)	2.7	2.7	1.5
(0,0,0)	(1,2,2)	3.3	3.0	1.5
(0,0,0,0,0,0)	(1,1,1,2,2,2)	5.4	5.4	1.5

Table 5.1: Results for 10^6 byte concurrent communications (s)

since e_r is not used anywhere else in the **par** expression. After applying the additional (lower bound) reduction

$$\mathbf{use}(f_{n_k}, 108) \parallel \mathbf{use}(x_{n_k}, 108)\ ;\ \mathbf{use}(f_{n_k}, 73) \rightarrow \mathbf{use}(f_{n_k}, 181) \parallel \mathbf{use}(x_{n_k}, 108)$$

the *bmove* model is approximated by

```
$bmove(s, r, l)$ = par $(i = 1, l/120)$ {
                        seq $(k = 2, K - 1)$ {
                            use$(f_{n_k}, 181)$ ||
                            use$(x_{n_k}, 108)$
                        } ;
                        use$(x_r, 108)$
                   }
```

Effectively, this model accounts for the fact that T cannot be less than the largest work load on either an f or x (or e) server. Note that this leads to a somewhat less tight bound because the approximate model involves less synchronization constraints than the earlier model. For the experiments described in Example 4.9 (cf. Table 4.1) Table 5.1 shows a comparison of T (based on the above model) and T^l that is based on the approximation. The static prediction T^t is added for reference. The table shows that, the approximation is quite acceptable, especially in view of the overall modeling approximation in which various communication aspects have been ignored (see Example 4.9). Hence, the approximation is used for the overall prediction experiment.

5.3.4 Results

In this section we present the measurement results for the execution of the 15 random SP graphs $G_1 \ldots G_{15}$ on the 4×4 transputer mesh. As mentioned earlier, each graph comprises $N = 100$ tasks that are randomly i.i.d. (independent identically distributed) uniformly over the 16 processors. The computational work load w_i is i.i.d. uniformly over $[10^4, 10^6]$ which corresponds to an average total problem size of 305 s in terms of the `comp` code presented earlier. In order for the communication to have a significant impact, the data size sent by each task to its successors is also i.i.d. uniformly over $[10^4, 10^6]$ which

G	T^m	T	T^l	T^t
G_1	118.7	114.7	83.7	25.9
G_2	93.9	92.5	57.4	21.2
G_3	95.6	92.8	63.1	25.8
G_4	94.1	87.4	61.2	31.5
G_5	73.4	70.9	46.8	30.3
G_6	105.8	103.9	58.3	58.3
G_7	98.4	87.0	52.2	47.4
G_8	89.2	87.1	59.2	52.5
G_9	87.6	84.4	65.8	65.7
G_{10}	109.5	106.4	79.8	79.5
G_{11}	141.2	138.4	120.5	107.6
G_{12}	149.5	144.8	126.0	125.0
G_{13}	165.9	163.2	140.4	140.2
G_{14}	172.3	171.0	165.4	165.4
G_{15}	188.6	186.8	173.4	174.3

Table 5.2: T^m vs. predictions

G	T_f^m	T_f	T_f^t	T_c^m	T_c	T_c^t
G_1	38.7	38.9	21.3	94.7	88.7	5.3
G_2	46.3	46.5	17.2	67.3	61.9	4.3
G_3	36.1	36.2	22.8	77.6	69.8	4.2
G_4	48.3	48.1	27.4	73.2	64.4	5.8
G_5	54.9	55.3	25.4	32.3	30.0	4.8
G_6	79.1	79.4	47.6	42.5	39.4	10.6
G_7	58.2	58.4	39.7	49.2	42.7	9.8
G_8	63.7	64.0	44.2	36.2	34.0	11.2
G_9	75.7	76.2	56.4	19.5	17.5	10.2
G_{10}	82.6	82.8	65.4	49.3	45.6	16.4
G_{11}	114.8	115.4	91.2	37.3	33.2	19.6
G_{12}	113.0	113.5	103.3	44.8	42.7	22.2
G_{13}	130.8	131.1	119.3	41.2	38.5	23.5
G_{14}	138.2	139.1	136.3	44.2	42.5	28.9
G_{15}	145.8	146.5	145.0	47.5	46.1	33.2

Table 5.3: Results for f-mode and c-mode

corresponds to an average communication delay between 450 ms and 750 ms per *isolated* transfer. The 15 graphs are generated such that they cover the entire spectrum from relatively parallel graphs (corresponding to low G indices) to relatively sequential graphs (corresponding to high G indices). Table 5.2 summarizes the main results for each of the 15 programs. T^m denotes the measured execution time (s). T denotes the simulation result (s) of the PAMELA model L. T^l denotes the result (s) of applying Eq. (3.15) to the PAMELA model version based on the approximate *bmove* model (in terms of simulation, the overall difference with the exact model is practically negligible). The total number of resources involved in the simulation and analysis is $M = 144$ ($p = P = 16, f = x = 4P = 64$). The T^t value (i.e., φ) has been included to demonstrate the (severe) prediction error of traditional static analysis.

The results show that the performance of the SPMD program is indeed captured by the PAMELA model with reasonably good accuracy. On average, T under-estimates T^m by about 4 % which is entirely due to the fact that the communication model ignores the effects of reverse communication and the additional CPU load (as discussed in Example 4.9; this will also be shown in the next table). The results for T^l follow the general trend as discussed earlier in the MRM, pipeline, matrix factorization and multiplication case studies. For $\omega \gg \varphi$ (low G indices) as well as for $\omega \ll \varphi$ (high G indices) T^l approaches T, while the deviation is maximal when both terms are of the same order. An extensive discussion of this important phenomenon will be presented in the next chapter.

In order to evaluate the PAMELA model in more detail, each of the 15 graphs is also executed under a mode f in which all communication (except task synchronization) is switched off ($\underline{l} = \underline{0}$), and a mode c, in which all computation has been disabled ($\underline{w} = \underline{0}$). Thus each original measurement T^m is complemented by a communication-less version T_f^m and a computation-less version T_c^m, representing both ends of the communication spectrum. Table 5.3 shows a comparison of T^m, T, and T^t for both execution modes. The results show that the inaccuracy of T is indeed due to the *bmove* model as explained before. The average accuracy of the PAMELA model for f-mode execution lies well within 1 % (indicating the correctness of the *comp* model), whereas the c-mode model underestimates communication delay by 8 % on average. Note that the results automatically demonstrate the general validity of the (approximate) communication model for various random concurrent communication patterns (in addition to the test patterns shown in Example 4.9). Finally, note that for a high communication density the error in T^t_c becomes quite spectacular.

5.4 Simulation Revisited

5.4.1 Introduction

Thus far, a distinction has been made between simulation and the lower bound analytic technique. As mentioned in Chapters 2 and 3, simulation relates to direct model evaluation in the PAMELA domain while the analytic technique is based on evaluation in the time domain model that is compiled from the PAMELA model. While both modes are equivalent for contention-free models in terms of the evaluation result ($T = \varphi$), in the presence of contention both techniques differ in the way mutual exclusion is approached ($T \geq T^l$). Although, especially for contention models, simulation is generally much more time-consuming than the analytic technique (highly iterative procedure, process overhead), there exist cases in which the actual computation time involved with simulation is comparable (in big-O terms) with the lower bound technique while the result (T) is essentially better. This coincides with the fact that there are models that are amenable to an alternative, numeric technique rather than just the lower bound approach. Like in the case of simple critical path analysis, whether the evaluation mode should be coined simulation or analytic has become more or less a technical matter. In this section we shall explore the difference between the analytic technique and simulation in somewhat more detail.

As an introduction to the problem we start with an application of our analytic technique by compiling a performance model from a dynamic message-passing program.

Example 5.1 Consider a simple data parallel operation on an N element vector $\underline{x}$ according to the pseudo code

```
forall i = 0 .. N-1
    y[i] = comp(x[f(i)]);
```

where `comp` denotes some unary computation and f denotes some index function. Consider a simple SPMD parallelization of the above program on a P processor distributed-memory machine using a cyclic partitioning scheme (x and y are aligned). In the following we will derive a simple performance model of the SPMD code.

For the purpose of the example, we assume a naive SPMD code generation model for an asynchronous communication interface according to the following pseudo code

```
for i = 0 .. N-1 {
    s = f(i) mod P;
    r = i mod P;
    if r != p and s = p
        send(r,y[f(i)]);
    if r = p and s != p
        y[i] = comp(recv(r));
    if r = p and s = p
        y[i] = comp(y[f(i)]);
}
```

where p denotes the processor index ("owner-computes" model), the computation and communication are still expressed in terms of the global data (index) space (cf. matrix multiplication in Section 5.2). Furthermore, note that various optimizations have been (intentionally) neglected (e.g., index space partitioning, message vectorization) given the objective of the example.

Let the asynchronous message-passing interface be given by the following simple model (unbounded buffer, see Section 4.2)

$$\begin{aligned} send(r,a) &= \mathbf{signal}(c_{si}) \\ recv(s,a) &= \mathbf{wait}(c_{si}) \end{aligned}$$

where c_{si} represents the communication channel between sending processor s and receiver r during iteration i (the address a is immaterial). Thus the processor network is thought ideal. Let the computation model be simply given by

$$comp = \mathbf{delay}(\tau_c)$$

Consequently (ignoring various overhead terms as usual) the PAMELA model of the SPMD program is given by

```
$L = \mathbf{par}\ (p = 0, P-1)$
        $\mathbf{seq}\ (i = 0, N-1)$ {
            $s = f(i) \bmod P$;
            $r = i \bmod P$;
            $\mathbf{if}\ (r \neq p \wedge s = p)$
              $\mathbf{signal}(c_{pr})$;
            $\mathbf{if}\ (r = p \wedge s \neq p)$ {
              $\mathbf{wait}(c_{sp})$;
              $\mathbf{delay}(\tau_c)$
            } ;
            $\mathbf{if}\ (r = p \wedge s = p)$
              $\mathbf{delay}(\tau_c)$
        }
```

Note that the above model is not an SP model. Hence, we cannot simply apply Eqs. (3.4) through (3.10). However, since the model is amenable to our functional analysis approach (as a result of the absence of $\mathbf{P}/\mathbf{V}$ operators), we can use the basic mapping rules as given in Eq. (3.1) through Eq. (3.3). By associating a (single assigned) variable with each task (**if**) statement, the following set of (conditional) equations is derived.

$$\begin{aligned}
T &= \max_{p=0\ldots P-1} r_p \\
\forall p:\ r_p &= r_{p,N-1} \\
\forall p,i:\ r_{p,i,0} &= \begin{cases} 0, & i=0; \\ r_{p,i-1,2}, & \text{otherwise.} \end{cases} \\
\forall p,i:\ c_{si} &= r_{p,i,0} \\
\forall p,i:\ r_{p,i,1} &= \begin{cases} \max(r_{p,i,0}, c_{si}) + \tau_c, & r=p \wedge s \neq p; \\ r_{p,i,0}, & \text{otherwise.} \end{cases} \\
\forall p,i:\ r_{p,i,2} &= \begin{cases} r_{p,i,1} + \tau_c, & r=p \wedge s=p; \\ r_{p,i,1}, & \text{otherwise.} \end{cases}
\end{aligned}$$

where $s = f(i) \bmod P$ and $r = i \bmod P$. □

While, the above system of equations can be evaluated by obeying their data dependencies (rearranging some equations), it is clear that the above model is not amenable to a straightforward computation on a sequential machine. The following algorithm shows a simple technique that dynamically evaluates the data dependencies between the above equations while evaluating the equations where possible.

```
while ($true$) {
    for ($p = 1 \ldots P-1$) {
        for ($i = 1 \ldots N-1$) {
            if ($\neg e_{p,i}$) {
                if ($r \neq p \wedge s = p$)
                    $c_{si} = r_p$;
                if ($r = p \wedge s \neq p$) {
                    if ($c_{si} \neq -1$) {
                        $r_p = \max(r_p, c_{si})$;
                        $r_p = r_p + \tau_c$;
                    }
                    break;
                }
                if ($r = p \wedge s = p$)
                    $r_p = r_p + \tau_c$;
                $e_{p,i} = true$;
                $e = e + 1$;
                $done = (e = PN)$;
            }
        }
    }
}
for ($p = 1 \ldots P-1$)
    $T = \max(T, r_p)$;
```

The algorithm scans each equation according to the above p and i loop and tests whether the equations corresponding to loop instance (p, i) (i.e., $r_{p,i,0}, \ldots, r_{p,i,2}$) can be evaluated (using the condition $e_{p,i}$ corresponding to the equations $r_{p,i,0}, \ldots r_{p,i,2}$). The primary time variable is r_p. All variables are assumed initially zero, except the c_{si} that have an initial value of -1 to encode their definition status (task completion). When a loop instance (p, i) cannot be completed, the algorithm switches context (using the break construct that has similar semantics as the construct in C) to another instance of the higher level loop (in this case the p loop). The top level while loop guarantees that all equations are evaluated. Note that the above computation is guaranteed to terminate since the SPMD program may be assumed deadlock-free (i.e., the program is assumed correct). The above algorithm has been verified to produce the same results as those obtained by direct simulation of the PAMELA model.

Clearly the resemblance between the structure of the analysis algorithm and the original PAMELA model is obvious. In fact, the sequential computation can be compiled from the PAMELA model using a mechanical scheme where (apart from the extra variables) each **par** maps to a for loop along with a while-break mechanism in order to correctly resolve the data dependencies between each (task) equation. Consequently, in this example the analysis method may well be thought of as simulation, although the sequential computation is formally an implementation of the (intermediate) *analytic* model on a sequential computer. It may therefore seem that the above analytical process

$$SL \rightarrow DG \rightarrow SE \rightarrow SC$$

where we now explicitly include the final sequential *computation* SC (SL denotes the original PAMELA model), is a somewhat elaborate way of describing the process

$$SL \rightarrow SC$$

which is quite comparable to a simulation approach. However, unlike real simulation, everything is now directly compiled into the time domain instead of interpreted by some run-time state machine for the ultimate time domain evaluation. Of course, there are more differences. The "analytical" route only applies to the subset of models that do not include the notion of state (as discussed in Chapter 3). Furthermore, because of this, a much more optimized computation can be generated than the general simulation approach that uses an interpreter. The most significant example is the compilation of SP models into just one single expression.

5.4.2 Alternative Techniques

As mentioned earlier, the major differences between the "analytic" and "simulation" technique are determined by

- evaluation domain
  While simulation corresponds to interpretation in the PAMELA domain using some state machine (e.g., a discrete-event simulator), the analytic method corresponds to interpretation (evaluation) in the time domain based on a (symbolic) model that is *compiled* from a PAMELA domain model.

- determinism
  While the numerical simulation result typically represents a mere draw from the result distribution due to model non-determinism of time and control flow (e.g., mutual exclusion), the analytical method yields a deterministic result (e.g., for non-deterministic models a mean value).

As shown above, in practice the differences with respect to the evaluation domain tend to be less rigid as, like simulation, the analytic approach typically gives rise to the compilation of an "interpretation" system that controls the evaluation order.

In this section we will simply unify the notion of analytic technique and simulation by assuming a run-time system that maintains a list of (PAMELA) processes of which the next time domain equation (i.e., PAMELA statement) can be executed. By introducing the list as an abstract data type, the distinction between the analysis algorithm and simulator has practically disappeared (note that many analytic methods or "algorithms" are based on lists, e.g., task event lists).

In the following, we will simply refer to the technique as "simulation", whether or not the actual evaluation takes place in the PAMELA domain or not. In general, it is tacitly assumed that the run-time system is compiled as a part of the model rather than linked based on a PAMELA domain interpreter (i.e., the classical simulator). Although from this perspective the choice of terminology has become arbitrary, we use the term simulation rather than analytic because of the fact that, given the above criteria, simulation admits a possibly non-deterministic result although in many cases the outcome is (practically) exact as we shall see.

Up until now, we have considered contention-free models where a comparison between simulation mode and analytic mode is appropriate due to the fact that in both cases T is computed. In the following we will consider more general cases involving contention. As in this dissertation we concentrate on the problem of analyzing mutual exclusion, we will (again) assume that non-determinism due to task variance and/or conditional control flow is negligible.

Although, from a static (symbolic) analysis point of view, contention models do induce non-determinism with respect to the *symbolic* compilation, from a *numeric* (simulation) point of view, for many contention models, the actual non-determinism is non-existent. Hence, for these models a *single* simulation run suffices to provide T (the matrix factorization model is an example of this). Consider the following model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{delay}(\tau_1) \; ; \; \mathbf{use}(r, \tau_2) \\ L_2 &= \mathbf{delay}(\tau_3) \; ; \; \mathbf{use}(r, \tau_4) \end{aligned}$$

where r is an FCFS-type resource. In order to compile a symbolic time domain model for T we *must* evaluate the precedence order between both contending tasks which depends on the actual value of τ_1 and τ_3 (unless, of course, we were willing to accept a conditional model for T, which, in general, is not a practical option). On the other hand, simulation, essentially being a numeric evaluation technique, simply evaluates the precedence relation dynamically in the course of its evaluation algorithm. For instance, the following model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{delay}(1) \ ; \ \mathbf{use}(r,2) \\ L_2 &= \mathbf{delay}(2) \ ; \ \mathbf{use}(r,1) \end{aligned}$$

yields the deterministic result $T = 4$. Note that the evaluation algorithm can equally be called analytic as the algorithm only manipulates a simple event list and yields a deterministic result at basically the same time complexity as a critical path algorithm (in fact, it is an enhancement to the CP algorithm that accounts for tasks being blocked by **use** operations as well as by **delay** operations). Thus in terms of the ultimate numeric result, the analysis algorithm performs better than the lower bound algorithm. While Eq. (3.14) yields $T^l = 3$ the numeric algorithm yields $T = 4$ at comparable cost (of course, T^l is a *symbolic* result with all the associated advantages).

While in many cases simulation provides a deterministic result, there are cases that involve "true" non-determinism, e.g., as a result of the generally non-deterministic outcome of a conflict arbitration. For example, consider the following model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r,1) \ ; \ \mathbf{delay}(2) \\ L_2 &= \mathbf{use}(r,2) \ ; \ \mathbf{delay}(1) \end{aligned}$$

Depending on the conflict arbitration it follows either $T = 4$ (L_1 scheduled prior to L_2), or $T = 5$ (L_2 scheduled prior to L_1). However, for many systems, the actual non-determinism is relatively small compared to the total amount of contention, especially for larger systems where the average influence of fine grain non-determinism is negligible compared to the overall (partly mutually exclusive) workload. Typical examples include models such as the macro data flow example where the variance in T is in the order of percents. Although non-deterministic, in such cases it is worthwhile to consider the outcome T of a single numeric pass instead of just using T^l, especially in an approximative application context. In fact, the extremely small variance of the simulation results in the macro data flow example has lead us to introduce an important optimization in the simulation model that yields significant speedup. In the model the granularity in terms of individual contentions is extremely high compared to the aggregate service demand. For instance, the unit of computational contention (service time) is the scheduling time slice ($\mathcal{O}(10^{-6})$ s) whereas each (macro) task corresponds to a service demand of $\mathcal{O}(1)$ s. The same applies to the communication grain size as one transfer involves $\mathcal{O}(10^3)$ individual packet transmissions. Consequently, the variance in T is negligible. As there is no point in deriving high-quality simulation results with practically no variance, based on a model of which the accuracy is inherently limited, the service time associated with the individual contentions is increased such that the number of contentions per macro task drops to $\mathcal{O}(10^2)$. While the variance in T increases to $\mathcal{O}(1)$ % (with no appreciable difference in terms of the mean value of T), the simulation time decreases by orders of magnitude. An important side effect of this optimization is that the simulation cost becomes independent of the total work load involved in the application.

Due to the fine granularity involved the macro data flow model serves as a good example to demonstrate the advantages of models based on PS-type resources rather than FCFS-type resources. While FCFS-type contention may introduce a large variance (cf.

difference between lower bound and upper bound on T in the 4-tasks example mentioned earlier), for PS-type contention the variance becomes zero. Again, consider

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r,1)\ ;\ \mathbf{delay}(2) \\ L_2 &= \mathbf{use}(r,2)\ ;\ \mathbf{delay}(1) \end{aligned}$$

If r is PS-type it follows $T = 4$ regardless of the non-deterministic conflict arbitration as shown in Fig. 5.4 (light shaded area denotes sharing, dark shared area denotes exclusive access). For applications like the macro data flow example this approach yields excellent

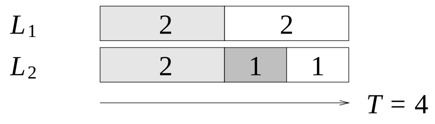


Figure 5.4: Trace of L for PS-type resource

results. First of all, due to the high granularity of computation and communication (as discussed above) modeling in terms of PS-type resources is appropriate. Furthermore, the cost of the above technique ("coarse grain PS") is much smaller than simulation based on "fine-grain FCFS". Thus, apart from the optimization discussed earlier, this technique offers an alternative optimization that yields a deterministic result at less computation time. The analysis algorithm simply involves updating the projected finish time for all current (PS) resource accesses for the event of either a new access or the event of finishing an ongoing access. The integration of the update algorithm in the critical path (simulation) algorithm is straightforward. A description in terms of the Pamela Run-Time Library call `pam_use()` has appeared in [109]. Table 5.4 shows a comparison between the result $T^{(1)}$ based on simulating the original, fine grain FCFS model (the unoptimized original), $T^{(2)}$ based on the first optimization in which the service times are increased, and $T^{(3)}$ based on simulating the coarse grain PS model. The actual measurements (T^m) have been included for reference. On average, the evaluation of $T^{(1)}$ takes $\mathcal{O}(10^3)$ s (on a Sun IPX workstation) while the standard deviation is given by $\sigma \approx 10^{-2}$. Both the approximations $T^{(2)}$ and $T^{(3)}$ are within a few percents of $T^{(1)}$. On average, the evaluation of $T^{(2)}$ takes $\mathcal{O}(10)$ s while $\sigma \approx 1$. On average, the evaluation of $T^{(3)}$ takes $\mathcal{O}(1)$ s while, of course, $\sigma = 0$. Consequently, PS-type models offer significant possibilities for alternative algorithms for the numerical estimation of T that are highly efficient compared to traditional approaches to simulation. Note, of course, that the limited accuracy of the simulation model itself is an important justification of the use of such an approximate technique.

5.4.3 Virtual Barrier

An interesting property of the analysis of PS-type models is that it can be expressed in terms of a symbolic technique. Assuming fair scheduling (as usual) the fact that multiple processes access to a PS-type resource implies that the processes are (infinitely) closely

G	T^m	$T^{(1)}$	$T^{(2)}$	$T^{(3)}$
G_1	118.7	116.8	114.7	115.8
G_2	93.9	95.4	92.5	93.9
G_3	95.6	93.4	92.8	93.4
G_4	94.1	89.2	87.4	88.7
G_5	73.4	72.2	70.9	71.4
G_6	105.8	104.9	103.9	104.7
G_7	98.4	89.3	87.0	87.7
G_8	89.2	86.6	87.1	86.6
G_9	87.6	87.0	84.4	87.0
G_{10}	109.5	108.4	106.4	107.6
G_{11}	141.2	139.4	138.4	139.0
G_{12}	149.5	147.5	144.8	145.5
G_{13}	165.9	164.3	163.2	163.7
G_{14}	172.3	171.7	171.0	171.3
G_{15}	188.6	187.2	186.8	187.8

Table 5.4: Three different simulation techniques compared (taken from [109])

synchronized. This additional knowledge can be exploited as can be seen in the 4-tasks example. Again, consider

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r, \tau_1) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \tau_3) \ ; \ \mathbf{delay}(\tau_4) \end{aligned}$$

However, the service demands are not known numerically. Currently, our only method to compile the model to a symbolic expression is by applying Eq. (3.14) that yields

$$T^l = \max(\tau_1 + \tau_2, \tau_3 + \tau_4, \tau_1 + \tau_3)$$

However, the fact that under the PS discipline the service bandwidth of r is exactly halved during the time L_1 and L_2 share access (i.e., $\min(\tau_1, \tau_3)$), we can transform L into the following model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r, \tau_s) \ ; \ \mathbf{use}(r, \max(\tau_1 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \tau_s) \ ; \ \mathbf{use}(r, \max(\tau_3 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_4) \end{aligned}$$

where both original **use** tasks are divided into a phase in which r is shared (duration $\tau_s = \min(\tau_1, \tau_3)$) and a phase in which r is owned exclusively. Due to the fact that both $\mathbf{use}(r, \tau_s)$ tasks finish at the same time we may write

$$\begin{aligned} L &= (\mathbf{use}(r, \tau_s) \parallel \mathbf{use}(r, \tau_s)) \ ; \ (L_1 \parallel L_2) \\ L_1 &= \mathbf{use}(r, \max(\tau_1 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \max(\tau_3 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_4) \end{aligned}$$

where this phenomenon is expressed by including the explicit barrier synchronization between both phases. Clearly the model reduces to

$$\begin{aligned} L &= \mathbf{use}(r, 2\tau_s) \ ; \ (L_1 \parallel L_2) \\ L_1 &= \mathbf{use}(r, \max(\tau_1 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_2) \\ L_2 &= \mathbf{use}(r, \max(\tau_3 - \tau_s, 0)) \ ; \ \mathbf{delay}(\tau_4) \end{aligned}$$

Although the model is equivalent, lower bound analysis now yields a tighter bound. For the numeric model instance mentioned earlier it now holds $T^l = T = 4$ whereas for the original model $T^l = 3$. The additional barrier is called a *virtual barrier* because it does not exist as a result of explicit synchronization in the model but as a result of the inter-process synchronization due to the intimate (fair) resource sharing between the threads. The above analysis technique is, of course, similar to the numeric algorithm mentioned earlier. In many practical cases the analysis technique used will be numeric rather than symbolic in order to avoid the delayed evaluation of the extra 'min' and 'max' terms that need to be generated in the course of the transformation. Nevertheless, the inclusion of virtual barriers in models where there exists significant resource sharing is a valuable extension of our symbolic analysis approach, especially with respect to its robustness in terms of worst case accuracy. This property will be further discussed in Chapter 6.

5.5 System Optimization

5.5.1 Introduction

In this section we will discuss how our performance modeling method can be applied to program optimization decisions such as the vectorization of computations or communications, or the choice between various data partitioning strategies. When the program involved is (sufficiently) static these optimization decisions can be made at compile-time. However, even when the parameters involved are not numerically known at compile-time, still a symbolic decision can be compiled given our low-cost, symbolic approach to performance modeling.

In order to illustrate our methodology we start with a simple example that shows how the optimization problem is addressed in the case of vectorization.

Example 5.2 Consider the following vector operation

```
forall i = 1 .. N
    x[i] = a * x[i];
```

The vectorization decision typically depends on whether N is large enough to sufficiently amortize the startup overhead of the vector operation. If we ignore memory traffic for simplicity, the operation can be modeled by

$$L = \mathbf{par}\ (i = 1, N)\ flop$$

where *flop* denotes the scalar multiplication. Execution on a scalar processor would be modeled by

$$flop = \mathbf{use}(s, \tau_f)$$

where τ_f denotes the instruction delay of the scalar floating point unit s. It follows

$$L_s = \textbf{par}\ (i = 1, N)\ \textbf{use}(s, \tau_f)$$

which, by Eq. (3.14), yields $T_s = N\tau_f$. On the other hand, execution on a vector processor would be modeled by using an S stage pipeline (cf. Example 3.5)

$$flop = \textbf{seq}\ (j = 1, S)\ \textbf{use}(u_j, \tau_c)$$

where u_j denotes stage j of the vector unit and τ_c denotes the cycle time. Note that in reality the startup time will be determined by more factors than just the pipeline hardware stages, e.g., call overhead, memory latency (cf. Example 4.5). However, the above model does account for the startup and bandwidth parameters as measured in practice simply by (re)defining the pipeline as a combined software/hardware pipeline such that S and τ_c satisfy (fit) the performance measurements. It follows

$$L_v = \textbf{par}\ (i = 1, N)\ \textbf{seq}\ (j = 1, S)\ \textbf{use}(u_j, \tau_c)$$

which, by Eq. (3.16), yields $T_v = (S + N - 1)\tau_c = \tau_s + N\tau_c$ where $\tau_s = (S - 1)\tau_c$ denotes the startup overhead. Consequently, the (boolean) vectorization decision v becomes

$$\begin{aligned}
&T_v < T_s \\
&\tau_s + N\tau_c < N\tau_f \\
&N \geq \left\lceil \frac{\tau_s}{\tau_f - \tau_c} \right\rceil \\
&N \geq N_v
\end{aligned}$$

□

The example shows two aspects. First, it demonstrates the *abstract* approach towards the two choices of mapping the (inherently) parallel algorithm onto the scalar or vector machine while merely by discussing alternative *flop* models using the same algorithmic description. (Note that the typically sequential implementation of the algorithm in the scalar case has not been modeled explicitly.) The advantages of the abstract modeling method will become clear later on. Second, it shows how the optimization problem is addressed in terms of applied PAMELA calculus. If N is known at compile-time, v can be evaluated. Even when N is only known symbolically the above expression can be compiled and evaluated at run-time. The compile-time decision then reduces to the question if it is worth while to generate the integer test $N \geq N_v$ considering the additional cost.

Formally, the above optimization problem can be expressed by the following 0-1 integer programming problem

$$T = [v]T_v + (1 - [v])T_s$$

where T is to be minimized, v is the boolean variable and T_v and T_s denote the performance models of the alternatives described earlier. As shown above, this simple optimization problem can be solved symbolically yielding $v = (N \geq N_v)$. In general, however, optimization will involve many decisions (e.g., various loop transformations, remapping) all of which are not independent, i.e., cannot be evaluated *locally*. Hence, the optimization problem needs to be expressed by an integrated performance model at *global* level in terms of which the dependency of the various optimizations can be expressed. An interesting example of the approach is presented by Kremer [88] in which the remapping decisions that can be taken between various phases of a program are expressed in terms of a 0-1 integer programming problem (note that each remapping decision affects remapping decisions between later program phases). In the following we will describe the principles involved in applying the PAMELA approach to program optimization. Based on our performance calculus, we will also refer to this specific application by the term "optimization calculus".

5.5.2 Optimization Calculus

In this section we will discuss the specific modeling approach that is most appropriate for the purpose of performance optimization. Note that performance modeling for obtaining (accurate) execution time predictions and modeling for the sole purpose of optimization do not necessarily imply the same approach. In many of the examples presented thus far performance prediction feedback has been based on modeling the code that is effectively generated by the compiler in the course of the translation process, i.e., usually in terms of optimized (partitioned) index loops and possibly explicit synchronizations (shared-memory systems, cf. matrix factorization in Section 5.2) or explicit message-passing instructions (distributed-memory systems, cf. matrix multiplication in Section 5.2). Although many optimization aspects are thus captured in the feedback loop the examples show that predictions are not always easily compiled, especially when the code generation paradigm is (i.e., has become) message-oriented, instead of the original procedure-oriented style in which the algorithm is typically expressed.

We will illustrate the problem with respect to analyzability (and subsequent optimizability) by a simple example. Consider a (simplified) line relaxation algorithm fragment [2] applied to an $N \times N$ matrix A according to

```
for i = 1 .. N-2
    forall j = 0 .. N-1
        A[i][j] = A[i-1][j] + A[i+1][j];
```

that constitutes the phase in which the relaxation sweep direction is in the i direction (typically followed by a sweep in the j direction but which is not considered now). In the parallelization for a P processor distributed-memory machine we consider the choice between two regular block partitioning strategies, i.e., either along the i axis or along the j axis (a choice, by the way, that is clearly trivial).

Of course, we could model the corresponding (SPMD) code in order to determine which alternative has the lowest execution time. For the j axis partitioning the code would be characterized by

```
for i = 1 .. N-2
    for j = L(p) .. U(p)
        A[i][j] = A[i-1][j] + A[i+1][j];
```

where p denotes the processor index and L and U denote the processor-specific index bounds ($U(p) - L(p) = \mathcal{O}(N/P)$). Although the above code clearly reveals the speedup gain, this does not apply to the alternative partitioning. For the i axis partitioning less straightforward code is generated (again, following the "owner-computes" convention) according to the following SPMD code (only shown for processors $p = 1, \ldots, P-2$), i.e.,

```
send(p-1,A[L(p)][:]);
recv(p-1,tempvec_l);               ! recv A[L(p)-1][:]
recv(p+1,tempvec_u);               ! recv A[U(p)+1][:]
for i = L(p) .. U(p) {
    if i = L(p)
        for j = 0 .. N-1
            A[i][j] = tempvec_l[j] + A[i+1][j];
    if i > L(p) and i < U(p)
        for j = 0 .. N-1
            A[i][j] = A[i-1][j] + A[i+1][j];
    if i = U(p)
        for j = 0 .. N-1
            A[i][j] = A[i-1][j] + tempvec_u[j];
}
send(p+1,A[U(p)][:]);
```

Like in earlier examples, the code is expressed in terms of the original (shared) data structure for simplicity. Note that the j loops are vectorizable.

While the first code can be easily compiled into a symbolic performance model, the second code illustrates the potential difficulties involved when compiling generated SPMD code into a performance model. Especially in the case of the above code, symbolic compilation is typically impossible as discussed earlier. Although the local bounds on the i loop are reduced by a factor P compared to the original algorithm, the message-passing scheme still serializes the entire computation. As discussed in Section 3.3 the "thread of condition synchronization" that determines the critical path now runs through each process that makes it hard to detect this in terms of a symbolic expression. However, this critical path is nothing but the result of the explicit sequential i loop at algorithm level. Consequently, it is much more advantageous to consider a modeling approach at *algorithm* level (i.e., procedure-oriented level) rather than at *implementation* level (i.e., message-oriented level).

In order to abstract from the actual partitioning implementation (either for shared-memory or distributed-memory systems) we will model the original computation with its full (potential) parallelism while each mapping decision is expressed in terms of a contention model. In fact, this is the same material-oriented "contention modeling" approach as in the vectorization example where the potential parallelism of the vector operation is expressed while the machine resources determine the actual parallelism. The performance model of the relaxation algorithm is expressed according to

$$\textbf{seq}\ (i = 1, N-2)\ \textbf{par}\ (j = 0, N-1)\ flop(i,j)$$

where $flop(i,j)$ denotes the update of element A_{ij} (ignoring data transfers for the moment). Let the mapping function $\mu(i,j)$ denote the processor resource responsible for the update of A_{ij}. Then the machine model is given by

$$flop(i,j) = \mathbf{use}(cpu_{\mu(i,j)}, \tau_f)$$

where τ_f denotes the computation time associated with the update of A_{ij}.

For the j axis partitioning it holds $\mu(i,j) = j/B$ where $B = N/P$ denotes the block size (for simplicity assume $P|N$). It follows

$$\mathbf{seq}\ (i = 1, N-2)\ \mathbf{par}\ (j = 0, N-1)\ \mathbf{use}(cpu_{j/B}, \tau_f)$$

For each parallel section (instance i) Eq. (3.14) yields ($m = p$ denotes the resource index)

$$\varphi = \max_{j=0\ldots N-1} \tau_f, \quad \omega = \max_{m=0\ldots P-1} \sum_{j=0}^{N-1} [j/B = m]\tau_f$$

Note that in contention models ω is typically much larger than φ. Consequently, Eq. (3.15) yields

$$T^l = \sum_{i=1}^{N-2} \max_{m=0\ldots P-1} \sum_{j=0}^{N-1} [j/B = m]\tau_f = (N-2)\frac{N}{P}\tau_f$$

corresponding to the speedup found earlier. For the i axis partitioning it holds $\mu(i,j) = i/B$. It follows

$$\mathbf{seq}\ (i = 1, N-2)\ \mathbf{par}\ (j = 0, N-1)\ \mathbf{use}(cpu_{i/B}, \tau_f)$$

which directly reveals the algorithm's sequential nature. Indeed, by Eq. (3.15) (and all the equations called within) it follows

$$T^l = \sum_{i=1}^{N-2} \max_{m=0\ldots P-1} \sum_{j=0}^{N-1} [i/B = m]\tau_f = (N-2)N\tau_f$$

Thus by exploiting the knowledge of the algorithm's inherent sequentialism as still present in the algorithm description, from simple serialization analysis it directly follows that an i axis partitioning will not yield any speedup[8]. In contrast to the SPMD implementation, the inherent sequentialism is still easily detectable. Note that the entire analysis is symbolic, whereas a comparison of predictions at implementation level would practically involve simulation (the relation between analytical models and simulation models is discussed in Section 5.4). Although many optimization decisions will have to be evaluated numerically (at run time), this implies that the expressions that need to be evaluated are highly optimized themselves, possibly to the extent that it becomes feasible to compile the optimization decision as a part of the run-time code (cf. Example 5.2).

[8]This property only holds for the current algorithm. Later on, we will consider a modified version of the algorithm in which i axis partitioning does yield speedup.

5.5.3 Line Relaxation

The choice of material-oriented modeling at the algorithmic level for the purpose of optimization makes it easy to reason about much more optimizations than just the choice of partitioning and/or vectorization. As an example we will discuss the line relaxation algorithm in somewhat more detail. We will consider a two-phase algorithm corresponding to the example discussed in [2]. In the first phase the line relaxation is swept along the i axis, after which the relaxation is swept along the j axis. The algorithm is given by

```
for i = 1 .. N-2                                ! sweep vertical
    forall j = 0 .. N-1
        A[i][j] = A[i-1][j] + A[i+1][j];
for j = 1 .. N-2                                ! sweep horizontal
    forall i = 0 .. N-1
        A[i][j] = A[i][j-1] + A[i][j+1];
```

Let L_v and L_h denote the PAMELA models of the vertical and horizontal phase, respectively. Let $\mu(i,j) = j/B$ denote the initial data layout. Let r denote the decision to remap after the vertical phase, and let $\mu'(i,j) = i/B$ denote the resulting map. Let L_r denote the PAMELA model of the remapping. In terms of PAMELA optimal system performance is given by

$$L_v(\mu) \;;\; \textbf{if}\ (r)\ L_r \;;\; \textbf{if}\ (r)\ L_h(\mu')\ \textbf{else}\ L_h(\mu)$$

Given optimal designs for $L_v(\mu)$, $L_h(\mu)$, and $L_h(\mu')$, the mapping decision follows from the solution of the minimization problem

$$T_v(\mu) + [r](T_r + T_h(\mu')) + [\neg r]T_h(\mu)$$

where T_x denotes the lower bound time estimate of L_x, $x \in \{v, h, r\}$. In the following we will investigate the various lower bound estimates. In order to avoid unnecessary complicated expressions we will use order terms only.

We start with L_v. In terms of PAMELA the vertical sweep phase is given by

$$\begin{aligned} L_v = \textbf{seq}\ (i = 1, N-2) & \\ \textbf{par}\ (j = 0, N-1)\ \{ & \\ & move(\mu(i-1,j), \mu(i,j)); \\ & move(\mu(i+1,j), \mu(i,j)); \\ & flop(\mu(i,j)) \\ \} & \end{aligned}$$

in which we now explicitly account for data accesses that (especially for distributed-memory machines) may involve data movement between processors (according to the "owner-computes" model). We will only account for non-local data transfers in which we assume that any associated condition synchronization (e.g., due to some message-passing implementation) is negligible (as in previous case studies). Let

$$flop(p) = \textbf{use}(cpu_p, \tau_f)$$

as before, and let

$$move(p, q) = \mathbf{use}(l_p, [p \neq q]\tau_m) \parallel \mathbf{use}(l_q, [p \neq q]\tau_m)$$

which accounts for both work load at sender and receiver. Note, that this is just an abstract model. However, it does account for the fact that a node can neither send nor receive data in parallel without (proportionally) increasing the associated time complexity (cf. the macro data flow example). Since $\mu(i, j) = j/B$ by straight-forward substitution it follows

```
$L_v$ = seq $(i = 1, N-2)$
        par $(j = 0, N-1)$ {
            $\{\mathbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m) \parallel \mathbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\}$;
            $\{\mathbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m) \parallel \mathbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\}$;
            $\mathbf{use}(cpu_{j/B}, \tau_f)$
        }
```

which yields

$$T_v = \sum_{i=1}^{N-2} \max_{m=0\ldots P-1} \sum_{j=0}^{N-1} [j/B = m]\tau_f = \mathcal{O}(\frac{N^2}{P})\tau_f$$

The horizontal phase is given by

```
$L_h$ = seq $(j = 1, N-2)$
        par $(i = 0, N-1)$ {
            $move(\mu(i, j-1), \mu(i, j))$;
            $move(\mu(i, j+1), \mu(i, j))$;
            $flop(\mu(i, j))$
        }
```

which equals

```
$L_h$ = seq $(j = 1, N-2)$
        par $(i = 0, N-1)$ {
            { $\mathbf{use}(l_{(j-1)/B}, [(j-1)/B \neq j/B]\tau_m) \parallel$
              $\mathbf{use}(l_{j/B}, [(j-1)/B \neq j/B]\tau_m)$
            } ;
            { $\mathbf{use}(l_{(j+1)/B}, [(j+1)/B \neq j/B]\tau_m) \parallel$
              $\mathbf{use}(l_{j/B}, [(j+1)/B \neq j/B]\tau_m)$
            } ;
            $\mathbf{use}(cpu_{j/B}, \tau_f)$
        }
```

which yields (after a few reductions)

$$T_h = \mathcal{O}(PN)\tau_m + \mathcal{O}(N^2)\tau_f$$

Clearly, it is interesting to evaluate the remapping cost. Remapping A implies a transposition according to

$$L_r = \textbf{par}\ (i = 0, N-1)\ \textbf{par}\ (j = 0, N-1)\ move(\mu(i,j), \mu(j,i))$$

Given the mapping $\mu(i,j) = j/B$ it follows

$$\begin{aligned} L_r = \textbf{par}\ & (i = 0, N-1) \\ & \textbf{par}\ (j = 0, N-1) \\ & \quad \{\textbf{use}(l_{j/B}, [j/B \neq i/B]\tau_m) \parallel \textbf{use}(l_{i/B}, [j/B \neq i/B]\tau_m)\} \end{aligned}$$

which yields

$$\begin{aligned} T_r &= \max_{m=0 \ldots P-1} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} ([j/B = m][j/B \neq i/B] + [i/B = m][j/B \neq i/B])\tau_m \\ &= \mathcal{O}(\frac{N^2}{P})\tau_m \end{aligned}$$

As $T_h(\mu') = T_v(\mu)$, the combined performance model becomes

$$\mathcal{O}(\frac{N^2}{P})\tau_f + [r](\mathcal{O}(\frac{N^2}{P})\tau_m + \mathcal{O}(\frac{N^2}{P})\tau_f) + [\neg r](\mathcal{O}(PN)\tau_m + \mathcal{O}(N^2)\tau_f)$$

The solution for r is given by

$$r = \mathcal{O}(\frac{N^2}{P})\tau_m + \mathcal{O}(\frac{N^2}{P})\tau_f < \mathcal{O}(PN)\tau_m + \mathcal{O}(N^2)\tau_f$$

Clearly there exists a value for P for which remapping is justified (depending on τ_f and τ_m).

Thus far, we have considered remapping while assuming that the same algorithm would have to be used for the horizontal phase. However, for the horizontal phase the algorithm can be optimized that puts the whole issue of remapping in a different perspective. The optimization we consider is based on pipelining the computation across the processors which is explained at length in [2]. Recall the original algorithm for the horizontal phase, i.e.,

```
for j = 1 .. N-2
    forall i = 0 .. N-1
        A[i][j] = A[i][j-1] + A[i][j+1];
```

that corresponds to

$$\textbf{seq}\ (j = 1, N-2)\ \textbf{par}\ (i = 0, N-1)\ \textbf{use}(cpu_{j/B}, \tau_f)$$

where we ignore moves for simplicity. By reversing the i and j loop the algorithm can be expressed as

```
forall i = 0 .. N-1
    for j = 1 .. N-2
        A[i][j] = A[i][j-1] + A[i][j+1];
```

Instead of running sequential, each i loop is pipelined such that the next processor executes a different i loop instance concurrently, yet obeying the j sequence [2]. Thus we assume a schedule such that the next i loop is executed when the previous j loop has traversed exactly one processor (a block of B indices). Note, that the above algorithm does not include this additional scheduling constraint which is hard to express without resorting to an explicit (and complicated) synchronization scheme if expressed in a conventional paradigm (see Section 3.3). However, the pipelining behavior is easily captured in terms of a PAMELA model due to our contention modeling approach. The following PAMELA model essentially expresses the intended pipelining optimization, i.e.,

$$L_{h'} = \textbf{par}\ (i = 0, N-1)\ \textbf{seq}\ (p = 0, P-1)\ \textbf{use}(cpu_p, B\tau_f)$$

The j loop is stripmined in terms of a p loop such that all B individual j accesses local to processor p are expressed by the same **use** statement (i.e., the lower level j loop local to p is reduced to a single **use** statement). The reason for this modification is that this model does express the intended schedule whereas the model

$$\textbf{par}\ (i = 0, N-1)\ \textbf{seq}\ (j = 1, N-2)\ \textbf{use}(cpu_{j/B}, \tau_f)$$

would not represent the pipelining behavior at all because of the virtual barrier effect[9]. The loop reversal according to $L_{h'}$ potentially has a great effect on performance as it holds

$$T_{h'} = \max_{m=0\ldots P-1} \sum_{i=0}^{N-1} \sum_{p=0}^{P-1} [p = m] B\tau_f = \mathcal{O}(\frac{N^2}{P})$$

Thus, by loop reversal, the same (order) of performance can be achieved as in the first phase, without remapping the data. Of course, $L_{h'}$ ignores the additional pipeline startup delay as well the additional communication overhead as the need for communication is still present. Especially when multiple sweeps are performed in both phases remapping can still be appropriate. Let s denote the pipelining optimization ($s$wap loops). In general, the optimization problem is given by

$$T_v(\mu) + [r](T_r + ([\neg s]T_h(\mu') + [s]T_{h'}(\mu')) + [\neg r]([\neg s]T_h(\mu) + [s]T_{h'}(\mu))$$

which expresses the mutual dependency between the remapping decision r and the pipelining decision s. Clearly, for single sweeps in the vertical and horizontal phase the solution is given by $(r, s) = (0, 1)$. Note, that in this particular problem it holds $r = \neg s$.

The discussion of the line relaxation example illustrates the advantage of the material-oriented modeling approach when applied at algorithmic level in order to express compile-time (i.e., symbolic) optimizations. The symbolic analysis permits the optimizations to be expressed in terms of an overall symbolic expression that is subject to minimization, e.g., by 0-1 integer programming. In many cases (like in the above examples) low-cost solutions can be derived which implies that the optimization decisions can be compiled and evaluated at run-time.

[9]Because of the general assumption of fair scheduling the (i, j) accesses would still be selected according to a column-major scheme rather than the intended row-major scheme. All parallel i loops experience mutual synchronization that causes P virtual barriers. Thus, the model would effectively behave the same as the original (without loop reversal).

Of course, there are cases where the choice to abstract from the actual code may induce a considerable error as the code generation model (i.e., compiler) is not included in the prediction. This applies in particular to the generation of message-passing code for distributed-memory machines. For instance, a naive message-passing code generation model can completely sequentialize an inherently parallel operation[10]. Clearly, the above prediction method will not take this into account (note that inherent sequentializations at algorithmic level are accounted for, though). However, a code generation model may be assumed that does not introduce pathological schedules (a simple inspector/executor model already solves the problem) which implies that the algorithmic level abstraction will not introduce an error larger than the inherent error of the lower bound technique.

Of course, the lower bound technique itself introduces a certain inaccuracy with respect to which optimization will actually yield better performance. On the other hand, the limitation on the inaccuracy (discussed at length in the next chapter) guarantees that if a wrong optimization is applied, the performance decrease is also limited. This is acceptable if one realizes that compile-time optimization must cover a large search space where (first-order) comparisons must be made in extremely short time rather than extremely accurately.

5.6 Summary

In this chapter we have presented an elaborate discussion on various aspects of the Pamela methodology, involving applications of the analytic technique, modeling an actual application, a discussion of alternative analytic techniques, and applications of our methodology in the field of system optimization.

In the first case study, we have discussed the application of our analytic prediction technique to a matrix factorization algorithm on a shared-memory machine and a matrix-vector multiplication on a distributed-memory machine, thus involving somewhat more complex models than the small examples discussed before. Rather than demonstrating the modeling technique itself we have concentrated on the mechanics of compiling the Pamela models into analytical performance models. It is shown that the greatest challenge in this process is not the transformation into the time domain model itself but rather the subsequent reduction into low-cost versions without (additional) loss of accuracy.

In the second case study, we have presented a performance model of macro data flow applications running on a distributed-memory machine that is compared with the measured execution times. Through this case study we have demonstrated the Pamela approach towards modeling real applications as well as the high accuracy of the Pamela models that is achieved in practice. As in the previous case study the analytical predictions T^l are typically within 50 % of T whereas conventional static techniques entail severe errors. Once again, the case study demonstrates the necessity of integrating contention analysis within analytical techniques. Especially now that multi-threaded computation and communication are becoming common place there is an increasing need for machine

[10]For example, consider a simple computation $y_i = f(x_{i-1})$ where each element of x and y are mapped onto a separate processor (x and y aligned). A naive, scalar "owner-computes" scheme in which each processor (sequentially) traverses the entire index space in the positive i direction (cf. Example 5.1) will completely sequentialize the computation.

contention models, useful in an analytic context, that account for the limited number of resources actually available. Our modeling approach delivers the accuracy needed where conventional static models fail.

In the third case study we have explored alternative solution techniques for Pamela models based on a numeric process rather than a symbolic process. While simulation in the sense as introduced in earlier chapters only serves as the default technique, especially when Pamela models are relatively deterministic this numeric approach provides a simple, solution technique that outperforms T^l at comparable cost. The fact that for many (fine grain) models the variance in T is very small offers the opportunity of significant model reduction, yet retaining accuracy. Furthermore, it has been shown that approximations in terms of PS-type resource models also yield considerable speedup. Another advantage of PS-type models is the applicability of a symbolic preprocessing phase that improves the robustness of the lower bound technique by the detection of so-called virtual barriers in the model.

In the last case study we have demonstrated the use of the Pamela methodology in system optimization, which, as mentioned in Chapter 1, is an essential motivation for our work. We have shown that, in order to enable optimization in an analytical, low-cost framework, the system must be modeled at algorithmic level rather than at implementation level. This requirement especially applies to message-passing implementations because of the loss of analyzability associated with the transformation from the procedure-oriented to the message-oriented paradigm as also discussed in Chapter 3. Apart from increased analyzability another reason for the abstraction is the fundamental problems associated with the performance modeling of the compiler-specific code generation as well as the unpredictable influence of the run-time environment. Although both aspects may introduce considerable variance, they are of no consequence as long as performances are only *compared*. We have illustrated the use of our optimization calculus by considering optimizations such as vectorization, choosing data mappings, as well as other algorithmic transformations. It is shown, that our symbolic approach delivers extremely low cost as well as sufficient discriminatory power needed to serve as an appropriate tool for system optimization.

Chapter 6

Accuracy

6.1 Introduction

In some of the previous examples such as the MRM, pipeline, matrix-vector multiplication, and the macro data flow computation, it has been shown that the difference between T^l and T can be as much as 50%. In general, the question that arises with the introduction of the approximate lower bound technique is how tight T^l is compared to T. In this chapter we will investigate some of the properties of T^l in relation to T. We will show that the prediction error of T^l is limited *and* predictable. These properties of T^l are an essential justification of our approximate, analytical approach to performance modeling. Parts of this work have been presented in [54, 55].

As mentioned earlier, in general the execution time of a PAMELA model is stochastic with a finite distribution between a lower bound T^l and an upper bound T^u. For models with coarse grain contention the likelihood of T being close to T^l or T^u is relatively high. Hence, for these models the *absolute* range of T relative to T^l is an important measure with respect to the accuracy of T^l. For fine grain models, on the other hand, the variance of T is much more limited. As the typical probability of T being near to T^u is extremely small (as we will show), the difference between the *mean* value of T relative to T^l is a much more appropriate measure to characterize the accuracy of T^l.

In this chapter we will investigate the accuracy of T^l for both coarse grain and fine grain models. First, we investigate the properties of T^u which, in turn, yields information on the maximum absolute deviation between T^l and T that may be anticipated. Theory and experiments suggest that for a large class of systems the maximum deviation between T and T^l is limited by a constant. Next, we investigate the properties of the mean value of T through a number of simulation experiments which show that the *average* deviation between T^l and T for large systems is quite limited (a factor 2 in the worst case).

It serves to note that earlier results on the bounds of T by Graham [58], and later by Sarkar [137], and Eager, Zahorjan, and Lazowska [39] do *not* apply to our problem. Their results[1] apply to parallel systems where resources are allocated according to a *work conserving* scheduling discipline [85]. This implies that there is *no* unforced idleness, very much *unlike* the *statically* mapped systems we consider (we assume no resource multiplicity) where many resources need not be used although tasks are ready.

[1] It has been shown that $T^u < 2T^l$ for any schedule as long as there is no unforced idleness.

Hence, our work is relatively new. While the results on the average deviation of T^l can be interpreted in terms of results known from queuing theory (and, in a sense, in terms of the scheduling results just mentioned above), this particularly applies to the research into the absolute accuracy of T^l. As the analysis into the properties of T^u is quite complicated (especially for the derivation of a *tight* bound, in contrast to T^l), we only present a number of conjectures and associated experiments. The motivations for including this introductory work are threefold. First, for coarse grain models the probability for T being close to T^u are simply quite real (as will be shown). A second motivation is that knowledge on T^u, rather than T^l, is a requirement when time constraints need to be verified (e.g., in real-time designs). A third motivation is that this introduction may inspire more work to be performed in what appears to be an interesting field.

6.2 Absolute Accuracy

6.2.1 Introduction

While the influence of fine grain contention (at the subtask-level) can be accounted for in terms of probabilistic models (next section), for cases with coarse grain (task-level) contention the likelihood of T reaching values in the neighborhood of the upper bound T^u can not be ignored. Examples are statically (and poorly) scheduled/programmed systems that may exhibit quite disappointing execution time results when compared with predictions based on an average or a lower bound. An example of this phenomenon has been shown in the matrix-vector case study in which a simple (communication) loop reversal reduced communication time by as much as a factor 2 due to the elimination of link contention. Similar observations are reported by Culler *et al.* [31]. As an example of the effect of non-determinism at the task level due to contention, consider the coarse-grain model

$$\begin{aligned} L &= L_1 \parallel L_2 \\ L_1 &= \mathbf{use}(r_1, 1);\ \mathbf{use}(r_2, 99) \\ L_2 &= \mathbf{use}(r_1, 99);\ \mathbf{use}(r_2, 1) \end{aligned}$$

where $r_1 = r_2 = 1$. As $\varphi = \omega = 100$ it follows $T^l = 100$. This prediction is accurate as long as L_1 takes priority over L_2 with respect to r_1. However, if the inverse were true, T would nearly double ($T^u = 199$). In this section we investigate T^u in order to assess the absolute deviation between T and T^l.

Because of the exponential complexity involved with deriving T^u (i.e., identifying the worst schedule possible) for any model, in the following we will only consider a *parallel section* L that cannot be further decomposed into a straight sequence of subsections (i.e., Eq. (3.14) applies). Furthermore, we only consider a section *without* any precedence relations between the P parallel processes within the section. Thus L can be written as

$$L = \mathbf{par}\ (i = 1 \ldots P)\ L_p$$

We also assume that no knowledge is available (or practically useful at compile-time) on the precise structure of each parallel process L_p other than that it involves accesses to maximally M resources $r_1, \ldots, r_M$ ($r_j = 1,\ j = 1, \ldots, M$). Let $\Delta = (\delta_{ij})$ denote the $P \times M$ *demand matrix* where δ_{ij} denotes the service demand of process i on resource j. First, we study the behavior of parallel sections in which each process accesses each resource exactly once. Consequently, if we define the access permutation in terms of the $P \times M$ *schedule* matrix S, the parallel section can be expressed by

$$L = \mathbf{par}\ (i = 1 \ldots P)\ \mathbf{seq}\ (j = 1 \ldots M)\ \mathbf{use}(r_{s_{i,j}}, \delta_{i,s_{i,j}})$$

Clearly, there are schedules that realize $T = T^l$, and there are schedules that realize $T = T^u$. In order to characterize the range between T^l and T^u, we introduce the *ratio* $\eta = T^u/T^l$ for which we derive a number of properties. As in general a sharp upper bound cannot be computed, we consider a number of specific cases that allow for computational tractability as well as provides an interpretation for practical situations.

In general, the analysis on the ratio of T^l and T^u can be stated as a problem, solely characterized by the demand matrix (including its dimension) or, more practically, by a tuple (P, M), where P denotes the number of concurrent processes, and M denotes the *maximum* number of resources involved in any of the processes. Thus, the above example is denoted the $(2,2)$ problem for which it will be proven that

$$\eta_{(2,2)} < 2$$

The general (P, M) problem, given Δ, can be solved by finding the permutations S^l and S^u that minimizes T^l and maximizes T^u, yielding the upper bound on $\eta_{(P,M)}$.

Before discussing the (P, M) problem, we revisit the $(2,2)$ problem, for which we derive a lower and upper bound as a function of Δ and subsequently prove that $\eta_{(2,2)} < 2$. We will present two proof versions. In the first proof we derive the expressions for T^l and T^u for *arbitrary* Δ, from which we infer the conditions for which $\eta \to 2$. In the second proof we simply introduce the *optimum* value of Δ and the related schedules, which we show to be optimal. The second proof style will be used frequently in the sequel.

First Proof

Consider a $(2,2)$ problem, with a given resource demand matrix Δ. Clearly,

$$S^l = \begin{pmatrix} k & l \\ l & k \end{pmatrix}$$

will produce a lower bound schedule, where k, l are given by

$$k \in \{1,2\},\ l = \begin{cases} 1, & k = 2; \\ 2, & k = 1. \end{cases}$$

The result for $k = 1$ is represented by Fig. 6.1, which illustrates that, in general, the Δ elements need not be equal. As can be seen from the trace, it immediately follows

$$T^l = \max(\delta_{1k}, \delta_{2l}) + \max(\delta_{1l}, \delta_{2k})$$

that implies that T^l is independent of k. On the other hand,

$$S^u = \begin{pmatrix} k & l \\ k & l \end{pmatrix}$$

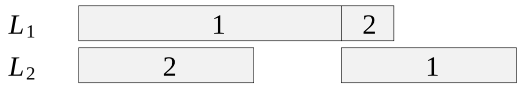


Figure 6.1: Lower bound schedule trace

will produce an upper bound schedule. Let

$$m \in \{1,2\},\ n = \begin{cases} 1, & m = 2; \\ 2, & m = 1. \end{cases}$$

The upper bound schedule is determined by values of m, n where the *overlap* of using resources r_n (by L_1) and r_m (by L_2) is *minimized* (m encodes the order by which the initial conflict is resolved). The result for $k = 1$ is illustrated by Fig. 6.2, corresponding

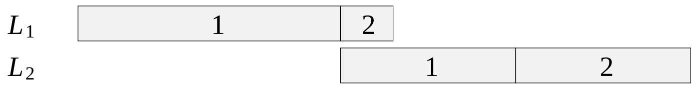


Figure 6.2: Upper bound schedule trace

to the assumption that $\min(\delta_{12}, \delta_{21}) \leq \min(\delta_{11}, \delta_{22})$ (i.e., $m = 1$). It is easily seen, that the choice for m is independent of k. Hence, without loss of generality, let

$$S^u = \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$$

Let m be chosen such that $\min(\delta_{1n}, \delta_{2m})$ is minimal. Then

$$T^u = \delta_{1m} + \max(\delta_{1n}, \delta_{2m}) + \delta_{2n}$$

which, given the choice of m, is maximal. We now derive an solution for η, based on the analysis of T^u. Since T^l is independent of k, in summary, we have

$$\begin{aligned} T^u &= \delta_{1m} + \delta_{2n} + \max(\delta_{1n}, \delta_{2m}) \\ T^l &= \max(\delta_{1m}, \delta_{2n}) + \max(\delta_{1n}, \delta_{2m}) \end{aligned}$$

Hence, η has the form

$$\eta_{(2,2)} = \frac{a + c + b}{\max(a, c) + b} < 2$$

reaching its upper bound when $a = c$ and $b \to 0$, the latter implying that the condition for m is indeed necessary.

Second Proof

While in the above proof expressions were derived for T^l and T^u as a function of arbitrary demand matrices, in the second proof we directly derive the bound on $\eta_{(2,2)}$. As suggested by the above result, the demand matrix, for which the maximum value of η is obtained, is given by

$$\Delta = \lim_{b \to 0} \Delta_b, \ \Delta_b = \begin{pmatrix} a & b \\ b & a \end{pmatrix}, \ a > 0$$

For a more practical value of a, the lower bound trace is given in Fig. 6.3, while the

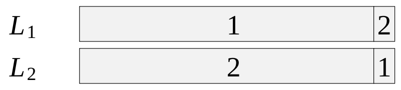


Figure 6.3: Lower bound schedule trace

corresponding upper bound trace is given in Fig. 6.4. From the traces it can be seen that

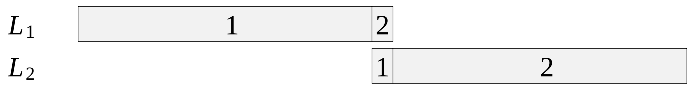


Figure 6.4: Upper bound schedule trace

Δ indeed produces the extreme schedules since, in the limit, the lower bound schedule exhibits 0 idleness (i.e., 100 % utilization), and the upper bound schedule exhibits 0 overlap (i.e., 100 % serialization). Hence, $\eta_{(2,2)} < 2$.

6.2.2 General Bound

In this section we derive a general bound on η when no information is available other than P and M.

Theorem 6.1 *Let* $L =$ **par** $(i = 1 \ldots P)$ L_i *denote a parallel section involving* M *resources. Then*

$$\eta < \min(P, M)$$

Proof: We prove the theorem by presenting the solution for Δ, and the lower and upper bound schedules that generate this bound on η. Subsequently, we show that the solution is optimal. First, we consider the case $M = P$. The optimal, *diagonal-dominant* demand matrix is given by

$$\Delta = \lim_{b \to 0} \Delta_b, \ \Delta_b = \begin{pmatrix} a & b & \ldots & b & b \\ b & a & \ldots & b & b \\ \vdots & \vdots & & \vdots & \vdots \\ b & b & \ldots & a & b \\ b & b & \ldots & b & a \end{pmatrix}, \ a > 0$$

Figure 6.5 shows both lower bound and upper bound schedules for $(P, M) = (3, 3)$. Each shaded box denotes the resource index involved in the access. It is easily seen that these

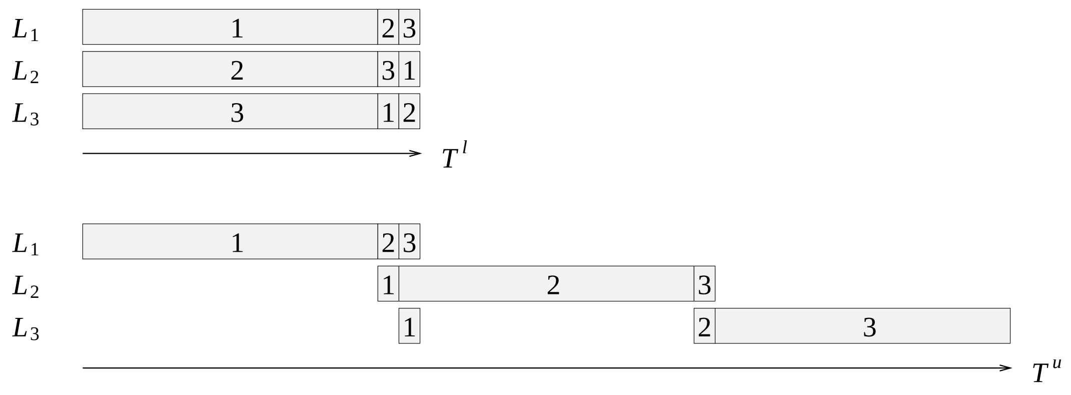


Figure 6.5: Lower (S^l) and upper bound (S^u) schedules for $(P, M) = (3, 3)$

schedules are optimal since S^l minimizes process idle time, while S^u minimizes process overlap when $b \rightarrow 0$. Thus, η is maximal. As, for $b \rightarrow 0$, $T^u = Pa$ and $T^l = \varphi = \omega = a$, it holds $\eta \leq P$. Next, we show that in order to attain this maximum ratio, Δ must indeed be "balanced", which in this sense means that

$$\forall i : \sum_{j=1}^{M} \delta_{i,j} = \varphi, \qquad \forall j : \sum_{i=1}^{P} \delta_{i,j} = \omega$$

Thus both the row sums and column sums must be equal which, for $M = P$, implies $\varphi = \omega$. We show the optimality as follows. If any of the service demands $\delta_{i,j}$ were to be increased with an amount ϵ, under the balanced condition T^l will also increase by ϵ. Since, T^u cannot increase more than ϵ, η will decrease. Next, we consider the case $M > P$. From the above argument, it is easily seen that adding resources will not improve η but rather decrease the ratio unless the associated demands are equal to b. Hence, the above bound also holds for $M > P$. Finally, we consider the case $M < P$. While it can be shown, that an upper bound schedule is still possible in which process overlap approaches zero, i.e., $T^u = Pa$, the lower bound now increases as access conflicts between processes can no longer be avoided. As $\omega = \lceil \frac{P}{M} \rceil a$, it follows $\eta < M$. Hence, it holds $\eta < \min(P, M)$ which concludes the proof. $\square$

The above theorem states the intuitive fact that, without additional knowledge on the parallel section, its execution may range from fully sequential to fully parallel. In the latter case, the actual parallelism is either limited by the number of resources or by the number of processes. Again, note that the result of Theorem 6.1 illustrates the difference between statically mapped systems and work conserving systems where it would hold $\eta < 2$, regardless of P and M. When the number of accesses per resource is higher than one, the theorem still holds, applying to the case when these multiple accesses would be concatenated to a single, non-preempted access (note that this, of course, is highly improbable for larger systems which is essentially the reason for the "mean value" approach discussed in Section 6.3).

Clearly, the above result does not tighten the upper bound on the range of possible prediction outcomes. In the next section, however, we will analyze the situation for more practical values of Δ.

6.2.3 Adding Knowledge

In the previous section it was shown, that, without any knowledge on Δ (and S) there is no guarantee that the execution time is bounded, else than between the default range, i.e., from full parallelism to full sequentialism. In this section, we will study a number of more practical cases that involves additional knowledge on the nature of Δ.

Many practical situations are far from being represented by demand matrices that exhibit such a large number of *near*-zero entries. In the following we will assume that the ratio between a and b is *limited* by a constant γ. Using the construction method of the upper bound schedule discussed in the above theorem, we conjecture on the bound on η for this case, in which we also assume $M = P$ since this has been shown to provide the maximum value for η ($M > P$ is not interesting).

Conjecture 6.1 *Let* $L = \textbf{par}\ (i = 1 \ldots P)\ L_i$ *denote a parallel section involving* P *resources. Let* Δ *be given as in the above theorem with the additional constraint that the ratio between* a *and* b *is limited according to* $\gamma = a/b$. *Then for* $\gamma \geq 2$ *it is conjectured that*

$$\eta_{(P,P)} \leq \frac{P(\gamma + 2) - 3}{\gamma + P - 1} \tag{6.1}$$

Argument: While $T^l = a + (P-1)b = (\gamma + P - 1)b$, we conjecture that an upper bound schedule is constructed as illustrated in Fig. 6.6 for $P = 8$. Consequently $T^u = Pa + (P-2)2b + b = (P(\gamma+2) - 3)b$, that leads to the conjectured result for η. For small

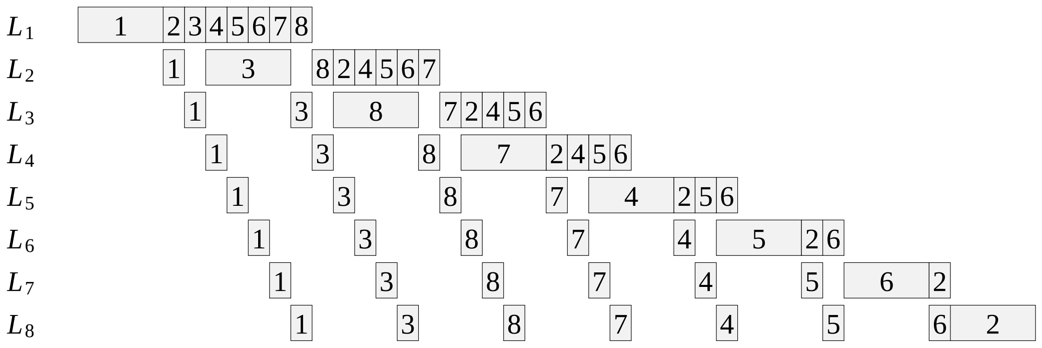


Figure 6.6: Upper bound schedule for $P = 8$ (conjectured)

P the above bound on η has indeed been verified for various values (integer and real) of γ by conducting simulation experiments in which all possible schedules are enumerated. Due to the exponential complexity of the exhaustive search, however, this computational proof has only been obtained for $P \leq 4$. A partial search for $P = 5$ has shown exactly the same phenomenon. For $\gamma < 2$ a slight increase of T^u (b) has been observed. Hence, the conjecture applies to $\gamma \geq 2$. Note that much more schedules apply than just the particular one illustrated in Fig. 6.6, as shown by the experiments. However, in all cases the value of T^u proves to be the same. □

Essentially, the above conjecture suggests that for large P, η is limited by γ. For large values of P and γ it holds $\eta = \min(P, \gamma)$ (for $\gamma \to \infty$ the conjecture corresponds to Theorem 6.1). Note that although the above conjecture applies to $M = P$, the general bound given by Theorem 6.1 still holds. For example, consider a parallel section with $M = P$. Let P be large and $\gamma = 2$. According to the conjecture, $\eta \leq 4$. Now consider another parallel section with equal T^l in which γ is increased to $P - 1$ such that $M = 2$. From Theorem 6.1 it follows $\eta < 2$. Thus, η *decreases*, contrary to what the conjecture might suggest.

Although we have no rigid proof based on the construction of the schedule, an alternative schedule shown in Fig. 6.7 constructed through a strategy that continuously aims to maximize the length of the schedule constructed so far, performs even worse. Although

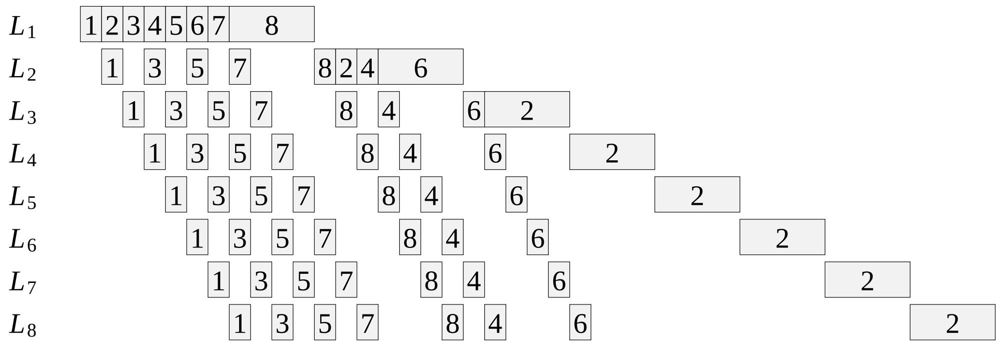


Figure 6.7: Alternative upper bound schedule for $P = 8$

for $P < M$, this algorithm clearly produces higher T^u values, for $P = M$ (i.e., the worst case), this alternative method is still outperformed by the first one, due to integer effects. Going from top-left to bottom-right, each a-segment i, $i = 1 \ldots P$ is directly preceded by $\lfloor \frac{P-1}{i} \rfloor$ b-segments. Hence,

$$\eta_{(P,P)} \leq \frac{P\gamma + \lfloor \frac{P-1}{1} \rfloor + \lfloor \frac{P-1}{2} \rfloor + \ldots}{\gamma + P - 1}$$

which does equal the previous result, but only when $P - 1$ is a power of 2 (in fact, all alternatives based on the same approach but only differing in the relative location of the a-segment, only equal the previous result under certain conditions).

So far, the analysis applies to parallel sections where in each process each resource is used only *once*. In general, however, the resource access pattern of many practical parallel sections is characterized by loops. The loop model on which the following analysis will be based is a direct extension of the Pamela model described earlier, i.e.,

$$L = \textbf{par}\ (i = 1, P)\ \textbf{seq}\ (n = 1, N)\ \textbf{seq}\ (j = 1, M)\ \textbf{use}(r_{s_{i,j}}, \delta_{i,s_{i,j}})$$

where N denotes the number of loop iterations involved. Note, that this deterministic loop model still differs from the "random" model involving multiple, randomly permuted resource accesses, which will be discussed in Section 6.3. However, the bound on η is expected to be somewhat lower than the previous case.

Intuitively, it is clear that the upper bound schedule can not be constructed by a mere concatenation of transient schedules without mutual overlap. Yet, a fully sequential schedule appears to be achievable up to a high degree. For $P = 2$, the schedule, shown in Fig. 6.8, is conjectured to be optimal, illustrating the fact, that generating a chain of a-segments at least involves 1 b-segment per a-segment. From the figure, it is seen that,

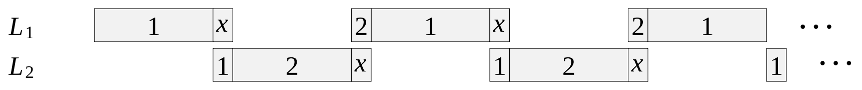


Figure 6.8: Upper bound schedule for the $(2, 3)$ problem (conjectured)

unlike the transient schedule, in order to maintain periodicity, a third resource (denoted x) is required. A $(3, 3)$ alternative to the above $(3, 2)$ scheme does not produce a higher η, since this, in turn, would require yet another resource[2]. Hence, the schedule, shown in Fig. 6.9 for the $(5, 5)$ problem, is conjectured to produce the upper bound. Again, the

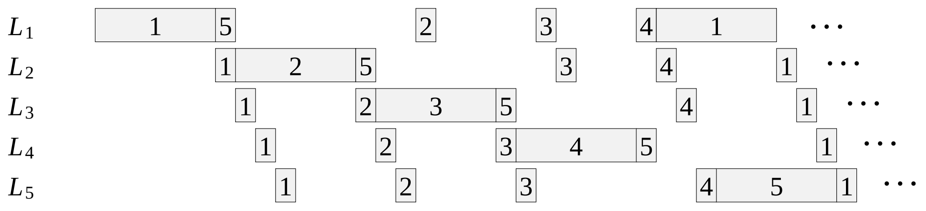


Figure 6.9: Upper bound schedule for $P = 5$ (conjectured)

sequential chain is determined by the first 4 processes, while the fifth process does not add to the upper bound (but neither to the lower bound). From the figure, it is easily seen, that

$$T^u = (P - 1)(\gamma + 1)b$$

Thus we conjecture

$$\eta_{(P,P)} \leq \frac{(P-1)(\gamma+1)}{\gamma + P - 1} \tag{6.2}$$

Compared to Eq. (6.1), the introduction of the loop decreases η by only a fraction.

From the above results, it would seem that the upper bound for deterministic loop systems is not fundamentally different from the case $N = 1$, apart from some parameter decrements. The maximum difference is a factor 2, that results from the following case.

[2]This essentially applies to the impossibility of inserting an extra a-segment. Even if some extra b-segments could still be inserted, in the limit, the $P - 1$ factor would still dominate.

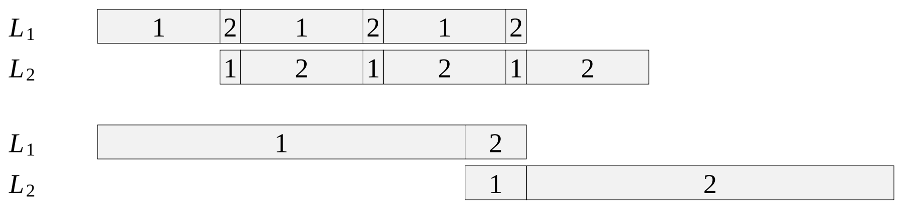


Figure 6.10: Loop vs. transient upper bound schedule

Consider the (P, P) loop system, where η is given by Eq. (6.2). If the loop enforcement were to be lifted in favor of a transient alternative, Eq. (6.1) would apply. In this alternative arrangement, all requests to the same resource would be concatenated together, rather than spread across the loops. It directly follows, that the ratio α between the upper bounds of this transient alternative and the original loop version is given by

$$\alpha = \frac{P(\gamma + 2) - 3}{(P-1)(\gamma + 1)}$$

which reaches a maximum value for $\gamma \to \infty$, and $P = 2$, i.e.,

$$\alpha \leq \frac{P}{(P-1)} \leq 2$$

For $\gamma = 6$, both cases are shown in Fig. 6.10 for $N = 3$, which indeed suggests the above phenomenon for large N.

6.2.4 General Conjecture

Although the derivations of the upper bound schedules have not been based upon rigorous proofs, there is experimental evidence to indicate that the results are at least close to reality. Apart from showing that the bound on η is limited by γ as well as P, the schedule of Conjecture 6.1 also suggests that approximately $3(P-1)$ resource accesses are involved in the determination of T^u. Also in more general cases, it may be assumed that the largest segments will be involved in the upper bound schedule. Note that this also applies for demand matrices that are *not* balanced. For example, consider a *column-dominant* demand matrix in which j-th column dominates. Figure 6.11 shows an upper bound schedule for $P = 5$ (where $j = 1$). Although in this case only $2P - 1$ segments are

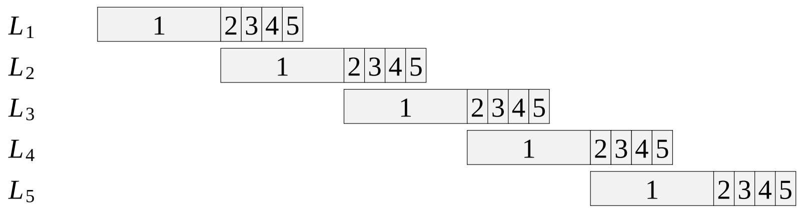


Figure 6.11: An upper bound schedule in column-dominant case ($P = 5$)

Run	T^u	$k = 6$	$k = 7$	$k = 8$	$k = 9$	k
1	5.55	4.64	5.17	5.58	6.16	8
2	6.02	4.87	5.47	6.05	6.61	8
3	4.27	4.00	4.45	4.82	5.07	7
4	5.94	4.46	5.13	5.80	6.44	9
5	5.06	4.43	4.84	5.24	5.61	8
6	5.98	5.06	5.71	6.27	6.81	8
7	5.85	5.71	5.34	5.79	6.23	9
8	5.09	4.50	5.08	5.59	6.06	8
9	5.10	4.38	4.95	5.38	5.79	7
10	5.83	5.54	6.22	6.58	6.92	7

Table 6.1: T^u vs. Eq. (6.3) for $k = 6 \ldots 9$ ($P = 4$), and the applicable k value

involved, Conjecture 6.1 essentially provides a sharp bound. Because the γ model used thus far is much to restrictive for practical applications we will now generalize the results obtained in terms of the following general conjecture.

Conjecture 6.2 *Let* $L =$ **par** $(i = 1 \ldots P)$ L_i *denote a parallel section involving* P *resources. Then*

$$T^u < \sum_{(i,j) \in \xi(k)} \delta_{ij}, \quad k = 3(P-1) \tag{6.3}$$

where $\xi(k)$ *denotes the set of* k *largest elements in* Δ. $\Box$

Thus, the upper bound on η can be simply determined by the size and structure of Δ, without explicit knowledge of special metrics like γ. Note that in general, the segments that determine the critical path need not exactly be the k largest segments. However, practically all the largest will be involved. In cases where γ is large, the error will become negligible. For diagonal-dominant cases it follows $T^u = \eta T^l$ where η is given by Theorem 6.1, while for column or row-dominant matrices T^u simply approaches T^l.

In order to test the above conjecture, for $P = 4$ we have run a series of 10 experiments based on random, balanced (i.e., worst case) Δ matrices where each element $\delta_{i,j}$ is i.i.d. uniformly[3] over the interval $[0, 1]$. Again, each experiment is based on exhaustive search. For all cases T^l is measured in agreement with Eq. (3.14). The results for T^u are shown in Table 6.1, compared to Eq. (6.3) for different values of k. Conjecture 6.2 is indeed verified to produce an upper bound that is reasonably sharp (partial results for $P = 5$ also agree with the conjecture).

The fact that Eq. (6.3) provides a reasonable estimator for T^u allows for a simple, first-order assessment of the average value of the upper bound of η in the above case of a Δ matrix with uniformly distributed elements. In the analysis, for T^u, we will assume the worst case, i.e., $k = 3P - 3$, in order to guarantee a valid upper bound. The analysis

[3]Note that the distribution applies to the overall demand matrix Δ, not to individual service times (which are deterministic). The choice for a uniform distribution is somewhat arbitrary. However, its low implementation cost is attractive considering the huge number of simulations involved.

is based on results from order statistics [36]. In Appendix D, it is shown that the mean of the upper bound can be estimated by

$$E(T^u) = \frac{P^2(P^2+1) - (P^2-3P+3)(P^2-3P+4)}{2(P^2+1)}$$

while for large P it holds $E(T^u) \sim 3P$.

In the appendix it is also shown that the mean of the lower bound can be estimated by

$$E(T^l) = \frac{P}{2} + 0.4\sqrt{P \log P}$$

Since for large P it holds $E(T^l) = P/2$, a coarse approximation[4] of the bound on η is given by

$$\lim_{P\to\infty} \eta_{(P,P)} \le 6 \qquad (6.4)$$

Thus, lifting the γ constraint by allowing elements to be uniformly distributed down to even zero, results in a mere doubling of η, compared to the fixed case of $\gamma = 1$. For small values of P the bound is much lower. For $P = 4$ the bound on η is predicted to be 2.1 which agrees with the measurements of T^l and T^u of the above random matrix case (average bound 2.1). Figure 6.12 shows a histogram of the measured η values over the 10 runs. Apart from the fact that $\eta \le 2$, the figure clearly demonstrates that the mean value

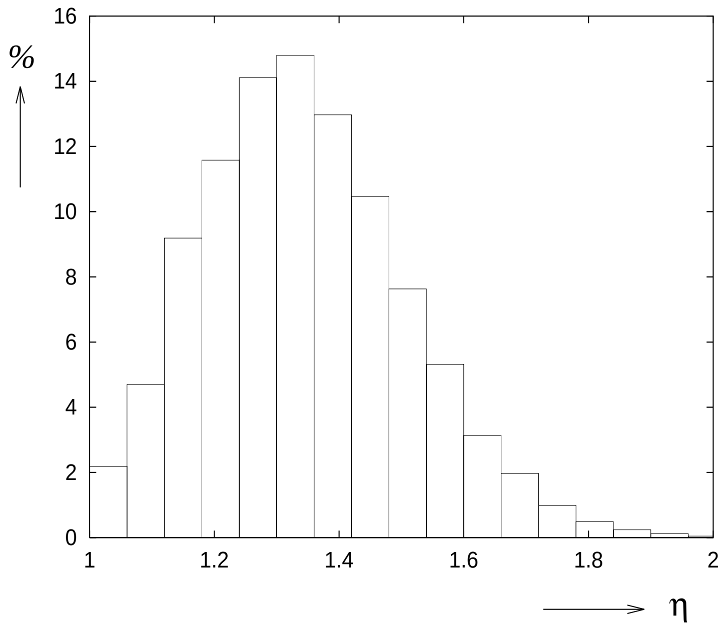


Figure 6.12: Distribution of $\eta_{(4,4)}$ for random Δ (uniform)

of η is typically much less than the upper bound.

[4]Of course, considering the division of the two mean values has no formal justification. Hence the use of the word "coarse approximation".

As discussed earlier, for loop models, the situation slightly improves. As discussed in the previous section, the maximum number of largest segments involved in the upper bound schedule would be $k = 2P - 2$ per iteration. Hence, for large N we have

$$E(T^u) = \frac{P^2(P^2+1) - (P^2 - 2P + 2)(P^2 - 2P + 3)}{2(P^2+1)}$$

while for large P it holds $E(T^u) \sim 2P$. The mean lower bound remains

$$E(T^l) = \frac{P}{2} + 0.4\sqrt{P \log P}$$

Consequently

$$\lim_{P\to\infty} \eta_{(P,P)} \leq 4 \tag{6.5}$$

Again, for small values of P the bound is much lower. For $P = 4$, η is predicted to be 1.6 which reasonably agrees with a series of 10 measurements (conducted for $N = 100$, average bound 1.42). Similar to the former (transient) case, the estimation for k appears to be about 1 unit higher than observed in practice which indicates that Conjecture 6.2 is quite reliable. Figure 6.13 shows the results which, again, illustrates that, on average, η is much smaller than the upper bound.

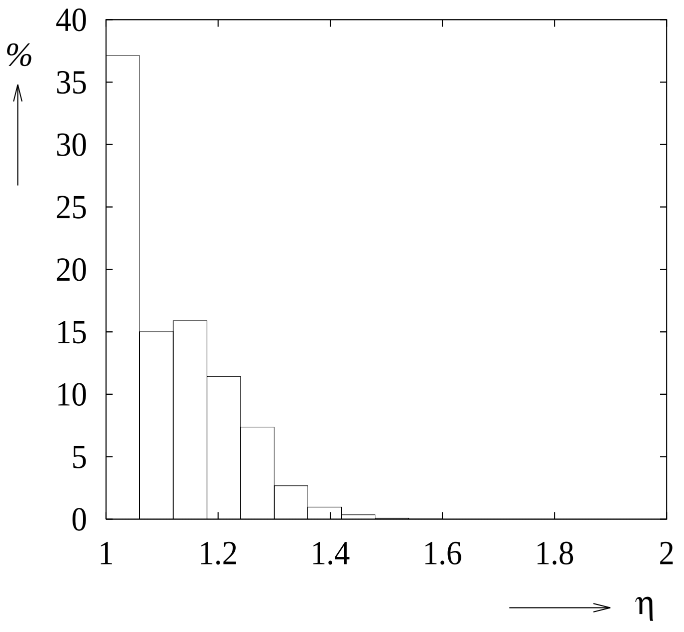


Figure 6.13: Distribution of $\eta_{(4,4)}$, for the loop model

6.2.5 Applications

In the following we show the use of the above results for an application to the matrix factorization and the matrix-vector multiplication case studies presented in the previous chapter.

First, recall the parallel LU factorization of an $N \times N$ matrix on a multiprocessor with P processors and M interleaved memory banks. Let Δ denote the $P \times M$ memory bank resource demand matrix. Let P^* denote the saturation point, i.e., the value of P after which memory bank saturation occurs. For small P, Δ is row-dominant, while for large P, Δ is column-dominant. Hence for those cases $\eta = 1$ which agrees with the high accuracy of T^l for these P values. For values of P close to P^* the largest deviation occurs between T and T^l. This corresponds with the fact that Δ is neither row-dominant, nor column-dominant. Because, due to the interleaving, Δ is neither diagonal-dominant, of the results derived in the previous section, Eq. (6.2) is the most characteristic for the above situation for $\gamma \approx 1$. It follows (large P) $\eta \leq 2$ which agrees with the measurements. The measured ratio is smaller, partly due to the fact that the floating point *delays*, scattered in between the resource accesses, have not been accounted for[5].

Next, recall the parallel matrix-vector update of an $N \times N$ matrix on the P node distributed-memory machine with a unidirectional ring of P point-to-point links $l_0 \ldots l_{P-1}$. Let Δ denote the $P \times P$ link resource demand matrix. As mentioned in the case study, for large P the communication phase dominates the performance and it holds $\varphi \approx \omega$. For $P = N = 4$, the resource access pattern is given by

	$i = 0:$	$i = 1:$	$i = 2:$	$i = 3:$
p = 0:	–	0	0,1	0,1,2
p = 1:	1,2,3	–	1	1,2
p = 2:	2,3	2,3,0	–	2
p = 3:	3	3,0	3,0,1	–

A typical trace of the link resource usage in the message-passing phase is shown in Fig. 6.14. In the following we only consider the case $P = N$. The communication phase

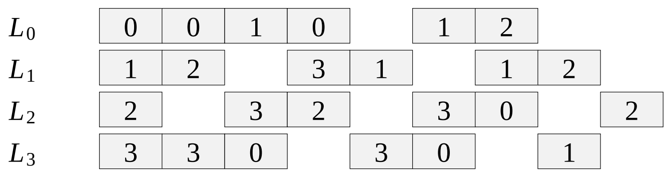


Figure 6.14: Communication phase for $P = N = 4$

is represented by the following $P \times P$ demand matrix, i.e.,

$$\Delta = \begin{pmatrix} P-1 & P-2 & \ldots & 0 \\ 0 & P-1 & \ldots & 1 \\ \vdots & \vdots & & \vdots \\ P-2 & P-3 & \ldots & P-1 \end{pmatrix}$$

[5]An extension of the analysis in the previous sections through the involvement of delay terms is beyond the scope of this initial study. However, it is clear that, inherently being *inert* operations, their effect is a decrease of η.

T	$P=4$	$P=8$	$P=16$	$P=32$	$P=64$
T^u	9	49	225	961	3969
T	9	43	207	900	3822
T^l	6	28	120	496	2016

Table 6.2: Timing results of the message-passing phase for $P = N$

Again, Δ is balanced and, to a slight extent, diagonal-dominant. Yet, it is not unrealistic to assume $\gamma \approx 1$ which, by Eq. (6.2) for large P implies $\eta \leq 2$. For $\tau_l = 1$, Table 6.2 compares actual timing results T with T^l given by

$$T^l = \frac{P(P-1)}{2}\,\tau_l$$

and T^u based on Eq. (6.2) for $\gamma = 1$ given by

$$T^u = \frac{2(P-1)}{P} T^l$$

From the table it would follow that T is near to T^u. Note, however, that the assumption $\gamma = 1$ is a bit conservative considering the near-zero entries far from the diagonal of Δ.

6.3 Average Accuracy

6.3.1 Introduction

In the previous section we have investigated T^u in order to assess the absolute deviation between T and T^l in the case of coarse-grain (task-level) contention. In this section, we study the effects of fine-grain (subtask-level) contention in terms of the *average* of T since for practical systems, typically featuring frequent, random resource access, the actual variance in T is much less compared to the above situation (although the histograms already provide an indication to this effect). As in the previous section, we only consider task graphs that do not comprise a sequence of subgraphs (such that Eq. (3.14) applies rather than Eq (3.15)). In contrast, however, we now consider any model structure comprising N tasks and M resources possibly involving parallel nestings.

As illustrated by, e.g., Example 3.7 and Example 3.8, for models in which the resource demand is reasonably uniform during the entire computation (i.e., in contrast to models such as in Example 3.10), T^l approaches the mean value of T either when $\varphi \gg \omega$ (critical path dominates) or when $\varphi \ll \omega$ (queuing dominates). Thus, in many cases the average error of the analytic prediction may be assumed to be quite acceptable (as results will show later on). As already mentioned in Example 3.7 the choice of the lower bound as a practical estimate is also inspired by similarities between the execution of L and interactive queuing systems. Although, formally, the resemblance is extremely remote it is interesting to relate the lower bound approach to the asymptotic bound analysis of an (operationally) comparable interactive queuing system[6] (see Fig. 6.15). If we define Z as

[6]Note that the comparison is purely intuitive as we disregard many details, e.g., the fact that each task should map to a unique job class; possible transient phases like startup and shutdown are ignored; the task graph should be cyclic in order to have steady state execution, etc.

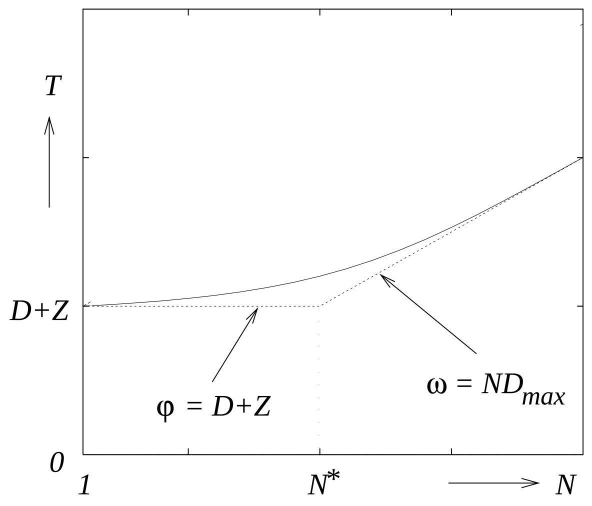


Figure 6.15: MRM cycle time interpretation in queuing theory and in Pamela

the think time [93], D as the total service demand and D_{max} as the service demand at the bottleneck device, we can interpret φ as the horizontal cycle time asymptote $D + Z$ (Z accounts for task synchronization delay), while ω corresponds to the ND_{max} asymptote. The largest deviation occurs at the saturation point, where $D + Z = ND_{max}$.

As a result of the above observations we propose to use an operational metric called "serialization index" or "contention index" that characterizes the degree of contention within a system. The metric is defined by

$$\theta = \log\left(\frac{\omega}{\varphi}\right) \tag{6.6}$$

The use of θ is to characterize a model as to the likelihood of T^l being an accurate prediction. For models with large $|\theta|$ the average accuracy of T^l is expected to be better than for models where $|\theta| \approx 0$. Note that θ bears a direct relationship with Δ since it holds

$$\varphi = \max_{i=1\ldots P} \sum_{j=1}^{M} \delta_{i,j}, \quad \omega = \max_{j=1\ldots M} \sum_{i=1}^{P} \delta_{i,j}$$

Thus a balanced or diagonal-dominant matrix (worst cases) indeed corresponds with $\theta = 0$.

6.3.2 Experiments

In this section we report on an experiment involving 1000+ random SP models in which the predictions T^l are compared to the simulation results T. The models are generated such that the θ values lie around the (worst case) region of interest ($|\theta| \approx 0$). Apart from the fact, that many computations of interest are SP structured[7], the choice for SP

[7]Note that the application range of Pamela SP models is essentially greater than just SP task graphs. For instance, pipelining is also expressed in terms of a parallel section of contending tasks (cf. Example 3.5).

models is also motivated by the fact that it enables an evaluation of the improvement of Eq. (3.15) on the accuracy compared to Eq. (3.14). Each model comprises $N = 100$ tasks while the number of resources involved varies from $M = 2 \ldots 150$. The graphs are generated by a simple algorithm that iteratively adds a new task t_i to a random selected task t_j within the graphs generated up to that moment (j is determined in each iteration). The probability that t_i is placed in series or parallel with t_j is determined by an input parameter, denoted s. Each task t_i is characterized by a unique service demand vector $\underline{\delta}_i = (\delta_{i,1}, \ldots, \delta_{i,M})$ in which each element is i.i.d. uniformly over $[0, 1]$. Thus, balanced systems are generated (on average). Experiments have verified that this choice indeed provides the worst case with respect to the accuracy of T^l. Each resource m is accessed multiple times based on the existence of some deterministic service time τ. Thus each task executes $\delta_{i,m}/\tau$ accesses to resource m. The order in which the resources are visited is random. In order to minimize simulation time (many models are simulated), τ is chosen such that the mean of T does not deviate significantly from results for $\tau \to 0$ (in practice, values in the order of 1 % of the largest service demand $\delta_{i,m}$ (i.e., ≈ 100 visits) have been found to suffice[8]). As N is fixed ($N = 100$), the parameters M and s determine the (mean) θ value of the generated models. As φ is proportional to M, large values of M will generate models with a negative θ. For low s, however, many parallel tasks are created on average which has a positive influence on θ.

Figure 6.16 shows the ratio T^l/T based on 1200 random models exhibiting θ values ranging from $-2 < \theta < 2$. Both the prediction ratios based on Eq. (3.14) and Eq. (3.15) are shown (α and β, respectively). Each data point of both series of 120 points represents an average value based on 10 random draws in order to reduce noise (for fixed generator parameters 10 models are generated). The results clearly reveal a high correlation between (α, β) and θ in which the deviation from unity is indeed maximal for models that exhibit $\theta = 0$. Thus, for random graphs the diagnostic value of the operational parameter θ appears to be quite significant, especially when considering the fact that two graphs with comparable θ values usually have quite a different structure. While the essential necessity of Eq. (3.15) has already been demonstrated (cf. Example 3.10), even for the models with uniform resource demand (in time) as produced by the random generator, its application still yields an improvement for models with highly parallel subsections (e.g., $\theta > 0$). In the following we will only consider α. In the above experiments the models are generated for $s = 0.1$ with M varying from $M = 2$ ($\theta \approx 2$) to $M = 150$ ($\theta \approx -2$). Models with $\theta \approx 0$ are generated for $M = 20$. For each value of M the variance of θ is approximately 0.05 which accounts for the reasonably continuous plot. Although no extensive experiments have been performed for different values of N, initial measurements indicate that the α curves tend to be more 'v'-shaped for small N, corresponding to the fact that the scale of θ is still somewhat dependent on the problem size. However, $N = 100$ also appears to be quite representative for larger models[9] as well. Additional measurements indicate that the *minimum* value of α at $\theta = 0$, i.e., α^*, highly correlates with M. For instance, each

[8]The justification is that for large visit counts the change in task-level variance proves to be small. This phenomenon has been observed for both exponential and deterministic service times. This optimization has been introduced in Section 5.4.

[9]For $N = 10$ the range of interest is $-0.5 < \theta < 0.5$, whereas for $N \geq 1000$ the range is still around $-2 < \theta < 2$.

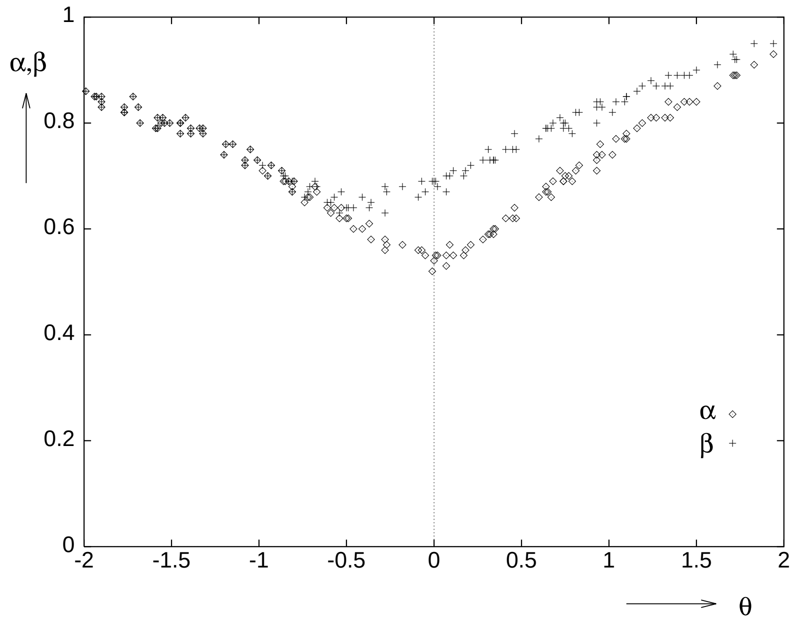


Figure 6.16: T^l accuracy (α, β) for 120 random SP models $(N = 100)$

of the following set of parameter tuples, i.e.,

$$(N, M, s) \in \{(30, 8, 0.1), (100, 8, 0.3), (300, 8, 0.5)\}$$

generates models with $|\theta| \approx 0$ that yield $\alpha^* \approx .6$ on average. The correlation is shown in Fig. 6.17 that plots a series of α^* values for various N, M, and s values. Each value is derived from an average of 20 random draws except for $N = 1000$ due to the computational costs involved. The results agree with the earlier observation that the α plot for $N = 100$ is quite typical for larger systems. The figure suggests the existence of a horizontal asymptote given by $\alpha^* \approx 0.5$. Again, it is tempting to compare this upper bound on the (mean) deviation with the result from asymptotic bounding analysis of interactive queuing systems. For instance, consider the MVA recursion for an M server balanced system [160] with total service demand D, given by

$$R(N) = D + \frac{D}{M} \, \frac{R(N-1)}{R(N-1) + Z} \, (N - 1) \tag{6.7}$$

where $R(N)$ denotes the response time as a function of the number of jobs in the system (N). Let $C(N) = R(N) + Z$ denote the mean cycle time (comparable to T). From Eq. (6.7) it follows that for small N the slope of $R(N)$ is less[10] than the asymptotic value D/M for $N \to \infty$. Consequently, at the saturation point $N = N^*$, for which the deviation between $C(N)$ and its lower bound $C^l = D + Z$ is the largest, it holds $C(N^*) < C^l + (D/M)N^*$. With $N^* = (D + Z)/(D/M)$ it follows $C(N^*) < 2C^l$, that corresponds to the lower bound on α^*. Generating models with large M also implies a large value for D. In terms of the analogy this implies a relatively decreasing Z. Indeed, from Eq. (6.7) it is easily seen that $\lim_{Z \to 0} C(N^*) = 2C^l$.

[10]An accurate analysis of the balanced upper bound is given by Zahorjan *et al.* [160]. However, for our purpose the above analysis suffices.

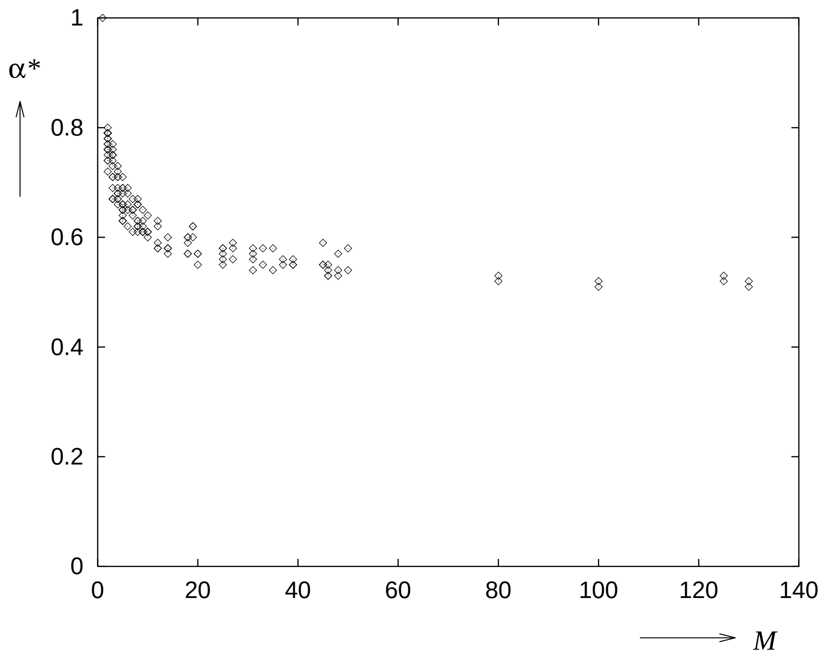


Figure 6.17: α^* vs. M for $N = 10 \ldots 1000$ and $s = 0.1 \ldots 0.5$

6.3.3 Applications

In the following we demonstrate the use of the above results by an application to the matrix factorization and the matrix-vector multiplication case studies presented previously as well as the macro data flow case study.

As mentioned earlier, the diagnostic value of Δ is also reflected in θ. Like the MRM the results of the factorization case study (in fact, an MRM-type model with M resources) are a good demonstration of the diagnostic value of θ. For $|P/P^*| \gg 0$ ($|\theta| \gg 0$) α (T^l/T) is close to 1, whereas for $P \approx P^*$ ($\theta \approx 0$) the deviation is maximal (note that α^* is much greater than 0.5 because of the few resources involved). The metric also predicts the deviation of T^l for the matrix-vector multiplication. For large P (i.e., $\theta = 0$ as the communication phase dominates) α^* approaches 0.5 as predicted by Fig. 6.17 (large number of link resources).

Next, recall the macro data flow case study that involved the discussion on the measurement results for the execution of 15 random SP graphs $G_1 \ldots G_{15}$ on the 4×4 transputer mesh. Table 6.3 shows the results of T^l in terms of α (applying Eq. (3.14) and β (applying Eq. (3.15)) as well as θ for the 15 graphs. The results for α and β indeed show that the average prediction error of serialization analysis is limited to a factor 2 while for models with $|\theta| \gg 0$ the predictions tend to approach the simulation results. However, the increase in accuracy for positive θ appears to be less when compared to the simulation results in Section 6.3. This phenomenon will be discussed later on. As expected, for relatively parallel graphs, the β values tend to be somewhat better than the α values.

As shown by the measurement results (Table 5.2) the accuracy of T is such that it is indeed acceptable to evaluate the accuracy of T^l by comparison to T only (as has been the basis for the previous correlation experiments). Since, simulation proves to be more

	G_1	G_2	G_3	G_4	G_5
α	0.66	0.53	0.60	0.53	0.56
β	0.73	0.62	0.68	0.70	0.66
θ	1.08	0.85	0.76	0.37	0.27
	G_6	G_7	G_8	G_9	G_{10}
α	0.55	0.54	0.61	0.78	0.75
β	0.55	0.60	0.68	0.78	0.75
θ	-0.19	-0.28	-0.35	-0.73	-0.91
	G_{11}	G_{12}	G_{13}	G_{14}	G_{15}
α	0.80	0.87	0.86	0.96	0.94
β	0.87	0.87	0.86	0.96	0.94
θ	-1.23	-1.51	-1.62	-1.70	-1.74

Table 6.3: Correlation between α, β and θ for $G_1, \ldots, G_{15}$

convenient than actually executing the graphs on the transputer mesh we have evaluated the accuracy of T^l for an additional 150 graphs. The results are shown in Fig. 6.18 for α and Fig. 6.19 for β. Unlike the theoretical plot of Fig. 6.16 each point now represents

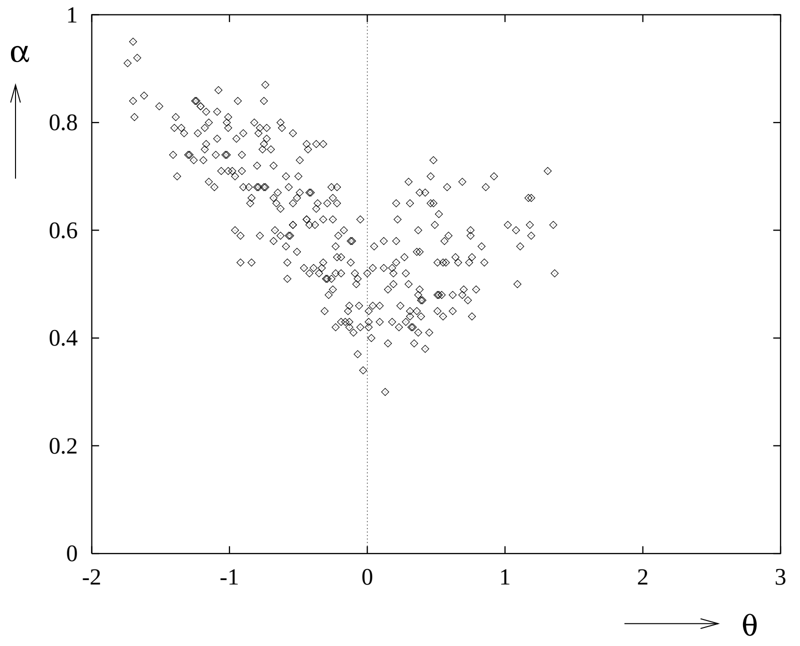


Figure 6.18: T^l accuracy (α) for 150 random SP models ($N = 100$)

an individual measurement, hence the noisy plots. Unlike Table 6.3 the plots give a good impression of the practical accuracy of T^l. Like in the original case study, each of the 150 graphs was also executed under a mode f in which all communication (except task synchronization) is switched off, and a mode c, in which all computation has been disabled. Figure 6.20 and Fig. 6.21 show the T^l accuracy for the f-mode execution, while Fig. 6.22 and Fig. 6.23 show the T^l accuracy for the c-mode execution. The results

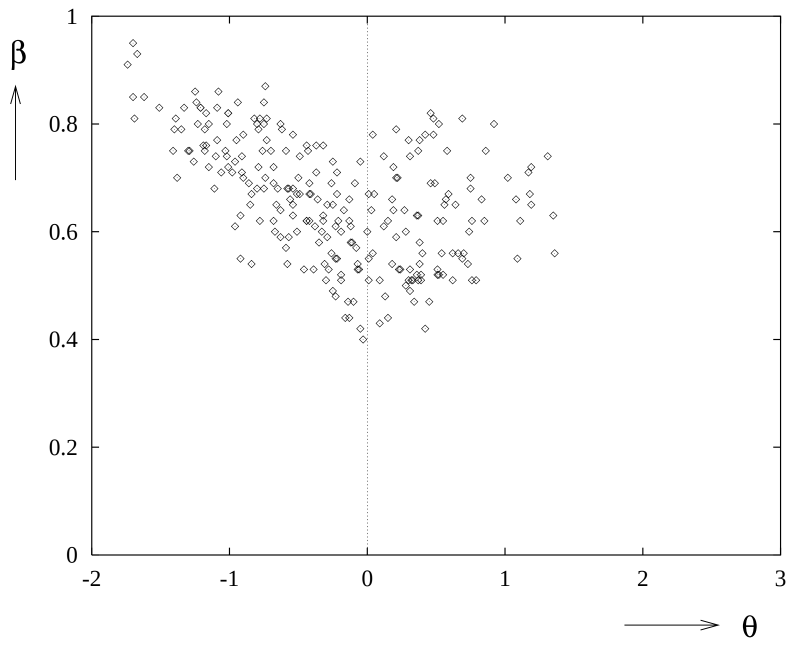


Figure 6.19: T^l accuracy (β) for 150 random SP models ($N = 100$)

confirm the simulation experiments which indicate that the *average* deviation between T^l and T is typically limited to $\alpha^* \approx 0.5$ in the worst case ($\theta = 0$, note that $M = 144$).

From the plots it is clear that the limited increase in accuracy for positive θ as mentioned earlier is due to the communication. (The computation plots, in contrast, completely agree with the correlation experiments.) Likewise, it are the communication mode graphs that exhibit quite some samples with $\alpha < 0.5$. This is caused by the fact that parallel communications involving the same resources may occur at relatively concentrated points in time, combined with the fact that the communication system interleaves packet transfers sharing the same links. The virtual barriers caused by this multiple interleaving across the same resources are not accounted for by T^l which yields a less tight bound than might be expected. We show the mechanism involved through a simple example (see Section 5.3 for modeling details). Consider the following task graph

$$G = t_0;\ \mathbf{par}\ (i = 1, N)\ t_i;\ t_{N+1}$$

where $\underline{p} = (0, 1, \ldots, 1, 0)$. Let $\underline{l} = (10^6, \ldots, 10^6, 0)$ and $\underline{w} = \underline{0}$. Then L is effectively given by

$$L = \mathbf{par}\ (i = 1, N)\ \{bmove(0, 1, 10^6)\ ;\ bmove(1, 0, 10^6)\}$$

The fanout phase (node $0 \rightarrow 1$) involves service e_1 while the subsequent fanin phase (node $1 \rightarrow 0$) involves service e_0. The lower bound corresponds to a schedule in which service requests to e_0 and e_1 would be (partially) overlapped as a result of an initial skewing during the fanout phase. It follows $T^l = 0.9N$ s. In reality, however, this overlap does not occur due to the fact that the $bmove(0, 1, \ldots)$ tasks (as well as the $bmove(1, 0, \ldots)$ tasks) execute simultaneously as a result of the interleaving at packet level (fair scheduling).

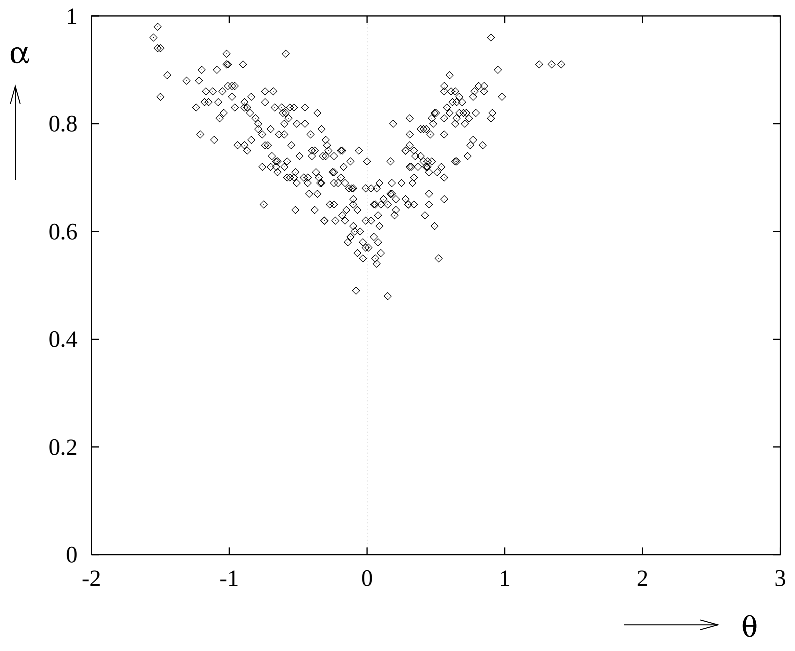


Figure 6.20: T^l accuracy (α) for 150 random SP models ($N = 100$) in f-mode

Indeed it holds $T = 1.8N$ s which is also measured in practice. (The Pamela simulation model accounts for this phenomenon, of course.) Thus, while θ is large ($\log N/2$) the value of α is still 0.5 (for $\underline{w} \neq \underline{0}$ the situation becomes much better). Effectively, the interleaved execution of the *bmove* tasks can be approximated by a virtual barrier (see Section 5.4) according to the following model

$$L = \textbf{par}\ (i = 1, N)\ bmove(0, 1, 10^6)\ ;\ \textbf{par}\ (i = 1, N)\ bmove(1, 0, 10^6)$$

It follows $T^l = T = 1.8N$ s. For the 15 test graphs as well as the 150 additional graphs we have included the barrier synchronization at the end of the concurrent broadcast for each task as described above. Indeed the results for β greatly improved in accordance with the correlation experiments ($\beta^* = 0.7$ instead of 0.45).

6.4 Summary

In this chapter we have studied the absolute accuracy as well as the mean accuracy of T^l relative to T. The results are based on theory, simulation studies, as well as actual machine measurements.

With regard to the *absolute* deviation between T^l and T, for parallel sections, it is conjectured that the ratio η between the lower bound and the upper bound is always less than a constant factor, depending on the total resource demand, characterized by the demand matrix Δ. Thus, given a lower bound prediction, the bounded value of η indicates within what range the actual execution time might differ from the prediction. Without specific information on Δ, it is shown that η may range between 1 and $\min(P, M)$, i.e., parallel sections may run fully sequential up to fully in parallel. However, given a ratio

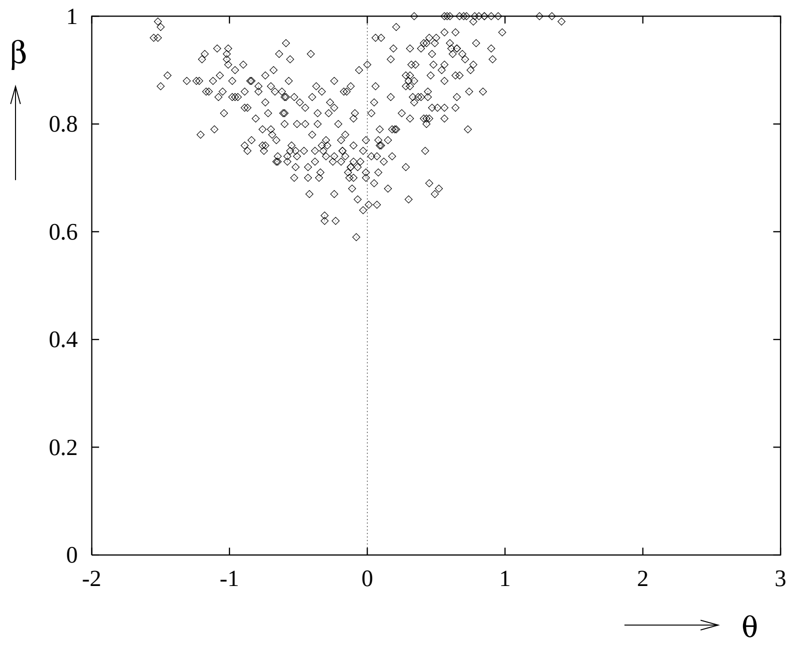


Figure 6.21: T^l accuracy (β) for 150 random SP models ($N = 100$) in f-mode

γ between the largest and smallest resource demands, for large P, M, it is shown that η only scales linearly with γ. All results have been verified by measurements, up to $P, M \leq 4$, in view of the huge computation costs involved. Although the results are in terms of η, they correspond to equivalent results on the upper bound T^u, thus providing a rudimentary tool for real-time system constraint verification. In order to obtain at least some insight for more "practical" cases, a simple statistical analysis is presented, assuming a Δ matrix with uniformly distributed elements, thus lifting the γ constraint. Again, the results suggest that, in practice, η is bounded by a constant, irrespective of P and M. Based on these results for η, by Eq. (6.3), a general estimator for the upper bound T^u is proposed, which, in particular, has been used in the above, statistical analysis. Although the estimator is correct for all synthetic cases, in random cases the value of its parameter k has only been validated for $P = 4$. Although experiments for $P > 4$ would provide crucial evidence in support of the conjecture, that $k \sim 3P$ for a generally valid upper bound, in the present workstation environment the associated computation costs proved to be prohibitive. If the estimator would indeed prove to be a reasonably tight upper bound this might be an important step towards the derivation of an upper bound for general task graphs. The proof of upper bounds on η is also useful in order to justify various reductions as described in the previous chapter. For instance, the (frequent) reduction of N similar **use** statements into one single statement, is allowed because the difference in resulting execution time (corresponding to the different schedule spaces) will be *limited*: both the lower bound *and* upper bound stay valid (the upper bound of the N statement case equals the single statement case considering the possibility of a schedule that exactly concatenates the N statements). This initial study only derives some basic results in what appears to be a very interesting area. Many more questions remain to be answered as the

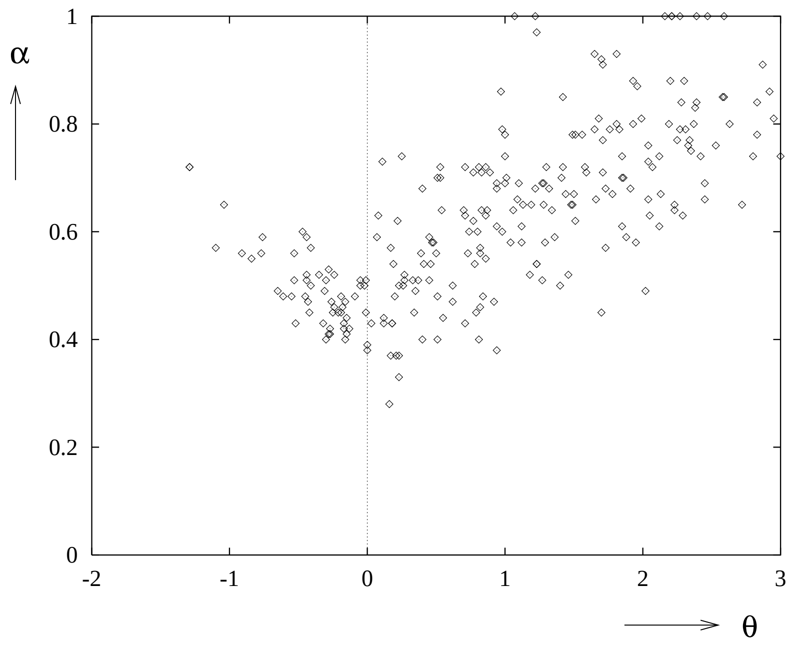


Figure 6.22: T^l accuracy (α) for 150 random SP models ($N = 100$) in c-mode

road towards a η framework for arbitrary task graphs is clearly a long one.

In this chapter we have also studied the *average* deviation between T^l and T both through a simulation study involving 1000+ random SP graphs as well as through measurements of 15 random SP graphs executed on a 16 node transputer mesh. Results of both case studies, show that the worst-case penalty for large random task systems is a mere 50 % under-estimation on average. Moreover, it is shown that for large random parallel sections the error of T^l compared to the average execution time can be predicted by the serialization index θ, that is symbolically compiled as a side result of T^l. However, like Δ, this metric should be used judiciously. The simulation study shows that for sufficiently random systems T^l approaches T except when $|\theta|$ is small. While this generally holds for low contention levels, the communication experiments show that for high contention levels T^l may still deviate from T due to the "virtual barrier" effect. Nevertheless, its diagnostic value is shown to be considerable for systems in which the resource demand is more or less distributed across a parallel section. Especially for large systems (in terms of the number of resource accesses) the shape of α can be interpreted in terms of the results of asymptotic bound analysis in queuing theory. This implies that the task-level synchronizations do not dominate overall synchronization behavior. Consequently, the performance of a large parallel task section with fine grain contention can be approximated by a queuing model in which the task synchronizations are (necessarily) ignored. Also note, that for large random systems where the (fine grain) resource accesses are uniformly distributed across the entire computation the factor 2 difference between T^l and T also has a remote correspondence with the factor 2 result from scheduling theory mentioned in the introduction. Because of the uniform distribution of the resource usage, for systems with $\theta \geq 0$ there is relatively little unforced idleness in the utilization of each resource, which implies

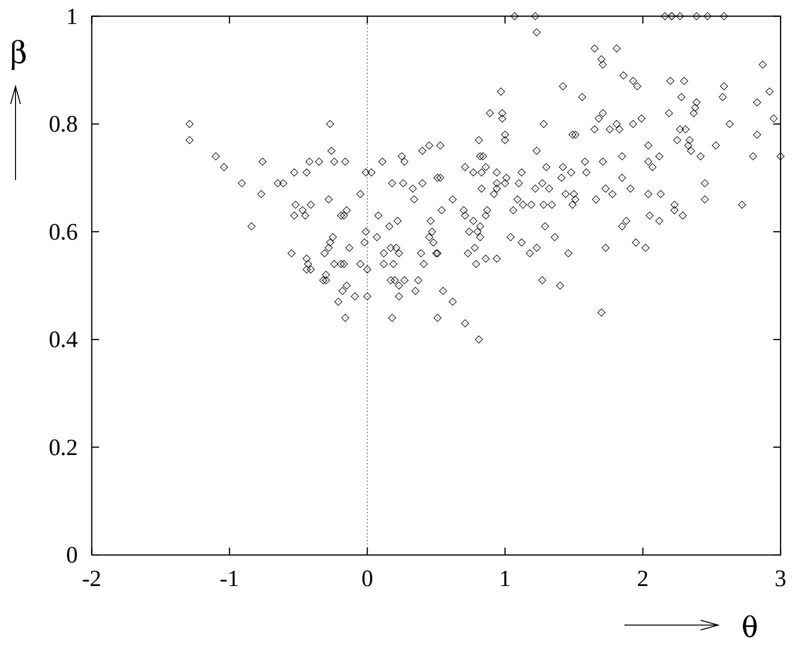


Figure 6.23: T^l accuracy (β) for 150 random SP models ($N = 100$) in c-mode

that the actual schedules are effectively work conserving.

Chapter 7

Conclusion

7.1 Contributions

In this dissertation we have discussed methods for the performance modeling of parallel computer systems. In order to resolve our conflicting goals that the model should have an explicit, analytical form, have an extremely low solution cost, and at the same time be reasonably accurate, we have introduced a modeling approach based on a new representation formalism that features certain restrictions with respect to the modeling of synchronization. Because of these restrictions, a static analysis technique can be applied yielding low-cost analytic models that are robust in terms of accuracy across the entire parameter space. Essentially, our approach is to emphasize the analyzability criterion rather than choosing for the appealing features of full modeling power. Throughout this dissertation we have shown that the subset of parallel computer systems that can be adequately modeled using our restricted approach does not differ substantially from those using alternative methods. Consequently, the approach strikes a good balance between the modeling power needed for a reasonable accuracy at PAMELA level as well as the subsequent analyzability needed for a reasonable accuracy of the resulting models in the time domain.

In summary, our major contributions can be stated as follows

- Formalism
  The introduction of the performance modeling language PAMELA. Although featuring all constructs for full modeling power (for compatibility with alternative analysis techniques) the language is specifically designed towards supporting the structured modeling of synchronization. Examples are the fork/join construct for condition synchronization (**par**), and the structured form for mutual exclusion (**use**).

- Paradigm
  The introduction of a parallel computer systems modeling methodology that is based on the use of a material-oriented paradigm in combination with highly structured synchronization operators, especially with regard to mutual exclusion. Results show that the sacrifice of modeling power due to the enforcement of structure is well balanced against the limited accuracy of the underlying analysis technique.

- Analysis
  The introduction of an analysis technique that integrates an approximate analysis of mutual exclusion within a conventional condition synchronization analysis technique. As a result, PAMELA models can be compiled into low-cost, analytic models that have a sustained minimum prediction accuracy with respect to both forms of synchronization across the entire parameter range.

The novelty of the approach is that it integrates the above concepts in a balanced way within one single methodology. As the language embodies these concepts by its programming paradigm and underlying compile-time calculus, we coin the overall approach the PAMELA methodology.

Apart from presenting this mathematical framework for the description and performance analysis of parallel systems the following contributions have been made

- Related work
  We have presented a survey of the main approaches to performance modeling of parallel systems, employing a taxonomy based on the various representation formalisms that are used. An essential aspect of this taxonomy is the distinction of three forms of control flow, i.e., condition synchronization, mutual exclusion, and conditional control flow. We have shown that the alternative approaches either (1) have a solution cost which is too high (e.g., Petri nets, process algebras, hybrid queuing networks), or (2) have a modeling power that is too low (e.g., queuing networks, task graphs), or (3) yield non-deterministic results (e.g., simulation). (Note, that each approach can be in more than one category.)

- Modeling approach
  We have presented our unified, top-down approach toward modeling shared-memory as well as distributed-memory (vector) programs and machines. Unlike many approaches, *all* software and hardware resources are treated exactly the same in terms of a chain of concurrent subroutines called by some root process that represents the algorithm. We have shown that the use of our restricted modeling paradigm allows us to capture the most important performance aspects that are relevant in the context of our approximate analysis.

- Case studies
  We have discussed a number of modeling case studies and small examples showing the use of the analysis technique in compiling PAMELA models into performance models. The data flow application study has shown that application behavior as measured in practice can be captured with reasonable accuracy even by simple PAMELA models. While discussing the concept of simulation we have presented extensions to our analytic technique as well as alternative (numeric) solution techniques.

- System optimization
  We have demonstrated the use of the PAMELA methodology in system optimization. We have shown that our contention modeling approach provides sufficient prediction power and yields low-cost, symbolic optimization solutions that can be easily

compiled and evaluated at run-time. In general, this implies that any intermediate (or even programming) language that is subject to automatic optimization procedures also be best defined in terms of a procedure-oriented paradigm with structured synchronization constructs.

- Accuracy
  We have investigated the accuracy of our static technique with respect to the approximation of the effects of mutual exclusion. With regard to the absolute variance of T (and hence, with regard to our lower bound T^l) we have presented a number of conjectures implying that the absolute variance can be computed as a function of the system demand matrix. With regard to the typical average deviation of T^l relative to T we have shown that for systems with random resource access patterns the worst-case deviation due to contention is limited to 50 %, throughout the entire parameter range. In view of the attractive cost features, this model robustness forms the ultimate justification of our approach.

With respect to related techniques our approach can be distinguished as follows.

- Modeling
  Unlike formalisms such as Petri nets and process algebras, the modeling power due to the **use** operator in PAMELA is limited. Both Petri nets and process algebras associate with a synchronization paradigm where mutual exclusion is expressed in terms of the basic non-deterministic control flow mechanism (conflicts in Petri nets, alternative composition in process algebras). While the choice for this low level construct necessitates state space analysis, our higher-order construct allows the use of our low-cost, symbolic analysis technique.

  Unlike approaches based on queuing networks or task graphs (stochastic or deterministic), PAMELA offers operators to account for *both* types of synchronization. This provides the minimum modeling power needed to realistically capture parallel system performance. Unlike hybrid queuing approaches, PAMELA does not distinguish separate formalisms to model programs and machines. Our unified approach allows the expression of both synchronization types for both programs and machines.

  Like process algebras and simulation languages PAMELA features composition operators that facilitate model construction. Unlike both formalisms, however, PAMELA features a symbolic analysis technique that really takes advantage of the possibility to describe parameterized models.

- analysis
  Unlike prediction techniques based on simulation languages, PAMELA extends traditional performance simulation approaches by offering a compile-time calculus that offers the opportunity of various forms of optimizations.

  Unlike stochastic approaches based on Petri nets, process algebras, queuing networks and stochastic task graphs, our analysis technique is not based on a state space analysis that entails both a costly and essentially numeric process. While trading exponential analysis complexity for a typically linear complexity, at the same time the average relative prediction error due to mutual exclusion synchronization is limited to a small constant.

The advantage of using structured mutual exclusion constructs has been recognized in queuing theory as can be seen by the existence of alternative solution techniques that have polynomial complexity. Although our lower bound technique is reminiscent of the bounding analysis in traditional queuing theory, our approach accounts for the effect of condition synchronization as it incorporates a critical path analysis.

Unlike the critical path analysis techniques associated with stochastic or deterministic task graphs that account for the effects of condition synchronization, our approach to static (compile-time) analysis incorporates an approximation of the effect of mutual exclusion, yet without entailing any increase in analysis complexity.

In summary, Pamela combines many insights from related approaches to the performance modeling of parallel systems, i.e., simulation modeling approach, process algebraic description, providing sufficient, yet limited modeling power enabling a low-cost, symbolic performance compilation technique inspired by static path analysis from task graphs and bounding analysis from queuing theory. The various ingredients are chosen in such a way that *analytical* performance models can be compiled that evaluate at the *lowest* possible cost, while the loss of accuracy due to the limited modeling power as well as the approximate analysis is kept to an *acceptable* level for first-order system design. To the best of our knowledge, this specific blend has not yet been introduced.

7.2 Improvements

The work we have described touches upon many fields such as concurrency, languages, simulation, scheduling, compile-time analysis, probability, discrete mathematics, complexity theory. As the main purpose of this dissertation is to argue that our approach to performance modeling of parallel systems satisfies the requirements as mentioned in the introduction, none of the above aspects have been treated in great depth. Clearly, the possible improvements to this work is numerous.

Apart from the obvious necessity of performing much more validation studies based on an extensive set of real-world applications modeling, some of the most important improvements of the current approach are the following.

- Modeling technique
  - Thus far, the trade-off between the machine-oriented and the material-oriented paradigm with respect to analyzability has been discussed rather informally. Because of the potential implications of an increased analyzability with respect to system design, this issue warrants a much more fundamental treatment in terms of concurrent formalisms in general, and models of parallel computation in particular.
  - Also the implications of the **use** restriction with respect to modeling concurrent systems that require a two-way synchronization (e.g., bounded buffers) has only been touched upon briefly. Being the essential restriction upon which our

static analysis approach is based, its implications must be investigated more thoroughly.

- Analysis accuracy
  - Merely intended to show that there exists a correlation between θ and α (β) the simulation experiments used to investigate the average accuracy of T^l have been necessarily brief. More experiments are necessary including the use of actual system measurements as well as extending the experiments to non-SP graphs, possibly with conditional control flow.
  - Although in many cases, an analysis result in terms of a (μ, σ) tuple provides sufficient information, in specific cases the absolute range in which T lies can be of interest (e.g., real-time systems). While the conjectures with respect to T^u as presented do provide insight in the basic properties of contention models, they only apply to simple parallel sections, and are supported by only few experiments. Clearly, the approach reveals an area where there is ample room for much improvement and where interesting results can be expected.
- Analysis technique
  - As a consequence of our focus on the effects of synchronization, one of the most important flaws of our approximate technique is the fact that task time variance (due to conditional control flow or mutual exclusion) is not accounted for. Especially, the offset of the mean value due to barrier synchronization may become quite large for high variance levels. By incorporating (symbolic) approximations for the offset, a better version of the static analysis may be developed where each task time parameter is represented by a (μ, σ) tuple rather than by a one (mean) value alone.
  - As shown by a number of examples and case studies, the PS-like resource sharing at aggregate level may cause an inaccurate lower bound estimate since the extra "virtual barriers" are not considered by the (default) lower bound technique. Hence, a preprocessing phase must be added to account for the synchronization effects that are essentially due to our assumption of fair scheduling.
  - As discussed, there are many cases in which a simulation run of a PAMELA contention model yields a result with only a small variance (assuming negligible conditional control flow variance). For these cases, it is possible to formulate an "analytic" technique for any PAMELA **P/V** model (i.e., not only **use** models as in the symbolic technique), based on manipulating resource queues. While the solution complexity of the technique is comparable to T^l (not considering possible reductions), the prediction accuracy is essentially better.
  - While the above alternative analysis techniques are not specific to PAMELA, neither are the techniques that are associated with alternative representation formalisms. For example, the above techniques could be complemented by defining a Markov analysis for PAMELA models, or a mean value analysis technique for the PAMELA equivalent of separable queuing networks, or extending

our static technique based on the analysis principles used in hybrid queuing networks. With its mixture of unstructured and structured synchronization operators, PAMELA could host a large variety of analysis techniques, thus offering a more flexible trade-off between cost and accuracy in the performance modeling of parallel systems.

Next to the above improvements, recommendations for future work clearly include the development of tools (e.g., PAMELA compiler) in order to support the application of the PAMELA methodology.

Appendix A

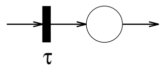
PAMELA Language Semantics

A.1 Introduction

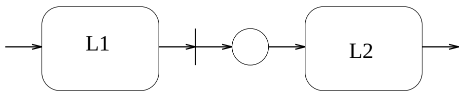
In this appendix we will describe the semantics of the most important PAMELA statements in terms of Deterministic and Stochastic Petri Nets (DSPN [6]). The choice for this formalism is motivated as follows. Because of the necessity to describe *time* as well as synchronization, traditional concurrency formalisms (e.g., CSP) are not appropriate, unless some *ad hoc* enhancements were introduced. As explained in Chapter 2, neither QN, SG, or DG have sufficient modeling power. In contrast, DSPN readily provides a simple means to express the semantics of the PAMELA constructs as far as synchronization and time are concerned.

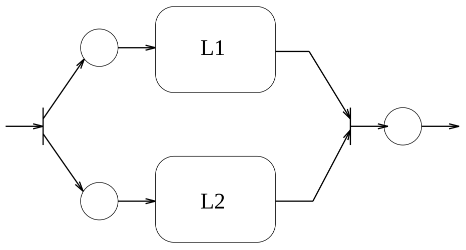
Because of the simple semantics of the language we will only model the most important constructs. The order of appearance of each construct is the same as in Chapter 3. As usual, thin bars denote immediate transitions, whereas thick bars denote timed transitions (either stochastic or deterministic). The action associated with each PAMELA construct is expressed by one or more transitions. Each DSPN representation of a PAMELA construct always ends with a place. As a result, like the composition of PAMELA constructs in a large model, each DSPN representation can be directly interconnected to form the DSPN representation of the PAMELA composition. DSPN representations of composite models (representing, e.g., L_1 or L_2) are depicted by boxes. Corresponding to the above convention, each box internally starts with a transition and ends with a place. Tokens either represent the thread of control of each PAMELA process, or represents the amount of resources that are available. For simplicity, in the DSPN models tokens have been omitted, i.e., all resources are assumed to have a zero value ($r_i = 0$). As each model starts with a transition, the transition is assumed to be enabled, without the presence of tokens being required.

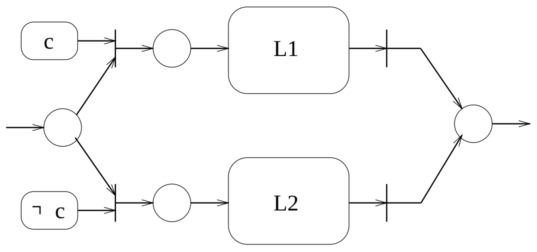
A.2 Time

PAMELA construct	DSPN representation
delay(τ)	

A.3 Control Flow

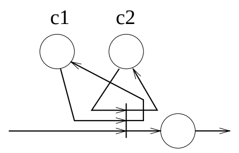
PAMELA construct	DSPN representation
$L_1\ ;\ L_2$	

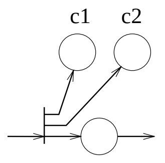
PAMELA construct	DSPN representation
$L_1 \parallel L_2$	

PAMELA construct	DSPN representation
if (c) L_1 **else** L_2	

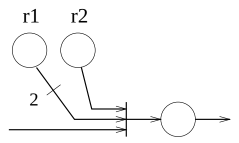
With respect to the **if** statement, c denotes a boolean computation which, in general, involves an extended PN (e.g., DSPN extended with inhibitors). Note that a stochastic choice would be implemented using two immediate transitions only (i.e., without the c subnet) specifying the appropriate switching distribution [5].

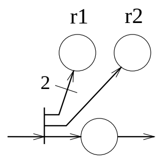
A.4 Condition Synchronization

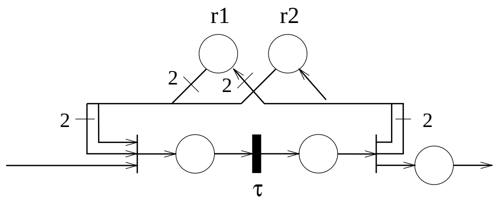
PAMELA construct	DSPN representation
wait$(\{c_1, c_2\})$	

PAMELA construct	DSPN representation
signal$(\{c_1, c_2\})$	

A.5 Mutual Exclusion

PAMELA construct	DSPN representation
P$(\{r_1, r_1, r_2\})$	

PAMELA construct	DSPN representation
V$(\{r_1, r_1, r_2\})$	

PAMELA construct	DSPN representation
use$(\{r_1, r_1, r_2\}, \tau)$	

Note, that the above **use** construct applies to (regular) FCFS resources. As defined in Chapter 3, for PS resources the number of sequential replications is infinite while τ approaches zero. Also the **using** construct is defined in terms of the above constructs, as defined in Chapter 3.

Appendix B

Partitioning Index Spaces

B.1 Introduction

In this appendix we derive expressions pertaining to the block-wise and cyclic partitioning of an indexed computation over M resources $m = 0 \ldots M-1$. Let

$$I = \{ i \mid a \leq i \leq b \}$$

be an index set. Let $f(i) = ci + d$ be an affine index function generating the references $f(a), f(a+1), \ldots, f(b)$ associated with the computation. Let π denote the partitioning function. Then the index partition associated with resource m is given by

$$I_m = \{ i \mid a \leq i \leq b \ \wedge \ \pi(f(i)) = m \}$$

In the following we will give some reductions of the above set enumeration to a less complex form for block and cyclic partitioning.

B.2 Block Partitioning

Let

$$\pi(i) = \lfloor \frac{i}{B} \rfloor$$

denote the block partitioning function where

$$B = \lceil \frac{b-a+1}{M} \rceil$$

denotes the block size. Then I_m comprises the solution of the equation

$$\lfloor \frac{ci+d}{B} \rfloor = m$$

This implies

$$Bm \leq \lfloor \frac{ci+d}{B} \rfloor \leq B(m+1) - 1$$

and it follows

$$\lceil \frac{Bm-d}{c} \rceil \leq i \leq \lceil \frac{B(m+1)-d}{c} \rceil - 1$$

subject to

$$a \leq i \leq b$$

Hence, in terms of a consecutive series j, I_m is given by

$$I_m = \{i |\ \alpha_m \leq j \leq \beta_m\}$$

where

$$\alpha_m = \max\left[a, \lceil \frac{Bm-d}{c} \rceil\right], \quad \beta_m = \min\left[b, \lceil \frac{B(m+1)-d}{c} \rceil - 1\right]$$

B.3 Cyclic Partitioning

Let

$$\pi(i) = i \bmod M$$

denote the cyclic partitioning function. Then I_m is the solution of the diophantine equation

$$ci + kM = m - d, \ \ a \leq i \leq b$$

where k is some integer variable. If no solution in i exists, i.e.,

$$(m-d) \bmod \gcd(c, M) = 0$$

then $I_m = \emptyset$, else I_m is given by the monotonic series

$$I_m = \{\iota_m, \iota_m + \delta, \iota_m + 2\delta, \ldots\}$$

where $\iota_m \geq a$ denotes the smallest solution in i of the diophantine equation, and the solution period is given by

$$\delta = \frac{M}{\gcd(c, M)}$$

In terms of a consecutive series j, I_m is given by

$$I_m = \{\iota_m + \delta j \mid \alpha_m \leq j \leq \beta_m\}$$

where

$$\alpha_m = \lceil \frac{a - \iota_m}{\delta} \rceil, \quad \beta_m = \lceil \frac{b + 1 - \iota_m}{\delta} \rceil - 1$$

B.4 General Results

Since

$$I = \bigcup_{m=0}^{M-1} I_m$$

it holds

$$\sum_{m=0}^{M-1} \beta_m - \alpha_m + 1 = b - a + 1$$

An important measure is the maximum partition size, i.e.,

$$S = \max_{m=0\ldots M-1} \beta_m - \alpha_m + 1$$

For block partitions this simply reduces to $S = B$. For cyclic partitions, a situation in which the partition I_m is a member of the larger power set occurs when $\iota_m = a$ (i.e., the first index is a direct hit). Let m' be the solution to the above condition. Then

$$S = \beta_{m'} - \alpha_{m'} + 1 = \lceil \frac{b - a + 1}{\delta} \rceil$$

For the simple linear function $f(i) = i$ a simple expression can be obtained for a cyclic partitioning. It follows $c = 1, d = 0$ which implies $\iota_m = m - d = m$, and $\delta = M$. Thus I_m is given by

$$I_m = \{m + Mj \mid \alpha_m \leq j \leq \beta_m$$

in which

$$\alpha_m = \lceil \frac{a - m}{M} \rceil, \quad \beta_m = \lceil \frac{b + 1 - m}{M} \rceil - 1$$

Appendix C

Reduction of Summation Terms

In this appendix we derive reductions for the expressions

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil$$

and

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil n$$

We proceed by subdividing the range $n = 1, \ldots, N-1$ in subranges where the term

$$\lceil \frac{n}{P} \rceil$$

is constant and larger than 0. Let

$$J = \lceil \frac{N-1}{P} \rceil$$

denote the total number of full subranges, and let

$$K = (N-2) \bmod P$$

corresponding to the highest index of the entries $(0, \ldots, K)$ in the highest subrange in the case the subrange does not contain P entries. With respect to the first expression, it follows

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil = \sum_{j=0}^{J-1} jP + (K+1)J$$

which reduces to

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil = \frac{1}{2} J(J-1)P + J(1+K)$$

With respect to the second expression it follows

$$\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil n = \sum_{j=1}^{J-1} \sum_{k=0}^{P-1} [j(1+(j-1)P) + jk] + \sum_{k=0}^{K} J(1+(J-1)P) + Jk$$

which reduces to

$$\begin{aligned}\sum_{n=1}^{N-1} \lceil \frac{n}{P} \rceil n &= \frac{J(J-1)(P-P^2+P(P-1)/2)}{2} + \frac{J(J-1)(2J-1)P^2}{6} + \\ & (K+1)J(1+(J-1)P) + \frac{K(K+1)M}{2}\end{aligned}$$

Appendix D

Bounds for Random Parallel Sections

D.1 Introduction

In this appendix we derive expressions for the mean of T^u and T^l for a $P \times P$ demand matrix of which the elements are uniformly distributed over the interval (0,1). First, we define some basic terminology from the field of order statistics [36]. Let

$$X_{(1)} \leq X_{(2)} \leq \ldots \leq X_{(n)}$$

denote n variates $X_1, \ldots, X_n$, each with a cumulative distribution function $P(x)$, arranged in *ascending order*. Then $X(i)$ is called the ith *order statistic*. The cumulative distribution function $F_r(x)$ of $X_{(r)}$ is given by

$$F_r(x) = \sum_{i=r}^{n} \left(\begin{array}{c} n \\ i \end{array} \right) P^i(x)(1 - P(x))^{n-i}$$

Let f_r denote the probability density function of $X_{(r)}$. Then the mean of $X_{(r)}$ is given by

$$\mu_r = \int_{-\infty}^{+\infty} x f_r(x) dx = n \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right) \int_{-\infty}^{+\infty} x P^{r-1}(x)(1 - P(x))^{n-r} dP(x)$$

D.2 The Upper Bound

Since the upper bound is approximated by

$$T^u = \sum_{(i,j) \in \xi(k)} \delta_{ij}$$

we will derive the mean $s(k)$ of the sum of the first k order statistics for a sample of P^2 from the uniform distribution.

Let $p(x)$ be uniform in (0,1). Then μ_r reduces to

$$\mu_r \quad = \quad n \left(\begin{array}{c} n-1 \\ r-1 \end{array} \right) \int_0^1 x^r (1-x)^{n-r} dx = \frac{r}{n+1}$$

Let $s(k)$ denote the sum of the k largest order statistics. Since the mean distributes over sums, we have

$$\begin{aligned} E(s(k)) &= \sum_{r=n-k+1}^{n} \mu_r \\ &= \frac{1}{n+1}\left[\sum_{r=1}^{n} r - \sum_{r=1}^{n-k} r\right] \\ &= \frac{n(n+1)-(n-k)(n-k+1)}{2(n+1)} \end{aligned}$$

Taking $n = P^2$ and $k = 3P-3$, it follows

$$E(T^u) = \frac{P^2(P^2+1)-(P^2-3P+3)(P^2-3P+4)}{2(P^2+1)}$$

Since

$$\frac{n(n+1)-(n-k)(n-k+1)}{2(n+1)} = \frac{2nk-k^2+k}{2(n+1)}$$

for large n and $k \ll n$ it holds

$$\lim_{n\to\infty} \frac{2nk-k^2+k}{2(n+1)} = k$$

it follows that for large P it holds

$$E(T^u) = 3P-3$$

D.3 The Lower Bound

Since the lower bound is given by

$$T^l = \max\left[\max_{j=1\ldots M} \sum_{i=1}^{P} \delta_{ij}, \max_{i=1\ldots P} \sum_{j=1}^{M} \delta_{ij}\right]$$

we will derive the mean μ_n of the highest order statistic for a sample of $2P$ row and column sums of uniformly distributed elements.

While for general distributions, the moments of order statistics can not be characterized by simple expressions, a general upper bound on the extremal is given by

$$\mu_n \leq \mu + \frac{n-1}{\sqrt{2n-1}}\,\sigma$$

where μ and σ^2 are mean and variance, respectively. In the case of a symmetric distribution (like for the normal distribution) this limit can be narrowed down to

$$\mu_n \leq \mu + \frac{\sqrt{n}}{2}\,\sigma$$

While for small n this limit proves to be a relatively accurate predictor for μ_n, for normal distributions, an approximation [60] which yields better results for larger n is given by

$$\mu_n = \mu + \sqrt{2 \log 0.4n}\ \sigma$$

Since the mean and variance distribute over sums, for the sum of P uniformly distributed elements we have $\mu = P/2$ and $\sigma^2 = P/12$. Taking $n = 2P$, for not too small P it follows

$$E(T^l) \approx \frac{P}{2} + 0.4\sqrt{P \log P}$$

in which it is tacitly assumed that for larger P the row sums and column sums of Δ may be considered independent and normally distributed.

Appendix E

PAMELA Run-time Library

E.1 Introduction

This appendix describes the main features of the Pamela run-time library that has been used for the experiments described in this dissertation (Version 1.2). Although primarily intended as the run-time system for the future Pamela compiler, the library comprises a general-purpose, stand-alone performance simulation kernel. The library has been used for all the simulation experiments that are mentioned in this dissertation.

The Pamela run-time library is the latest development in a series of discrete-event simulation kernels, aimed to provide a concurrent, general-purpose performance simulation interface, based on the procedure-oriented ("P/V-style") paradigm [8]. The kernel is directly mapped onto light-weight threads, yet extending this concurrent layer with the notion of *virtual time*. Partially inspired by the need for this temporal enhancement, typically not present in most thread, task, or class packages, the choice for yet another in-house development is further motivated by:

- Simplicity
  Partially intended as an educational tool, the library should be extremely simple to use. This rules out, e.g., C^{++} class libraries, which assume a working knowledge of C^{++}. Furthermore, the interface should provide only a small, orthogonal kernel, directly corresponding to the user's basic needs and understanding of the concept of processes and semaphores. Featuring a straightforward C implementation, the Pamela run-time library is instantly usable for any average user who is familiar with basic concurrency.

- Portability
  While many thread libraries are hardware-specific, the choice for a separate layer provides the possibility to abstract from the actual platform, without significant performance loss. Mapped in terms of only a few macros, the Pamela run-time library is easily ported to different processors.

- Maintenance
  An in-house development, the Pamela run-time library is well-documented which eases maintenance and, especially, the development of functional enhancements.

- Accessibility
  Intended to deliver public domain software, the PAMELA project aims to minimize the use of third-party software, which is either costly, non-portable, or does not deliver sufficient functionality. Given the relatively small investment, the advantage of a proprietary kernel simply outweighs any alternative.

The development path towards the PAMELA run-time library has been marked by a number of historic events. Although an official implementation was never been released, the initial concept of a "P/V"-style performance simulation interface on top of a light-weight process kernel has been introduced in [48]. The first version was released as part of the "CPE" performance modeling technique [132]. Subsequently, its successor, an optimized version called the "VOP library" [124], has been in use, up to the release of the PAMELA run-time library. Although the VOP library offers the functionality needed for basic performance simulation, the implementation is not sufficiently engineered towards extensibility and portability over alternative light-weight processing packages. As the VOP library is hard-coded in terms of Sun's LWP library, the need for versions on other types of PCs and workstations, as well as the need for additional functionality, has inspired the development of its successor (the PAMELA run-time library includes advanced features, not present in its predecessor, such as an *interrupt* mechanism to implement e.g., timeouts). The PAMELA library runs on SunOs 4.x [147], as well as on individual nodes under the Amoeba operating system [16, 149]. Recently, a fully portable version has been released (Version 1.3 [109]) which has been successfully installed on various 80x86-based PCs (DOS/Linux) as well as HP workstations.

In the following, we describe some of the main features of the library. An elaborate description including the more specialized functions (like the timeout mechanism) can be found in [52, 109].

E.2 System Architecture

The library is defined in terms of the following two data types:

- Process type `pam_proc`
  With each process entity, a lightweight thread of control is associated which executes a user function, passed at creation. Each process has a local *time stamp* which is used to store the (global) time, either at which it has been suspended (in the past) or at which it has been scheduled to resume (in the future). The time stamps are used to implement the discrete event simulation mechanism, discussed later on.

- Semaphore type `pam_sema`
  With each semaphore is associated a *queue* in which processes are stored whose execution has to be suspended. Queued processes are *ordered* by increasing time stamp. Apart from storing processes, blocked on a user semaphore, the queue mechanism is also used to implement the *event list* (or ready list) which contains the runnable processes, scheduled to resume execution according to their time stamp.

At any time, only one process can be *running*. Its time stamp value is interpreted as the *current time*. All other processes are either *runnable*, i.e., scheduled for future execution at the time designated by their time stamp, of *blocked*, i.e., waiting on a semaphore until another process lifts this block by executing a `pam_V()` operation.

The concurrent execution of PAMELA threads revolves around a queue of runnable processes, implemented by a system semaphore, called `pam_sched`, i.e., the system scheduler queue (or event list), allocated upon library initialization[1]. A reference diagram is depicted by Fig. E.1, showing the scheduler queue `pam_sched`, along with one user semaphore queue `s`.

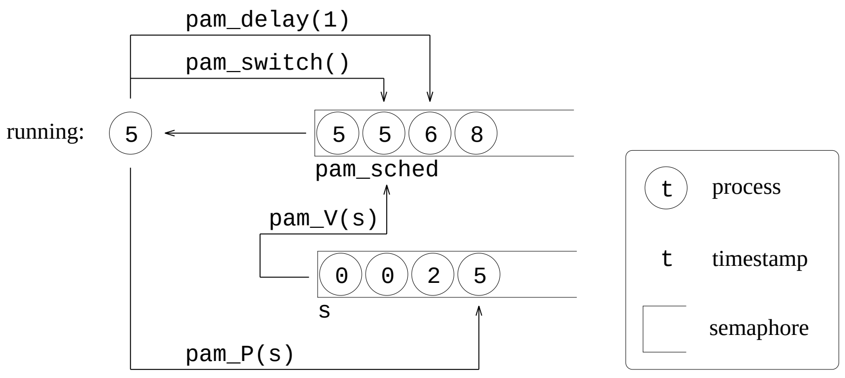


Figure E.1: System Operation

Like most thread libraries, PAMELA threads are scheduled non-preemptively. Thus, execution of the current process continues until it voluntarily relinquishes control. Apart from process deletion, this can only occur when the following three library routines are invoked (illustrated in the figure):

- `pam_delay()`
  The process stays runnable but is rescheduled (suspended) to be resumed after the designated interval.

- `pam_P()`
  If the semaphore credit value is zero, the process becomes blocked in the semaphore queue, only to be released through execution of a `pam_V()` call which reschedules the process as a runnable one with its time stamp made equal to the current time.

- `pam_switch()`
  By this explicit request for a context switch, the running process is traded for a runnable process which has an equivalent time stamp.

In all cases, the runnable process at the head of the `pam_sched` queue is selected to become the next running process, which (with exception of `pam_switch()`) results in an increment of the global time.

[1]The fact, that system scheduling is performed through a semaphore queue, symbolizes the *contention* of (logical) PAMELA threads for the single (physical) processing resource (i.e., the single CPU responsible for the interleaved execution of all thread programs).

E.3 Library Functions

In the following we briefly list the most relevant library functions. For brevity, the interrupt and exception handling functions have been omitted. In all cases the header file `pam.h` must be included.

- Initialization
  `void pam_init(char *name)`, the very first library function to be invoked, turns the caller into a Pamela thread, with user name `name` (see `pam_name()`). The function initializes various internal data structures, amongst which the scheduler (semaphore) ready queue `pam_sched`. Upon invocation the following conditions hold:

  - `pam_time()` = 0.0, i.e., zero global virtual time
  - `pam_level(0)`, i.e., library-level debugging off
  - `pam_trap(NULL)`, i.e., default debugging on exception

  `void pam_quit()` unconditionally returns control to the shell.

- Processes
  `pam_proc *pam_fork(char *name, void (*func)(), int args)` creates a new thread in suspended state which is enqueued in the ready queue `pam_sched` for eventual execution. A thread handle is returned for reference purpose (alternatively, this handle may be obtained by calling `pam_me()`). The `name` argument specifies a user-defined string which is intended for debugging purposes. `func` specifies the address of the actual code which is to be executed. As no arguments are supplied to this function, `args` specifies a user-defined integer which is intended to provide a very basic way of parameter passing using the `pam_args()` facility (see elsewhere). During the execution of `func()`, invoked by `p = pam_fork(name,func,args)`, the following equalities hold:

  - `pam_me()` = `p`
  - `pam_name(p)` = `name`
  - `pam_args(p)` = `args`
  - `pam_mode(p)` = `pam_sched`

  `void pam_kill(pam_proc *p)` deletes the thread specified by the handle `p`.

  `void pam_exit()` deletes the calling thread, i.e., `pam_kill(pam_me())`.

  `int pam_switch()` performs a context switch to another runnable thread which has an equal time stamp. In case of multiple candidates with equal time, the one at the head of `pam_sched` is chosen. Thus, `pam_switch()` operates in a round robin fashion. If no context switch could be performed, a 0 value if returned. In view of the non-preemptive scheduling discipline, in some situations, calls to `pam_switch()` are necessary in order to avoid starvation.

- Identification

  `pam_proc *pam_me()` returns a handle to the calling thread similar to the one originally returned by `pam_fork()`.

  `char *pam_name(pam_proc *p)` returns a pointer to the user-defined `name` string, originally passed through `pam_fork()`. The string is used by the debugging monitor for referencing purpose.

  `int pam_args(pam_proc *p)` returns a user-defined `args` string, originally passed through `pam_fork()`. The integer is not used by the library software and is intended to provide a basic parameter passing scheme (as shown in the examples presented earlier).

  `pam_sema *pam_mode(pam_proc *p)` returns the semaphore handle with respect to which the thread `p` is queued. The handle can take the following values

  – NULL, i.e., the thread is the caller, which is not blocked

  – `pam_sched`, i.e., the thread is runnable and scheduled

  – `s`, otherwise, when the thread is blocked on a user semaphore `s`.

- Timing

  `pam_proc *pam_delay(double delta)` suspends the caller's execution for `delta` time units. If other threads are scheduled at earlier times, the caller is enqueued in `pam_sched` to be resumed at the designated time. Otherwise, the caller stays in control and system time is simply incremented by `delta` time units. A process handle is passed on return. Possible values are

  – `pam_me()`, if the delay was normally executed

  – otherwise, if the thread was prematurely unblocked by a timeout interrupt (null process). The return value refers to this handle.

  `double pam_time()` returns the system time.

  `void pam_reset()` resets the system time to zero. In iterative simulation runs the value of `pam_time()` may become very large. This may cause resolution problems in which small increments (due to `pam_delay()`) are no longer properly accounted for. By periodically resetting system time to zero, these problems can be avoided.

- Semaphores

  `pam_sema *pam_alloc(char *name, int cred)` returns the handle of a newly created counting semaphore with a user-defined string `name` for debugging purposes. The semaphore is initialized to the value of `cred` (i.e., the number of initial `pam_P()` calls that will not block).

  `void pam_free(pam_sema *s)` deletes the semaphore specified by handle `s`.

  `pam_proc *pam_P(pam_sema *s)` decrements semaphore `s`. If zero on invocation, the caller is suspended and put in the queue associated with the semaphore. Once queued, the thread can only be resumed as a result of a `pam_V(s)` call by another thread, which handle is returned by `pam_P()`. Possible return values are

  - `pam_me()`, i.e., the caller was not blocked
  - otherwise, i.e., the handle of the process lifting the block, either by `pam_V()`, or through timeout interrupt (null process).

  `void pam_V(pam_sema *s)` increments semaphore `s` unless threads are queued for `s`, in which case one thread is dequeued and scheduled for future execution. The dequeue selection is according to a FIFO discipline, where, in case of multiple candidates with equal time, one is randomly chosen.

  `int pam_T(pam_sema *s)` returns the current credit of semaphore `s`. If smaller than 0, the value indicates the number of blocked processes. The call can be used to test if a subsequent `pam_P()` call would block.

- Debugging
  The `void pam_debug(char *prompt, int level)` break point facility provides a simple and effective means of selectively tracing the execution of Pamela threads. By invoking a debugging monitor, all existing threads and semaphores can be temporarily examined before program execution is continued (for a full description, see the manual). When invoked, the monitor prompts for command input using the string specified by `prompt` which typically comprises some tracing information. Whether the monitor is actually invoked depends on the value of `level`, which must be smaller than or equal to the system value set by the last `pam_level()` call. If `level` is greater than this value, `pam_debug()` has no effect. This feature enables applications to be instrumented with many `pam_debug()` calls in a selective tracing hierarchy. The Pamela run-time library, itself, is instrumented with `pam_debug()` calls which offers tracing at library call entry level. Currently, the following levels are implemented:

  0 no tracing (i.e., first level upwards, user-available)
  1 trace `pam_fork()`, `pam_exit()`, `pam_kill()`, `pam_switch()`, `pam_quit()`
  2 In addition to level 1, trace `pam_delay()`, `pam_P()`, `pam_V()`, `pam_T()`, `pam_reset()`
  3 In addition to level 2, trace `pam_alloc()`, `pam_free()`
  4 In addition to level 3, `pam_post()`, `pam_cancel()`
  5 In addition to level 4, trace internal context and queue operations (for maintenance only)

  `void pam_level(int level)` sets the system trace level to the value specified by `level`.

E.4 Programming Example

To give a quick impression of the ease of programming, we show a listing of a simple implementation of the MRM example, discussed throughout the dissertation (Example 3.4). Declarations have been omitted for brevity.

```
void    main()

        int     p;

        pam_init("main");
        server = pam_alloc("server",1);
        barrier = pam_alloc("barrier",0);

        /* create clients ("par (p = 1, P)")
         */
        for (p = 1; p <= P; p++)
                pam_fork("client",client,p);

        /* main blocks; clients unblock
         */
        for (p = 0; p < P; p++)
                pam_P(barrier);

        /* clients finished; main unblocks
         */
        printf("Cycle time %e\n",pam_time()/N);
        pam_quit();
}

void    client()
{
        int     i;

        /* cycle through network ("seq (i = 1, N)")
         */
        for (i = 1; i <= N; i++) {

                /* compute locally ("delay(\tau_l)")
                 */
                pam_delay(exprnd(10.0));

                /* request service ("use(s,\tau_s)")
                 */
                pam_P(server);
                pam_delay(exprnd(1.0));
                pam_V(server);
        }
        pam_V(barrier);
        pam_exit();
}
```

E.5 Debugging

The PAMELA run-time debugging monitor provides a basic mechanism to examine all existing threads and semaphores, through an interactive command line interpreter. The monitor is automatically invoked by the `pam_debug()` break point facility (see above). At invocation, the following prompt is generated

```
<time> (<name>) <trace> >
```

in which `<time>` is the current time, `<name>` is the name of the invoking thread (defined at creation), and `<trace>` is the trace message argument of the responsible `pam_debug()` call. For example, consider the following code, i.e.,

```
main()
{
        int        f();

        pam_init("main");
        s = pam_alloc("s",0);
        pam_fork("f",f,1);
        pam_delay(2.0);
        pam_debug("checkpoint 2",0);
        pam_V(s);
        ...
}

f()
{
        pam_delay(1.0);
        pam_debug("checkpoint 1",0);
        pam_P(s);
        ...
}
```

the following prompt is generated

```
1.000000e+00 (f) checkpoint 1 >
```

With the `i` command ("inspect") the current contents of all existing semaphores are listed in the following format, i.e.,

```
<semaphore>: <process> (<time>)  <process> (<time>)  ...
    ...
<semaphore>: <process> (<time>)  <process> (<time>)  ...
```

where `<semaphore>` denotes its name, `<process>` and `<time>` denote process name, and time stamp, respectively. In any case, the (scheduling) semaphore `pam_sched` is listed, which is created at initialization. If a semaphore queue is empty, the current semaphore count value is listed according to

Command	*Description*	*Library function*
`i`	Inspect semaphore queues	(explained above)
`c`	Continue program execution	(explained above)
`l <level>`	set tracing Level	`pam_level(<level>)`
`q`	Quit PAMELA system	`pam_quit()`
`d <itv>`	Delay current process	`pam_delay(<itv>)`
`k <name list>`	Kill list of processes	`pam_kill(<proc>)`
`f <name list>`	Fork another monitor	`pam_fork(<name>)`
`s`	Switch context	`pam_switch()`
`x`	eXit current process	`pam_exit()`
`a <name> <crd>`	Allocate semaphore	`pam_alloc(<name>,<crd>)`
`p <name>`	apply P operation	`pam_P(<sema>)`
`v <name>`	apply V operation	`pam_V(<sema>)`
`e <name list>`	frEe semaphores	`pam_free(<sema>)`
`o <name> <itv>`	pOst interrupt	`pam_post(<proc>,<itv>)`
`n`	caNcel interrupt	`pam_cancel(<proc>)`

Table E.1: Monitor command menu

```
<semaphore>: <+<value>>
```

In the above example, `i` would generate

```
pam_sched: main (2.000000e+00)
s: <+0>
```

With the `c` command ("continue"), the monitor is left, after which program execution continues. In the above example, the second breakpoint would generate

```
2.000000e+00 (main) checkpoint 2 >
```

Now, `i` would generate

```
pam_sched: <+0>
s: f (1.000000e+00)
```

corresponding to the fact that `f()` is blocked by `pam_P(s)`.

Apart from inspecting semaphores and continuing execution, the monitor offers commands to create and delete threads and semaphores, in order to experiment with the library on a command line interpreter basis. Table E.1 lists the commands which are supported in the current release. In the commands, a process or semaphore is referenced by the name, passed as argument to the creation command. Hence, `<proc>` and `<sema>` are the handles associated to the corresponding `<name>` arguments (in the `n` command, `<proc>` refers to the first association with the null process `pam_null`).

Due to the complexity of a generic scheme to supply a user-defined function pointer, the fork command simply forks a thread which executes an internal dummy function, which, in turn, calls `pam_debug("monitor",0)`, invoking another monitor instantiation, running in an infinite loop. Thus, multiple copies of the monitor may be active during the debugging session, each instantiation identifiable by the process name argument, passed at the creation command. The following session illustrates the educational value of the debugger. The program is given by

```
main()
{
        pam_init("main");
        pam_debug("monitor",0);
        pam_exit();
}
```

The following is a record of an interactive console session upon execution of the above program:

```
0.000000e+00 (main) monitor > f p1 p2
0.000000e+00 (main) monitor > i

    pam_sched: p1 (0.000000e+00)  p2 (0.000000e+00)

0.000000e+00 (main) monitor > a s1 0
0.000000e+00 (main) monitor > i

    pam_sched: p1 (0.000000e+00)  p2 (0.000000e+00)
    s1: <+0>

0.000000e+00 (main) monitor > d 1
0.000000e+00 (p1) monitor > i

    pam_sched: p2 (0.000000e+00)  main (1.000000e+00)
    s1: <+0>

0.000000e+00 (p1) monitor > p s1
0.000000e+00 (p2) monitor > i

    pam_sched: main (1.000000e+00)
    s1: p1 (0.000000e+00)

0.000000e+00 (p2) monitor > d 2
1.000000e+00 (main) monitor > i

    pam_sched: p2 (2.000000e+00)
    s1: p1 (0.000000e+00)

1.000000e+00 (main) monitor > p s1
2.000000e+00 (p2) monitor > i
```

```
    pam_sched: <+0>
    s1: p1 (0.000000e+00)  main (1.000000e+00)

2.000000e+00 (p2) monitor > d 1
3.000000e+00 (p2) monitor > v s1
3.000000e+00 (p2) monitor > i

    pam_sched: p1 (3.000000e+00)
    s1: main (1.000000e+00)

3.000000e+00 (p2) monitor > s
3.000000e+00 (p1) monitor > v s1
3.000000e+00 (p1) monitor > i

    pam_sched: p2 (3.000000e+00)  main (3.000000e+00)
    s1: <+0>

3.000000e+00 (p1) monitor > k p2
3.000000e+00 (p1) monitor > i

    pam_sched: main (3.000000e+00)
    s1: <+0>

3.000000e+00 (p1) monitor > c
3.000000e+00 (main) monitor > i

    pam_sched: <+0>
    s1: <+0>

3.000000e+00 (main) monitor > q
```

Bibliography

[1] V.S. Adve, *Analyzing the Behavior and Performance of Parallel Programs.* PhD thesis, University of Wisconsin, Madison, WI, Dec. 1993. Tech. Rep. #1201.

[2] V.S. Adve, A. Carle, E. Granston, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, J. Mellor-Crummey, S. Warren and C-W. Tseng, "Requirements for data-parallel programming environments," *IEEE Parallel and Distributed Technology*, July 1994, pp. 234–239.

[3] V.S. Adve and M.K. Vernon, "The influence of random delays on parallel execution times," in *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, May 1993, pp. 61–73.

[4] M. Ajmone Marsan, G. Balbo and G. Conte, "A class of Generalized Stochastic Petri Nets for the performance analysis of multiprocessor systems," *ACM Tr. on Comp. Syst.*, vol. 2, May 1984, pp. 93–122.

[5] M. Ajmone Marsan, G. Balbo and G. Conte, *Performance Models of Multiprocessor Systems.* MIT Press, 1986.

[6] M. Ajmone Marsan and G. Chiola, "On Petri nets with deterministic and exponentially distributed firing times," *Lecture Notes in Computer Science*, vol. 266, no. 24, 1987, pp. 132–145.

[7] F. Allen, M. Burke, R. Cytron, J. Ferrante, W. Hsieh and V. Sarkar, "A framework for determining useful parallelism," in *Proc. 1988 Int. Conf. Parallel Proc.*, IEEE, Aug. 1988, pp. 207–215.

[8] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no. 24, 1983, pp. 132–145.

[9] M. Annaratone, C. Pommerell and R. Rühl, "Interprocessor communication and performance in distributed-memory parallel processors," in *Proc. 16th Symp. on Comp. Archit.*, ACM, May 1989, pp. 315–324.

[10] D. Atapattu and D. Gannon, "Building analytical models into an interactive prediction tool," in *Proc. Supercomputing '89*, ACM, 1989, pp. 521–530.

[11] J. Baeten and P. Weijland, *Process Algebra.* Cambridge Univ. Press, 1990.

[12] D.H. Bailey, "Vector computer memory bank contention," *IEEE Transactions on Computers*, vol. C-36, Mar. 1987, pp. 293–298.

[13] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "A static performance estimator to guide data partioning decisions," in *Proc. 3rd ACM SIGPLAN Symposium on PPoPP*, Apr. 1991.

[14] Y. Bard, "Some extensions to multiclass queueing network analysis," in *Performance of Computer Systems* (M. Arato, A. Butrimenko and E. Gelenbe, eds.), North-Holland, 1979.

[15] F. Baskett, K.M. Chandy, R.R. Muntz and F.G. Palacios, "Open, closed, and mixed networks of queues with different classes of customers," *Journal of the ACM*, vol. 22, Apr. 1975, pp. 248–260.

[16] R. Bhoedjang and T. Ruhl, "PAMELA-Amoeba *macro package*." Personal Communication.

[17] L.N. Bhuyan, Q. Yang and D.P. Agrawal, "Performance of multiprocessor interconnection networks," *Computer*, Feb. 1989, pp. 25–37.

[18] L. Bomans and D. Roose, "Benchmarking the iPSC/2 hypercube multiprocessor," *Concurrency–Practice and Experience*, vol. 1, Sept. 1989, pp. 3–18.

[19] M. Bontekoe, "Generalizing interconnection network models," Tech. Rep. 1-68340-28(1994)26, Delft University of Technology, Delft, The Netherlands, Sept. 1994.

[20] I.Y. Bucher and D.A. Calahan, "Access conflicts in multiprocessor memories queueing models and simulation studies," in *Proc. 4th ACM Int. Conf. on Supercomputing*, 1990, pp. 428–438.

[21] I.Y. Bucher and M.L. Simmons, "A close look at vector performance of register-to-register vector computers and a new model," *Performance Evaluation Review*, vol. 15, no. 1, 1987, pp. 39–45.

[22] P. Buchholz, "Hierarchical Markovian models: Symmetries and reduction," in *Proc. 6th Int. Conf. Modelling Techniques and Tools for Comp. Perf. Eval.*, Edinburgh, Sept. 1992.

[23] D.A. Calahan, "An analysis of vector startup access delays," *IEEE Transactions on Computers*, vol. 37, Sept. 1988, pp. 1134–1137.

[24] D.A. Calahan, "Characterization of memory conflict loading on the CRAY-2," in *Proc. 1988 Int. Conf. Parallel Proc.*, IEEE, Aug. 1988, pp. 299–302.

[25] D. Callahan, K.D. Cooper, R.T. Hood, K. Kennedy and L. Torczon, "ParaScope: A parallel programming environment," *Int. Journ. of Supercomp. Applic.*, vol. 4, no. 2, 1988, pp. 84–99.

[26] D. Callahan and K. Kennedy, "Compiling programs for distributed-memory multiprocessors," *The Journal of Supercomputing*, no. 2, 1988, pp. 151–169.

[27] R. Candlin and J. Phillips, "A statistical study of factors that affect the performance of a class of parallel programs on a MIMD computer," in *Proc. Int'l. Conf. on Decentralized and Distributed Systems (IFIP Transactions A-39)*, Palma, 1993.

[28] T. Cheung and J.E. Smith, "A simulation study of the CRAY X-MP memory system," *IEEE Transactions on Computers*, vol. C-35, July 1986, pp. 631–622.

[29] M.J. Clement and M.J. Quinn, "Multivariate statistical techniques for parallel performance prediction," in *Proc. 28th Hawaii Int. Conf. on System Sciences, Vol. II*, IEEE, Jan. 1995, pp. 446–455.

[30] M.E. Crovella and T.J. Leblanc, "The search for lost cycles: A new approach to parallel program performance evaluation," in *Proc. Supercomputing '94*, ACM, 1994, pp. 600–609.

[31] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken, "LogP: Towards a realistic model of parallel computation," in *Proc. 4th ACM SIGPLAN Symposium on PPoPP*, May 1993, pp. 1–12.

[32] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *Proc. 1986 Int. Conf. Parallel Proc.*, IEEE, Aug. 1986, pp. 836–844.

[33] O.J. Dahl and K. Nygaard, "Simula - an ALGOL-based simulation language," *Communications of the ACM*, vol. 9, no. 9, 1966, pp. 671–678.

[34] F. Darema, D.A. George, V.A. Norton and G.F. Pfister, "A single-program-multiple-data computation model for EPEX/FORTRAN," *Parallel Computing*, vol. 7, 1988, pp. 11–24.

[35] S. Dasgupta, "A hierarchical taxonomic system for computer architectures," *Computer*, Mar. 1990, pp. 64–74.

[36] H.A. David, *Order Statistics.* John Wiley & Sons, 1970.

[37] E.W. Dijkstra, "Cooperating sequential processes," in *Programming Languages* (F. Geunys, ed.), Academic Press, 1968, pp. 43–112.

[38] A.N. Dunlop, A.J.G. Hey, D.A. Nicole and D.J. Pritchard, "Performance estimating for parallel performance optimisation," *Supercomputer*, vol. 66, 1995.

[39] D.L. Eager, J. Zahorjan and E.D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, vol. 38, Mar. 1989, pp. 408–423.

[40] W. Ewinger, O. Haan, E. Haupenthal and C. Siemers, "Modelling and measurement of memory access in SIEMENS VP supercomputers," *Parallel Computing*, vol. 11, 1989, pp. 361–365.

[41] T. Fahringer and H.P. Zima, "A static parameter-based performance prediction tool for parallel programs," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 207–219.

[42] T. Feng, "A survey of interconnection networks," *Computer*, Dec. 1981, pp. 12–27.

[43] A. Ferscha and A.D. Maloney, "Performance-oriented development of irregular, unstructured and unbalanced parallel applications in the N-MAP environment," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 977) (H. Beilner and F. Bause, eds.), Berlin, Springer-Verlag, Sept. 1995, pp. 340–356.

[44] P. Flajolet and J-M. Steyart, "A compexity calculus for recursive tree algorithms," *Math. Systems Theory*, vol. 19, 1987, pp. 301–331.

[45] S. Fortune and J. Wyllie, "Parallelism in random access machines," in *Proc. 10th Annual Symp. on Theory of Comput.*, 1978, pp. 114–118.

[46] K. Gallivan, W. Jalby, A. Malony and H. Wijshoff, "Performance prediction of loop constructs on multiprocessor hierarchical-memory systems," in *Proc. 3rd ACM Int. Conf. on Supercomputing*, 1989, pp. 433–442.

[47] E. Gelenbe, E. Montagne, R. Suros and C.M. Woodside, "Performance of block-structured parallel programs," in *Parallel Algorithms and Architectures* (M. Cosnard *et al.*, eds.), North-Holland, 1986, pp. 127–138.

[48] A.J.C. van Gemund, "Research notes on processor modeling," Tech. Rep. 90 ITI 2031, TNO Institute of Applied Computer Science, Delft, The Netherlands, Dec. 1990.

[49] A.J.C. van Gemund, "Performance prediction of parallel processing systems: The PAMELA methodology," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 318–327.

[50] A.J.C. van Gemund, "Compile-time performance prediction with PAMELA," in *Proc. 4th Int. Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993, pp. 428–435.

[51] A.J.C. van Gemund, "Compiling performance models from parallel programs," in *Proc. 8th ACM Int. Conf. on Supercomputing*, Manchester, July 1994, pp. 303–312.

[52] A.J.C. van Gemund, "The PAMELA run-time library version 1.0," Tech. Rep. 1-68340-44(1994)03, Delft University of Technology, Delft, The Netherlands, Apr. 1994.

[53] A.J.C. van Gemund, "Predicting contention in distributed-memory machines," in *Proc. Second Workshop on Automatic Data Layout and Performance Prediction (tech. rep. CRPC-TR95548)*, Rice University, Houston, Apr. 1995.

[54] A.J.C. van Gemund, “On the accuracy of compile-time performance prediction,” in *Proc. Fifth Workshop on Compilers for Parallel Computers*, Malaga, June 1995, pp. 157–166.

[55] A.J.C. van Gemund, “Compile-time performance prediction of parallel systems,” in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 977) (H. Beilner and F. Bause, eds.), Berlin, Springer-Verlag, Sept. 1995, pp. 299–313.

[56] A. Gibbons, *Algorithmic Graph Theory.* Cambridge University Press, 1988.

[57] N. Götz, U. Herzog and M. Rettelbach, “Multiprocessor and distributed system design: The integration of functional specification and performance analysis using stochastic process algebras,” in *Performance Evaluation of Computer and Communication Systems (Combined Tutorial Proceedings SIGMETRICS’93 and PERFORMANCE’93, LNCS 729)* (L. Donatiello and R. Nelson, eds.), Springer, 1993.

[58] R.L. Graham, “Bounds on multiprocessing timing anomalies,” *SIAM J. Appl. Math.*, vol. 17, no. 2, 1969, pp. 416–429.

[59] V.A. Guarna, D. Gannon, D. Jablonowski, A.D. Maloney and Y. Gaur, “Faust: An integrated environment for parallel programming,” *Software*, July 1989, pp. 20–27.

[60] E.J. Gumbel, “Statistical theory of extreme values (main results),” in *Contributions to Order Statistics* (A.E. Sarhan and B.G. Greenberg, eds.), John Wiley & Sons, 1962, pp. 56–93.

[61] M. Gupta and P. Banerjee, “Compile-time estimation of communication costs of programs,” in *Proc. Second Workshop on Automatic Data Layout and Performance Prediction (tech. rep. CRPC-TR95548)*, Rice University, Houston, Apr. 1995.

[62] B. van Halderen, “A tool for application performance prediction,” Master’s thesis, University of Amsterdam, Amsterdam, Sept. 1995.

[63] F. Hartleb and V. Mertsiotakis, “Bounds for the mean runtime of parallel programs,” in *Proc. 6th Int. Conf. Modelling Techniques and Tools for Comp. Perf. Eval.*, Edinburgh, Sept. 1992, pp. 197–210.

[64] K. Harzallah and K.M. Sevcik, “Predicting application behavior in large-scale shared-memory multiprocessors,” in *Proc. Supercomputing ’95*, ACM, 1995.

[65] P. Heidelberger and K.S. Trivedi, “Analytic queueing models for programs with internal concurrency,” *IEEE Transactions on Computers*, vol. 32, Jan. 1983, pp. 73–82.

[66] T. Hickey and J. Cohen, “Automating program analysis,” *Journal of the ACM*, vol. 35, Jan. 1988, pp. 185–219.

[67] J. Hillston, *A Compositional Approach to Performance Modelling.* PhD thesis, University of Edinburgh, 1994.

[68] S. Hiranandani, K. Kennedy and C-W. Tseng, "Compiling FORTRAN-D for MIMD distributed-memory machines," *Communications of the ACM*, vol. 35, Aug. 1992, pp. 66–80.

[69] C.A.R. Hoare, *Communicating Sequential Processes.* Prentice-Hall, 1985.

[70] R.W. Hockney, "Performance parameters and benchmarking of supercomputers," *Parallel Computing*, vol. 17, 1991, pp. 1111–1130.

[71] R.W. Hockney and I.J. Curington, "($f_{1/2}$): A parameter to characterize memory and communication bottlenecks," *Parallel Computing*, vol. 10, 1989, pp. 277–286.

[72] R.W. Hockney and C.R. Jesshope, *Parallel Computers 2 - Architecture, Programming and Algorithms.* Adam Hilger, 1981.

[73] J. Hollingsworth and B.P. Miller, "Parallel program performance metrics: A comparison and validation," in *Proc. Supercomputing '92*, ACM, 1992.

[74] INMOS Limited, *The Transputer Databook*, 1989. Doc. No. 72 TRN 203 01.

[75] K.E. Iverson, *A Programming Language.* Wiley, 1962.

[76] K.K. Jain and V. Rajaraman, "Lower and upper bounds on time for multiprocessor optimal schedules," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, Aug. 1994, pp. 879–886.

[77] K. Jensen, "Coloured Petri nets: A high level language for system design and analysis," in *High-level Petri Nets: Theory and Application* (K. Jensen and G. Rozenberg, eds.), Springer-Verlag, 1991, pp. 44–122.

[78] H. Jonkers, *Performance Analysis of Parallel Systems: A Hybrid Approach.* PhD thesis, Delft University of Technology, The Netherlands, Oct. 1995.

[79] H. Jonkers, A.J.C. van Gemund and G.L. Reijns, "Efficient performance evaluation of parallel systems," in *Massively Parallel Processing Applications and Development* (L. Dekker *et al.*, eds.), Delft, North-Holland, 1994, pp. 389–396.

[80] H.F. Jordan, "The Force," in *The Characteristics of Parallel Algorithms* (L.H. Jamieson, D. Gannon and R.J. Douglas, eds.), MIT Press, 1987, pp. 395–436.

[81] A. Kapelnikov, R.R. Muntz and M.D. Ercegovac, "A modeling methodology for the analysis of concurrent systems and computations," *Journal of Parallel and Distributed Computing*, vol. 6, 1989, pp. 568–597.

[82] A.H. Karp, "Programming for parallelism," *Computer*, May 1987, pp. 43–57.

[83] A.H. Karp and R.G. Babb, "A comparison of 12 parallel FORTRAN dialects," *Software*, Sept. 1988, pp. 52–67.

[84] C.W. Kessler, *Automatische Parallelisierung numerischer Programme durch Mustererkennung.* PhD thesis, Universität Saarbrücken, Germany, 1994.

[85] L. Kleinrock, *Queueing Systems: Vol. 2, Computer Applications.* Wiley, 1976.

[86] C. Koelbel, D. Loveman, R. Schreiber, G. Steele Jr. and M. Zosel, *The High-Performance Fortran Handbook.* MIT Press, 1994.

[87] C. Koelbel and P. Mehrotra, "Compiling global name-space parallel loops for distributed execution," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, Oct. 1991, pp. 440–451.

[88] Ulrich Kremer, "NP-completeness of dynamic remapping," in *Proc. 4th Int. Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, Dec. 1993, pp. 135–141.

[89] W. Kreutzer, *System simulation, programming styles and languages.* Addison-Wesley, 1986.

[90] C.P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *IEEE Transactions on Software Engineering*, vol. 11, Oct. 1985, pp. 1001–1016.

[91] S.S. Lavenberg, *Computer Performance Modeling Handbook.* Academic Press, 1983. ISBN 0-12-438720-9.

[92] D.H. Lawrie, "Access and alignment of data in an array processor," *IEEE Transactions on Computers*, vol. 24, Dec. 1975, pp. 1145–1155.

[93] E.D. Lazowska *et al.*, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, 1984.

[94] K.Y. Lee, W. Abu-Sufah and D.J. Kuck, "On modeling performance degradation due to data movement in vector machines," in *Proc. 1984 Int. Conf. Parallel Proc.*, IEEE, Aug. 1984, pp. 269–277.

[95] B.P. Lester, "A system for computing the speedup of parallel programs," in *Proc. 1986 Int. Conf. Parallel Proc.*, IEEE, Aug. 1986, pp. 145–152.

[96] H.X. Lin and H.J. Sips, "Parallel direct solution of large sparse systems in finite element computations," in *Proc. 7th ACM Int. Conf. on Supercomputing*, Tokyo, July 1993, pp. 261–270.

[97] M. Maekawa, A.E. Oldehoeft and R.R. Oldehoeft, *Operating Systems, Advanced Concepts.* Benjamin/Cummings, Ca., 1987.

[98] V.W. Mak and S.F. Lundstrom, "Predicting performance of parallel computations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, July 1990, pp. 257–270.

[99] A.D. Maloney, V. Mertsiotakis and A. Quick, "Automatic scalability analysis of parallel programs based on modeling techniques," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 794) (G. Haring and G. Kotsis, eds.), Berlin, Springer-Verlag, May 1994, pp. 139–158.

[100] O.A. McBryan, "An overview of message-passing environments," *Parallel Computing*, vol. 20, 1994, pp. 417–444.

[101] P. Mehra, C.H. Schulbach and J.C. Yan, "A comparison of two model-based performance-prediction techniques for message-passing parallel programs," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, Nashville, May 1994, pp. 181–189.

[102] C.L. Mendes, J-C. Wang and D.A. Reed, "Automatic performance prediction and scalability analysis for data parallel programs," in *Proc. Second Workshop on Automatic Data Layout and Performance Prediction (tech. rep. CRPC-TR95548)*, Rice University, Houston, Apr. 1995.

[103] J.F. Meyer, A. Movaghar and W.H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proc. of the Int. Conf. on Timed Petri nets*, Torino, July 1985, pp. 106–115.

[104] M.K. Molloy, "Performance analysis using stochastic Petri nets," *IEEE Transactions on Computers*, vol. C-31, Sept. 1982, pp. 913–917.

[105] H.L. Muller, *Simulating Computer Architectures.* PhD thesis, Department of Computer Systems, University of Amsterdam, Amsterdam, The Netherlands, 1993.

[106] D. Müller-Wichards, "Performance estimates for applications: An algebraic framework," *Parallel Computing*, vol. 9, Dec. 1988, pp. 77–106.

[107] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, Apr. 1989, pp. 541–580.

[108] K.M. Nichols and J.T. Edmark, "Modeling multicomputer systems with PARET," *Computer*, May 1988, pp. 39–48.

[109] M. Nijweide, "The Pamela run-time library version 1.3: Extensions and applications," Tech. Rep. 1-68340-27(1995)06, Delft University of Technology, Delft, The Netherlands, June 1995.

[110] A. Norton and G.F. Pfister, "A methodology for predicting multiprocessor performance," in *Proc. 1985 Int. Conf. Parallel Proc.*, IEEE, Aug. 1985, pp. 772–781.

[111] S.F. Nugent, "The iPSC/2 direct-connect technology," in *Proc. 3rd Hypercube Conference*, ACM, 1988.

[112] W. Oed and O. Lange, "Modelling, measurement and simulation of memory interference in the CRAY X-MP," *Parallel Computing*, vol. 3, 1986, pp. 343–358.

[113] E.M.R.M. Paalvast, *Programming for Parallelism and Compiling for Efficiency.* PhD thesis, Delft University of Technology, Delft, The Netherlands, 1992.

[114] K. Padmanabhan, "Cube structures for multiprocessors," *Communications of the ACM*, vol. 33, Jan. 1990, pp. 43–52.

[115] Parsytec Computer GmbH, *Parix release 1.2 software documentation*, Mar. 1993.

[116] S. Patil, "Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes," tech. rep., MIT, Feb. 1971.

[117] J.L. Peterson, *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[118] C.A. Petri, *Kommunikation mit Automaten.* PhD thesis, Institut für Instrumentelle Mathematik, Bonn, Germany, 1962.

[119] C.A. Petri, "Communication with automata," Tech. Rep. RADC-TR-68-305, Griffiss Air Force Base, New York, 1966. (Translation of [118]).

[120] A. Pimentel, J. van Brummen, T. Papathanassianis, P.M.A. Sloot and L.O. Hertzberger, "Mermaid: Modelling and evaluation research in Mimd ArchItecture Design," in *Proc. HPCN Conf. (LNCS)*, Springer, 1995, pp. 335–340.

[121] B. Plateau, J.M. Fourneau and K.H. Lee, "PEPS: A package for solving complex Markov models of parallel systems," in *Proc. 4th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, Palma, Mallorca, Sept. 1988, pp. 341–360.

[122] C.D. Polychronopoulos and U. Banerjee, "Speedup bounds and processor allocation for parallel programs on multiprocessors," in *Proc. 1986 Int. Conf. Parallel Proc.*, IEEE, Aug. 1986, pp. 961–968.

[123] C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung and D. Schouten, "Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors," in *Proc. 1989 Int. Conf. Parallel Proc.*, IEEE, Aug. 1989, pp. II:39–48.

[124] R. Pulleman, "Simulation of VOP models," Tech. Rep. 92 TPD-ZP 938, TNO Institute for Applied Physics, Delft, The Netherlands, Sept. 1992.

[125] B. Qin, H.A. Sholl and R.A. Ammar, "Micro time cost analysis of parallel computations," *IEEE Transactions on Computers*, vol. 40, May 1991, pp. 613–628.

[126] C.V. Ramamoorthy, "Discrete Markov analysis of computer programs," in *20th ACM National Conference*, Cleveland, ACM, 1965, pp. 386–391.

[127] C.V. Ramamoorthy and G.S. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," *IEEE Transactions on Software Engineering*, vol. 6, Sept. 1980, pp. 440–449.

[128] S.K. Reinhardt, M.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis and D.A. Wood, "The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers," in *Proc. 1993 ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, May 1993, pp. 48–60.

[129] M. Reiser and S.S. Lavenberg, "Mean value analysis of closed multichain queueing networks," *Journal of the ACM*, vol. 27, Apr. 1980, pp. 313–322.

[130] W. Reisig, *Petri Nets*. Springer Verlag, 1985.

[131] J.T. Robinson, "Some analysis techniques for asynchronous multiprocessor algorithms," *IEEE Transactions on Software Engineering*, vol. 5, Jan. 1979, pp. 24–31.

[132] M.R.T. Roest, "The CPE modelling technique," Tech. Rep. 91 ITI 1675, TNO Institute of Applied Computer Science, Delft, The Netherlands, Nov. 1991.

[133] J.F. de Ronde, A.W. van Halderen, A. de Mes, M. Beemster and P.M.A. Sloot, "Automatic performance estimation of SPMD programs on MPP," in *Massively Parallel Processing Applications and Development* (L. Dekker *et al.*, eds.), Delft, North-Holland, 1994, pp. 381–388.

[134] R.A. Sahner and K.S. Trivedi, "SPADE: A tool for performance and reliability evaluation," in *Modelling Techniques and Tools for Performance Analysis '85* (N. Abu El Ata, ed.), Elsevier Science Publishers, 1986, pp. 147–163.

[135] W.H. Sanders, W.D. Obal, M.A. Qureshi and F.K. Widjanarko, "The *UltraSan* modeling environment," *Performance Evaluation Journal, special issue on Performance Modeling Tools*, 1995.

[136] V. Sarkar, "Determining average program execution times and their variance," in *Proc. 1989 ACM SIGPLAN Conf. on Prog. Lang. Des. and Impl.*, 1989, pp. 298–312.

[137] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[138] P. Schweitzer, "Approximate analysis of multiclass closed networks of queues," in *Proc. of International Conf. on Control and Optimization*, Amsterdam, 1979.

[139] H. Schwetman, "Object-oriented simulation modeling with C++/CSIM17," in *Proc. 1995 Winter Simulation Conference*, 1995.

[140] A.C. Shaw, "Deterministic timing schema for parallel programs," in *Proc. 5th Int. Parallel Processing Symposium*, IEEE, 1991, pp. 56–63.

[141] M. Siegle, "Using structured modelling for efficient performance prediction of parallel systems," in *Parallel Computing: Trends and Applications* (G.R. Joubert *et al.*, eds.), North-Holland, 1994, pp. 453–460.

[142] A. Sivasubramaniam, A. Singla, U. Ramachandran and H. Venkateswaran, "An approach to scalability of shared memory parallel systems," in *Proc. ACM SIGMETRICS Conf. on Measurement and Modelling of Computer Systems*, Nashville, May 1994, pp. 171–180.

[143] K. So, A.S. Bolmarcich, F. Darema and V.A. Norton, "A speedup analyzer for parallel programs," in *Proc. 1987 Int. Conf. Parallel Proc.*, IEEE, Aug. 1987, pp. 653–661.

[144] F. Sötz, "A method for performance prediction of parallel programs," in *Proc. CONPAR 90-VAPP IV (LNCS 457)* (H. Burkhart, ed.), Springer-Verlag, 1990, pp. 98–107.

[145] P. Stenström, "Reducing contention in shared-memory multiprocessors," *Computer*, Nov. 1988, pp. 26–37.

[146] B. Stramm and F. Berman, "Predicting the performance of large programs on scalable multicomputers," in *Scalable HPC Conference*, Apr. 1992, pp. 22–29.

[147] Sun Microsystems, *SunOS 4.1.3 Programming Utilities and Libraries*, Mar. 1990. Part Number 800-3847-10.

[148] V. Sundaram, "PVM: A framework for parallel distributed computing," *Concurrency–Practice and Experience*, vol. 2, Dec. 1990, pp. 315–339.

[149] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, A.J. Jansen and G. van Rossum, "Experiences with the Amoeba distributed operating system," *Communications ACM*, vol. 33, Dec. 1990, pp. 46–63.

[150] A. Thomasian and P.F. Bay, "Analytic queueing network models for parallel processing task systems," *IEEE Transactions on Computers*, vol. 35, Dec. 1986, pp. 1045–1054.

[151] L. Valiant, "A bridging model for parallel computation," *Communications of the ACM*, vol. 33, Aug. 1990, pp. 103–111.

[152] H. Wabnig and G. Haring, "PAPS - the parallel program performance prediction toolset," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 794) (G. Haring and G. Kotsis, eds.), Berlin, Springer-Verlag, May 1994.

[153] T.A. Wagner, V. Maverick, S.L. Graham and M.A. Harrison, "Accurate static estimators for program optimization," *SIGPLAN Notices*, June 1994, pp. 85–96.

[154] D. Walker, "The design of a standard message passing interface for distributed memory concurrent computers," *Parallel Computing*, vol. 20, 1994.

[155] K-Y. Wang, "Intelligent program optimization and parallelization for parallel computers," Tech. Rep. CSD-TR 91-030, Purdue University Apr. 1991.

[156] K-Y. Wang, "Precise compile-time performance prediction for superscalar-based computers," in *Proc. ACM SIGPLAN PLDI'94*, Orlando, June 1994, pp. 73–84.

[157] B. Wegbreit, "Mechanical program analysis," *Communications of the ACM*, vol. 18, Sept. 1975, pp. 528–539.

[158] P. Woodbury, A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Barttlet and Z. Aral, "Shared memory multiprocessors: The right approach to parallel processing," in *Proc. COMPCON Spring '89*, IEEE, 1989, pp. 72–80.

[159] N. Yazici-Pekergin and J-M. Vincent, "Stochastic bounds on execution times of parallel programs," *IEEE Transactions on Software Engineering*, vol. 17, Oct. 1991, pp. 1005–1012.

[160] J. Zahorjan *et al.*, "Balanced job bound analysis of queueing networks," *Communications of the ACM*, vol. 25, Feb. 1982, pp. 134–141.

[161] H. Zima, P. Brezany, B. Chapman, P. Mehrotra and A. Schwald, "Vienna Fortran - a language specification, version 1.1," Tech. Rep. Interim Report 21, ICASE, NASA Langley Research Center, Mar. 1992.

Samenvatting

Prestatiemodellering speelt een fundamentele rol in het ontwerp van zowel applicaties als computers. Dit geldt met name voor parallelle systemen waar de prestatie een primaire rol speelt. Waar de prestatiemodellering van sequentiële computers reeds aanzienlijke problemen oproept, zijn de problemen bij parallelle systemen zo mogelijk nog fundamenteler. Dit is in essentie het gevolg van de grote rol die processynchronisatie speelt in parallelle verwerking. Naast de inherente overhead ten gevolge van het parallelliseren, zijn het, met name voor slecht ontworpen systemen, de synchronisatietijden die tot een enorme prestatieverlies kunnen leiden.

In parallelle systemen kan men zowel een statische vorm als een dynamische vorm van processynchronisatie onderscheiden. De statische vorm, genaamd conditie-synchronisatie, heeft betrekking op precedentierelaties tussen taken die op grond van de parallellisatie vooraf zijn bepaald. De andere vorm, genaamd wederzijdse uitsluiting, betreft de dynamische toewijzing van procesvolgorde als gevolg van de beperkte beschikbaarheid van software of hardware middelen. Hoewel de aanwezigheid van conditiesynchronisatie reeds aanzienlijke prestatieanalysekosten met zich mee brengen kunnen de kosten gemoeid met de analyse van wederzijdse uitsluiting nog veel hoger liggen vanwege het inherente non-determinisme van deze synchronisatievorm.

Er bestaat een grote verscheidenheid aan methoden voor de prestatiemodellering van parallelle systemen, waarbij elke methode een specifieke afweging vertegenwoordigt tussen de nauwkeurigheid van de prestatieanalyse en de rekenkosten die hiermee gemoeid zijn. Enerzijds bestaan er technieken, gebaseerd op representatievormen zoals Petri-netwerken, die een dermate hoge modelleerkracht bieden dat elke vorm van synchronisatie nauwkeurig kan worden uitgedrukt, maar waarvan de prijs een exponentiële rekencomplexiteit in de probleemgrootte is. Anderszijds bestaan er goedkopere technieken, zoals die gebaseerd op simpele taakgraaf representaties, die alleen conditiesynchronisaties verdisconteren. Hoewel de analyse complexiteit slechts lineair is leidt de verwaarlozing van de verliezen ten gevolge van wederzijdse uitsluiting tot een zeer beperkte nauwkeurigheid.

Dit proefschrift beschrijft een nieuwe aanpak voor de prestatiemodellering van parallelle systemen. Vergelijkbaar met sommige bestaande aanpakken richt de methode zich met name op de beginfase van het ontwerpproces van parallelle systemen waar de nadruk meer ligt op minimale analysekosten dan op een hoge nauwkeurigheid. In afwijking van vergelijkbare goedkope methoden is de nauwkeurigheid echter sterk verbeterd door naast de analyse van conditiesynchronisatie een benadering van de vertragingstijden als gevolg van wederzijdse uitsluiting te introduceren zonder de gunstige rekencomplexiteit op te offeren. Tevens levert de analysetechniek expliciete analytische modellen op zodat programma- en machineparameters in symbolische vorm behouden blijven in het

model. Behalve de lage evaluatiekosten wordt op deze manier bereikt dat parameterstudies of mogelijk geautomatiseerde parameteroptimalisaties kunnen worden uitgevoerd zonder dat opnieuw dient te worden gemodelleerd. Naast de lage rekenkosten is ook dit een belangrijke voorwaarde voor een optimale ontwerp doelmatigheid.

De aanpak is gebaseerd op het gebruik van een nieuw simulatieformalisme, genaamd Pamela (PerformAnce ModEling LAnguage). Hoewel de taal synchronisatieconstructies bevat teneinde *a priori* geen onnodige benaderingen te introduceren, bevat Pamela ook gestructureerde operatoren, met name voor de beschrijving van wederzijdse uitsluiting. In combinatie met het gebruik van een materiaal-georiënteerd modelleerparadigma kan zodoende belangrijke informatie met betrekking tot de aanwezige synchronisatiepatronen worden behouden. Als gevolg hiervan kan Pamela, naast simulatie, worden gebruikt als een brontaal ten behoeve van een automatische vertaaltechniek die een expliciet, analytisch prestatiemodel oplevert. Het model benadert de prestatieverliezen ten gevolge van wederzijdse uitsluiting in de vorm van een ondergrens aan de executietijd. Het nieuwe van de aanpak is de integratie van een taal, een materiaal-georiënteerde paradigma, en een vertaaltechniek binnen één modelleermethodiek.

Terwijl hoofstuk 1 ingaat op de probleemanalyse en de doelstellingen van het onderzoek, geeft hoofdstuk 2 een overzicht van het vele werk dat reeds is verricht op het gebied van de prestatiemodellering van parallelle systemen, teneinde de aanpak in een juist kader te plaatsen. Het werk dat aan de orde komt beslaat methodieken gebaseerd op representatievormen zoals taakgrafen, wachtrijnetwerken, Petri-netwerken, simulatietalen en procesalgebra. In het overzicht wordt een eigen categorisatietechniek gehanteerd teneinde de grote variëteit binnen één raamwerk te kunnen plaatsen.

Hoofdstuk 3 presenteert Pamela, bestaande uit de modelleertaal en de bijbehorende analysetechniek. Er wordt aangetoond dat de expliciete en gestructureerde wijze waarop de materiaal-georiënteerde modelleermethode beide synchronisatievormen tot uitdrukking brengt grote voordelen biedt met betrekking tot de analyseerbaarheid van het model. Naast een beschrijving van de analysetechniek worden een aantal kenmerkende voorbeelden behandeld.

Hoofdstuk 4 beschrijft de principes hoe parallelle computersystemen kunnen worden gemodelleerd met behulp van Pamela. De methodiek die gehanteerd wordt bij de modellering van zowel gemeenschappelijk- als gedistribueerd-geheugensystemen wordt beschreven aan de hand van vele voorbeelden. Er wordt aangetoond dat, ondanks de restricties in het modelleren van synchronisaties, de beschrijving van de essentiële prestatieaspecten voldoende is, gegeven de benaderende analysetechniek.

Hoodstuk 5 presenteert een aantal gevallen waarin diverse kanten van de Pamela methodiek worden belicht. De onderwerpen die aan bod komen zijn onder meer voorbeelden hoe parallelle applicatiemodellen worden vertaald naar analytische modellen, de modellering van een data flow applicatie op een gedistribueerd-geheugensysteem inclusief een vergelijking van de modelresultaten met praktijkmetingen, een beschouwing van de relatie tussen de analytische techniek en simulatietechnieken, en voorbeelden hoe de Pamela methodiek wordt gebruikt ten behoeve van programma-optimalisatie, een van de uiteindelijke doelstellingen van de methodiek.

Hoofdstuk 6 staat opnieuw stil bij de benaderende analysetechniek en gaat in op de nauwkeurigheid van de analytische techniek vergeleken met simulatie. Aan de hand van een uitgebreide studie wordt aangetoond dat de ondergrensbenadering een goede schatting oplevert. Tevens wordt aangetoond dat voor systemen met willekeurige volgordepatronen van wederzijdse uitsluiting de gemiddelde relatieve afwijking als gevolg hiervan in het ergste geval nog binnen 50 % ligt, onafhankelijk van de grootte van de systeemparameters. Gezien de hoge mate van parametrisering van de modellen vormt deze robuustheid de uiteindelijke rechtvaardiging van de nieuwe aanpak.

Tot slot biedt hoofdstuk 7 een terugblik op het onderzoek en geeft een aantal aanbevelingen voor toekomstige verbeteringen.

Curriculum Vitae

Arjan J.C. van Gemund was born in Eindhoven, the Netherlands on September 4, 1955. He received a BSc degree in Physics in 1981, and an MSc degree in Computer Science in 1989 from Delft University of Technology.

In 1981 he joined the R & D organization of a Dutch multinational company as an Electrical Engineer and Systems Programmer. Between 1989 and 1992 he joined the Dutch TNO research organization as a Research Scientist where he was actively involved in various international projects, mostly in the field of high-performance computing.

Currently, he is with the Department of Electrical Engineering of Delft University of Technology as an Assistant Professor. His research interests are in the area of performance modeling of parallel systems as well as in the area of parallel programming languages and compilation techniques.