P^3T+: A Performance Estimator for Distributed and Parallel Programs *

T. Fahringer and A. Požgaj
Institute for Software Technology and Parallel Systems, University of Vienna
Liechtensteinstrasse 22, A-1092, Vienna, Austria
[tf,alex]@par.univie.ac.at

Abstract

Developing distributed and parallel programs on today's multiprocessor architectures is still a challenging task. Particular distressing is the lack of effective performance tools that support the programmer in evaluating changes in code, problem and machine sizes, and target architectures. In this paper we introduce P^3T+ which is a performance estimator for distributed and parallel programs. P^3T+ is unique by modeling programs, compiler code transformations, and parallel and distributed architectures. It computes at compile-time a variety of performance parameters including work distribution, number of transfers, amount of data transferred, transfer times, computation times, and number of cache misses. Several novel technologies are employed to compute these parameters: loop iteration spaces, array access patterns, and data distributions are modeled by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of a communication library used by the underlying compiler. Computation times are predicted through pre-measured kernels on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte, etc. P^3T+ has been implemented and is currently evaluated by several application developers.

1 Introduction

Parallelizing and optimizing programs for multiprocessor systems with distributed memory is still a notoriously hard task. In most cases it is the programmer's responsibility to find parallelism, to distribute data (data parallelism) and computations (task parallelism) onto the target architecture, and to apply code transformations in order to improve performance. Programmers are faced with many problems when it comes to examine the performance of their codes:

- What is the effect of a code change in the performance of a program?
- What happens to the performance if problem and machine sizes are modified?
- What is the impact on the performance when a code is ported to another architecture?
- How much performance can be gained by changing a specific machine parameter (e.g. communication bandwidth or cache size)?

Clearly this list is incomplete, but it shows, that tools providing accurate performance information to examine some of these effects are of paramount importance.

Historically there have been two classes of performance tools. On the one hand, there is extensive work done on monitoring distributed and parallel applications which implies several drawbacks: availability of program and target architecture, long execution times, perturbation of measured performance data, and vast amounts of performance data. Monitoring, however, in principle can handle arbitrary complex and large codes and commonly also provides quite accurate results. On the other hand, there is the class of performance estimators that try to statically examine a program's performance without executing it

* This research is partially supported by the Austrian Science Fund as part of Aurora Project under contract SFBF1104.

on a target architecture. This approach suffers mostly by restricting programs and machines that can be modeled as well as by less accurate results. Performance prediction does not require that the target architecture must be available. Moreover, the time needed to compute performance information can be very short.

In this paper we concentrate primarily on performance prediction which has seen many research efforts in the last several years. Traditionally, the quality of performance prediction has been hampered by modeling either programs or architectures with good accuracy but not both of them. Firstly, those methods that provided accurate predictions for applications suffered by some severe restrictions imposed on modeling architectures. Commonly these tools are unable to determine useful parameters reflecting computational and communication overhead. Secondly, performance prediction that concentrates on modeling architectures may not have enough information about the application that executes on this architecture. Statistical models are commonly used to assume a more or less virtual and often unrealistic application behavior. Moreover, very few performance estimators actually consider code transformations and optimizations applied by a compiler.

In this paper we introduce P^3T+, the successor tool of P^3T [20, 13, 14], which models programs, code transformations, and parallel and distributed architectures. The input programs of P^3T+ are written in High Performance Fortran [25, 1] which represents the de-facto standard of high-level data parallel programming. Moreover, P^3T+ analyzes Fortran90 message passing programs generated by the underlying compiler (VFC [2]) which can be executed on parallel and distributed machines such as network of workstations. P^3T+ models communication overhead, work distribution, computation times, and cache misses which is important for both distributed and parallel programs.

P^3T+ invokes a single profile run of the original sequential input program – ignoring all explicit parallel language constructs such as HPF directives – by using SCALA [19] in order to determine execution frequencies and branching probabilities. In order to achieve high estimation accuracy, we aggressively exploit compiler analysis and optimization information. P^3T+ computes a variety of parameters that reflect some of the most important performance aspects of a parallel program which includes: work distribution, number of transfers, amount of data transferred, transfer times, computation times, and cache misses.

Our estimation technology is based on modeling loop iteration spaces, array access patterns, and data distributions by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of the communication library as employed by the underlying compiler. Computation times are predicted through pre-measured kernels on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte, etc.

The rest of this paper is organized as follows: Section 2 outlines the underlying programming and compiler model and presents some basic terminology which we use throughout the paper. The next section discusses related work. In Section 4 we describe P^3T+ and its performance parameters. Finally, some concluding remarks are made and future work is outlined.

2 Model and Basic Terminology

The programs which are estimated by P^3T+ are based on the underlying compilation and programming model of VFC [2] which is a source-to-source parallelization system that translates Fortran90/HPF programs to Fortran90/MPI message-passing SPMD programs. Moreover, P^3T+ also models Fortran90 message-passing programs.

The parallelization strategy of *VFC* is based on data decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. With this method, data arrays in the original program are each partitioned and mapped to the processors of the target architecture. The specification of the mapping of the array elements to the set of processors is called the *data distribution* of that program. A processor is then thought of as *owning* the data assigned to it; these data elements are stored in its local memory. The work contained in the program is distributed according to the data distribution: computations which define the data elements owned by a processor are performed by it – this is known as the *owner computes* paradigm. The processors then execute essentially the same code in parallel, each on the data stored locally. If a computation requires data which is owned by a remote processor, then such non-local data is accessed through inter-processor communication, which is automatically implemented by *VFC* through message passing.

P^3T+ currently supports only a subset of HPF by excluding irregular codes.

The *nesting level* of a statement S is defined as the number of loops enclosing that statement. If S is not enclosed in a loop then S has loop nesting level 0.

3 Related Work

J. Brehm et al. [4] built a user-driven performance prediction tool $PerPreT$ based on an analytical model to predict speedup, execution time, computation time and communication time for parallelization strategies. The tool examines application strategies without requiring a program. Communication and computation times are described by parameterized formulas where parameters describe the the application's problem size and the number of processors. The target machine is modeled by architectural parameters such as the setup times for computation, link bandwidth and sustained computing performance per node (expressed in MFLOP/s). The user can describe the application and machine model through a specific language called LOOP [26]. While $PerPreT$ offers an interesting possibility to evaluate the computation and communication times required by a parallel application, it does not provide informations about work distribution or number of cache misses.

In [28] W. Kaplow et al. present a compile-time method for determining the cache performance of the loop nests in a program and a heuristic that uses this method for compile-time optimization of loop ranges in iteration-space blocking. The cache misses estimations are produced by applying the program's reference string of a loop nest, determined during compilation, to an architecturally parameterized cache simulator. Data reference strings are generated while parsing the source code as opposed to most hardware cache simulators where reference strings are generated at run-time. Data reference strings are then used by a simulator whose results are less accurate than hardware simulation. However, their approach appears to be effective enough for loop optimization techniques.

In [9, 8] M. Clement et al. present a compiler-generated analytical model for the prediction of cache behavior, CPU execution time, and message passing overhead for scalable algorithms implemented in high level data-parallel languages. The performance prediction requires a single instrumentation run of the program with a reduced problem size to generate a symbolic equation for execution time which includes the contributions of each basic block in a program expressed as a function of the problem size and the number of processors. Since the result of this model is an equation rather than a time estimate for a given problem size, the execution time can be differentiated with respect to a given system parameter. The resulting equation is used to determine the sensitivity of the application to changes in that parameter as the problem is scaled up. Their approach is more restricted in terms of program classes that can be handled (e.g. more restricted loops, no GOTOs, etc.) as compared to P^3T+.

M. Faerman et al. [12] introduced the *Adaptive Regression Modeling (AdRM)* which is a method for performance prediction of data transfer operations in network-bound distributed data-intensive applications. The presented technique predicts performance in multi-user distributed environments by employing small network bandwidth probes (provided by the Network Weather Service (NWS) [39]) to make short-term predictions of transfer times for a range of problem sizes. The NWS gathers performance probe data from a distributed collection of resources and catalogues that data as individual performance histories for each resource. It then applies lightweight time series analysis models to each performance history to produce short-term forecasts of future performance levels. *AdRM* combines the NWS measurements with instrumentation data taken from actual application runs to predict the future performance of the application. To capture the relationship between NWS probes and application benchmark data, regression models are used which calibrate the application execution performance to the dynamic state of the system measured by the NWS. The result is an accurate performance model that can be parameterized by "live" NWS measurements to make time-sensitive performance predictions which can be used to support adaptive scheduling of individual components of a distributed system.

In [21], W. Fang et al. present a method for the evaluation of the communication overhead in the SHRIMP multicomputer under a variety of workloads: analytic modeling and event-driven simulation. Using both methods, the authors study the behavior of the system under different communication patterns and report on system performance parameters such as message latency, occupancy of system buffers and network congestion. The purpose of their work is to learn about the behavior of the SHRIMP machine, and to explore the tradeoffs between analytic modeling and simulation as performance prediction techniques. Their analytic model is based on two assumptions: (i) packet inter-arrival times and service times at every component are exponentially distributed, and (ii) the states of any pair of components are independent random variables. While these assumptions do not match the way the system really operates, the authors believe they do not introduce a significant error in the model. Furthermore, the model assumes that each processor executes the same program, that all messages are of the same size and that messages are sent to uniformly distributed destinations.

In [36, 32], A. van Gemund presents a methodology that yields parameterized performance models of parallel programs running on shared-memory as well as distributed-memory (vector) machines. The aim of this research is to estimate performance degradation due to synchronization effects, covering both condition synchronization (task dependency) as well as mutual exclusion (resource contention). The author introduces an explicit, highly structured formalism called PAMELA together with an analysis technique that integrates an approximate analysis of mutual exclusion within a conventional condition synchronisation analysis technique.

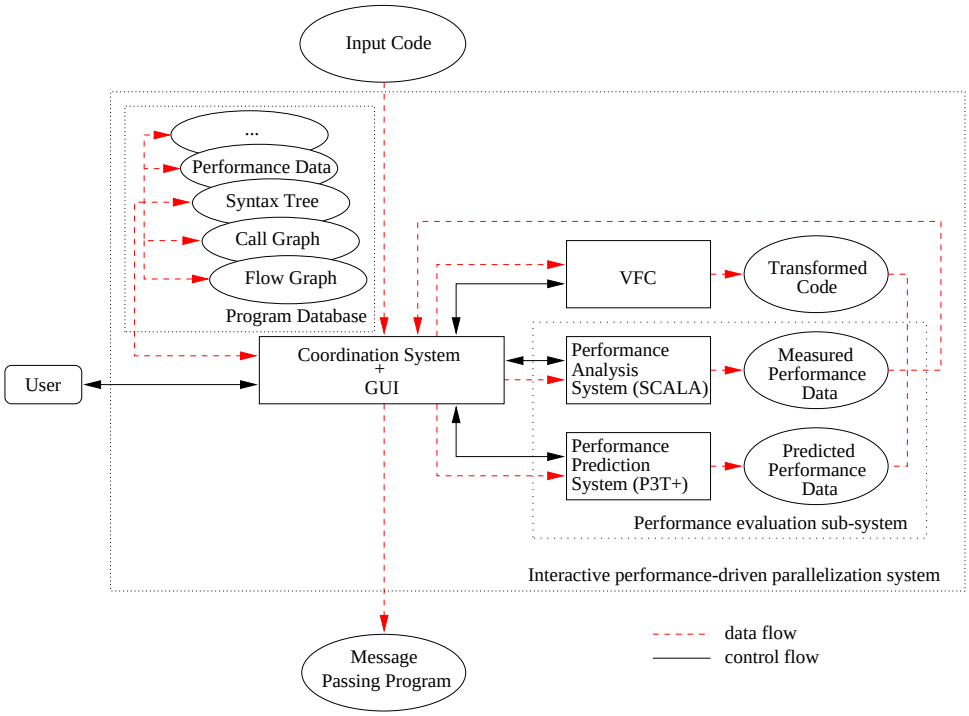

Figure 1. Performance-driven development of distributed and parallel programs

4 P^3T+ A Performance Estimator for Distributed and Parallel Programs

P^3T+ is a state-of-the-art performance estimator that targets both distributed and parallel programs. Figure 1 shows P^3T+ as part of a program development and optimization system. Input programs are parsed and analyzed by VFC which generates syntax trees, call graphs, flow graphs, etc. and stores them in a program database. VFC applies various code transformations and optimizations onto the program with/without user control. The programmer can invoke a performance analysis system (SCALA) to instrument, compile, and execute a distributed or parallel program on the target architecture. Based on the instrumented program execution, performance data is gathered and stored in the program database. Moreover, P^3T+ can be employed to predict the performance behavior of the code transformations and optimizations applied by VFC. P^3T+'s performance data is also stored in the program database. All three tools (VFC, SCALA, and P^3T+) are coordinated and controlled through a coordination system that also includes a graphical user interface (GUI) for displaying source code and performance data and for enabling user interaction. Finally, as a result of performance-driven program development, an optimized distributed or parallel program is created by VFC.

A key issue for a useful performance estimator is to provide critical information to the programmer and compiler which allows steering of the performance tuning process. Most existing tools estimate only execution time. The problem with this parameter is that all important information is hidden in a single runtime figure. As a consequence, the cause of potential performance losses remains unknown. It is not clear whether a parallel program's performance is poor due to cache, load balance, communication or computation behavior. Other performance parameters may as well play an important role. Without making such information transparent, performance tuning is extremely difficult. P^3T+ at compile-time computes a set of performance parameters each of which reflects a different performance aspect. In the following all P^3T+ performance parameters are described.

4.1 Work Distribution

It is well known [7, 5, 37, 35, 27, 33, 11, 34, 30, 23] that the work distribution has a strong influence on the cost/performance ratio of a parallel system. An uneven work distribution may lead to a significant reduction in a program's performance. Therefore, providing both programmer and parallelizing compiler with a work distribution parameter for parallel programs is critical to steer the selection of an efficient data distribution.

Two problems must be solved towards computing the work distribution of a parallel program: first, how much work is contained in a program and second, how much work is being processed by every individual processor. We first consider these problems for loops and then extend our approach to full programs. Consider the following loop nest with a statement S included in a conditional statement.

```
   DO J1 =1,N1
      DO J2=1,N2*J1
         IF (J1 ≤ N2) THEN
S :         A = A + ...
            ...
         ENDIF
      ENDDO
   ENDDO
```

Computing how many times S is executed is equivalent to counting the number of integer solutions of $\mathcal{I} = \{1 \leq J_1 \leq N_1, 1 \leq J_2 \leq N_2 * J_1, J_1 \leq N_2\}$. J_1 and J_2 are (loop) *variables* and N_1, N_2 are *parameters* (loop invariants). Note that we consider $J_2 \leq N_2 * J_1$ to be non-linear, although N_2 is loop invariant. The statement execution count for S is given by:

$$\sum_{J_1=1}^{min(N_1,N_2)} \sum_{J_2=1}^{N_2*J_1} 1 = \begin{cases} \frac{N_1^2*N_2}{2} + \frac{N_1*N_2}{2} & : \text{ if } 1 \leq N_1 \leq N_2 \\ \frac{N_2^3}{2} + \frac{N_2^2}{2} & : \text{ if } 1 \leq N_2 < N_1 \end{cases}$$

In general, every loop implies at least two constraints on its loop variable, one for its upper and one for its lower bound. Additional constraints on both parameters and variables can be implied, for instance, by conditional statements, minimum and maximum functions, data declarations, etc.

We briefly describe a symbolic algorithm which computes the number of integer solutions of a set of linear and non-linear constraints $\mathcal{I}$ defined over $\mathcal{V} \cup \mathcal{P}$ where $\mathcal{P}$ is the set of parameters and $\mathcal{V}$ the set of variables. Every $I \in \mathcal{I}$ is restricted to be of the following form:

$$p_1(\vec{P}) * v_1 + \ldots + p_k(\vec{P}) * v_k \quad REL \quad 0 \tag{1}$$

where REL $\in \{\leq, \geq, <, >, =, \neq\}$ represents an equality or inequality relationship. $\vec{P}$ is a vector defined over parameters of $\mathcal{P}$. $p_i(\vec{P})$ are linear or non-linear expressions over $\mathcal{P}$, whose operations can be addition, subtraction, multiplication, division, floor, ceiling, and exponentiation. Minimum and maximum functions are substituted where possible by constraints free of minimum and maximum functions.

Figure 2 shows the algorithm for counting the number of solutions to a set of constraints, given $\mathcal{I}$ (set of constraints), $\mathcal{P}$, $\mathcal{V}$, E, and $\mathcal{R}$. E is an intermediate result (symbolic expression) for a specific solution E_i of the symbolic sum algorithm. The result $\mathcal{R}$ is a set of tuples $(\mathcal{C}_i, E_i)$ where $1 \leq i \leq k$. Each tuple corresponds to a conditional solution of the sum algorithm. Note that the conditions $\mathcal{C}$ (satisfying (1)) among all solution tuples are not necessarily disjoint. The result has to be interpreted as the sum over all E_i under the condition of $\mathcal{C}_i$ as follows:

$$\sum_{1 \leq i \leq k} \gamma(\mathcal{C}_i) * E_i \tag{2}$$

where γ is defined as

$$\gamma(\mathcal{C}) = \begin{cases} 1 & : \text{ if } \mathcal{C} = \text{TRUE} \\ 0 & : \text{ otherwise} \end{cases} \tag{3}$$

E and $\mathcal{R}$ must be respectively set to 1 and ϕ (empty set) at the initial call of the algorithm.

In each recursion the algorithm (see Figure 2) is eliminating one variable $v \in \mathcal{V}$. First, all lower and upper bounds of v in $\mathcal{I}$ are determined. Then the maximum lower and minimum upper bound of v are searched by generating disjoint subsets of constraints based on $\mathcal{I}$. For each such subset $\mathcal{I}'$, the algebraic sum of the current E over v is computed. Then the sum algorithm is recursively called for $\mathcal{I}'$, the newly computed E, $\mathcal{V} - \{v\}$, $\mathcal{P}$, and $\mathcal{R}$. Eventually at the deepest recursion level, $\mathcal{V}$ is empty, then E and its associated $\mathcal{I}$ represent one solution tuple defined solely over parameters. More details about this algorithm are given in [17].

In what follows we demonstrate how the symbolic sum algorithm can be used to determine the work contained in a loop nest as well as the work to be processed by a generic processor.

SUM($\mathcal{I}$,$\mathcal{V}$,$\mathcal{P}$,E,$\mathcal{R}$)

- **INPUT:**
  $\mathcal{I}$: set of linear and non-linear constraints defined over $\mathcal{V} \cup \mathcal{P}$
  $\mathcal{V}$: set of variables
  $\mathcal{P}$: set of parameters
  E: symbolic expression defined over $\mathcal{V} \cup \mathcal{P}$
- **INPUT-OUTPUT:**
  $\mathcal{R}$: set of solution tuples $(\mathcal{C}_i, E_i)$ where $1 \leq i \leq k$. $\mathcal{C}_i$ is a conjunction of linear or non-linear constraints defined over $\mathcal{P}$. E_i is a linear or non-linear symbolic expression defined over $\mathcal{P}$.
- **ALGORITHM:**

S1: Simplify $\mathcal{I}$

S2: **if** $\mathcal{I}$ is inconsistent (no solution) **then**
return
endif

S3: **if** $\mathcal{V} = \phi$ **then**
$\mathcal{R} := \mathcal{R} \cup \{\mathcal{I}, E\}$
return
endif

S4: Split $\mathcal{I}$

S4.1: Choose variable $v \in \mathcal{V}$ for being eliminated

S4.2: $\mathcal{I}''$:= subset of $\mathcal{I}$ not involving v
$\mathcal{L}$=$\{l_1$,...,$l_a\}$:= set of lower bounds of v in $\mathcal{I}$
$\mathcal{U}$=$\{u_1$,...,$u_b\}$:= set of upper bounds of v in $\mathcal{I}$
a := cardinality of $\mathcal{L}$
b := cardinality of $\mathcal{U}$

S4.3: **for each** $(l_i, u_j) \in \mathcal{L} \times \mathcal{U}$ **do**

$$\mathcal{I}'_{i,j} := \mathcal{I}'' \cup \{ l_1 < l_i, .., l_{i-1} < l_i, l_{i+1} \leq l_i, .., l_a \leq l_i\} \cup \{u_1 > u_j, .., u_{j-1} > u_j, u_{j+1} \geq u_j, .., u_b \geq u_j\} \cup \{l_i \leq u_j\}$$

$$E_{i,j} := \sum_{v=l_i}^{u_j} E$$

SUM($\mathcal{I}'_{i,j}, \mathcal{V} - \{v\}, \mathcal{P}, E_{i,j}, \mathcal{R}$)

endfor

S5: **return**

Figure 2. Symbolic sum algorithm for computing the number of solutions of a set of constraints $\mathcal{I}$

The following code shows a High Performance Fortran - HPF code excerpt with a processor array *PR* of size P.

```
   INTEGER A(N2)
   !HPF$ PROCESSORS :: PR(P)
   !HPF$ DISTRIBUTE (BLOCK) ONTO PR :: A
   DO J1 =1,N1
      DO J2=1,J1 * N1
         IF ( J2 <= N2 ) THEN
S:          A(J2) = ...
         ENDIF
      ENDDO
   ENDDO
```

The loop nest contains a write operation to a one-dimensional array A which is block-distributed onto P processors. Let k ($1 \leq k \leq P$) denote a specific processor of the processor array. Computations that define the data elements owned by a processor k are performed exclusively by k. For the sake of simplicity we assume that P evenly divides N_2. Therefore, a processor k is executing the assignment to A based on the underlying block distribution if $\frac{N_2*(k-1)}{P} + 1 \leq J_2 \leq \frac{N_2*k}{P}$. The precise work to be processed by a processor k is the number of times k is writing A, which is defined by *work(k)*.

The problem to estimate the amount of work to be done by processor k can now be formulated as counting the number of integer solutions to $\mathcal{I}$ which is given by:

$$\begin{array}{rcccl} 1 & \leq & J_1 & \leq & N_1 \\ 1 & \leq & J_2 & \leq & J_1 * N_1 \\ & & J_2 & \leq & N_2 \\ \frac{N_2*(k-1)}{P} + 1 & \leq & J_2 & \leq & \frac{N_2*k}{P} \end{array} \tag{4}$$

In the following we substitute $\frac{N_2*(k-1)}{P} + 1$ by LB and $\frac{N_2*k}{P}$ by UB.

By applying our algorithm we can automatically determine that statement S is approximately executed

$$work(k) = \sum_{1 \leq i \leq 3} \gamma(C_i) * E_i(k)$$

times by a specific processor k ($1 \leq k \leq P$) for the parameters N_1, N_2 and P. $\gamma(C_i)$ is defined by (3) and

- $C_1 = \{UB \leq N_1^2,\ P \leq N_2\}$ with $E_1(k) = \frac{(N_1+UB-LB)*(LB-2*N_1+2*LB*N_1+UB)}{2*N_1^2}$.
- $C_2 = \{\frac{UB}{N_1} > N_1,\ \frac{LB}{N_1} \leq N_1\}$ with $E_2(k) = (N_1 - \frac{LB}{N_1} + 1) * (\frac{N_1^2}{2} - \frac{LB}{2} + 1)$
- $C_3 = \{N_2 \geq P,\ N_1^2 \geq UB + 1\}$ with $E_3(k) = \frac{N_2}{P} * (N_1 - \frac{UB+1}{N_1} + 1)$

Note that by omitting the last two inequalities in (4), we can use the same symbolic sum algorithm to compute the overall work contained in the HPF code excerpt shown above.

Most conventional performance estimators must repeat the entire performance analysis whenever the problem size or the number of processors used are changing. However, our symbolic performance analysis provides the solution of the above problem as a symbolic expression of the program unknowns (P,N_1,N_2, and k). For each change in the value of any program unknown we simply re-evaluate the result, instead of repeating the entire performance analysis.

Let S be an array assignment statement inside of a loop L, where A is the left hand-side array. P^A is the set of processors onto which A is distributed.

Definition 4.1 Optimal amount of work

The arithmetic mean: $owork(S) = work(S, P^A)/|P^A|$ defines the optimal amount of work to be processed by every single processor in P^A.

Based on the optimal amount of work a goodness function for the *useful work distribution* of an array assignment statement in a loop L is defined.

Definition 4.2 Useful work distribution goodness for an array assignment

The goodness of the useful work distribution with respect to an array assignment statement S is defined by

$$wd(S) = \frac{1}{owork(S)} \sqrt[2]{\frac{1}{|P^A|} \sum_{p \in P^A} \big(work(S,p) - owork(S)\big)^2}$$

The above formula is the standard deviation (σ) divided by the arithmetic mean ($owork(S)$), which is known as the variation coefficient in statistics [3]. In [14] we have presented a proof for the lower and upper bound of $wd(S)$ with the following result: $0 \leq wd(S) \leq |P^A| - 1$. Best-case and worst-case work distribution are, respectively, given by $wd(S) = 0$ and $wd(S) = |P^A| - 1$.

Based on Definition 4.2, a work distribution goodness function for loops, procedures, and programs can be defined.

Definition 4.3 Work distribution goodness for loops, procedures, and programs

Let E be a loop, procedure or an entire program with $\varrho(E)$ the set of array assignment and procedure call statements in E, and $freq(S)$ is the execution time frequency of S, then the work distribution goodness for E is defined by:

$$wd(E) = \sum_{S \in \varrho(E)} \frac{freq(S)}{\sum_{S' \in \varrho(E)} freq(S')} wd(S)$$

If S represents a call to a procedure E, then $wd(S) := wd(E)$.

4.2 Communication Parameters

The overhead to access nonlocal data from remote processors on distributed memory architectures is commonly orders of magnitude higher than the cost of accessing local data. P^3T+ estimates this critical performance aspect of a distributed or parallel program by simulating the associated communication behavior and computing the following performance parameters: the number of transfers (NT), the amount of data transfered (TD), and the overall communication time (TT) on a von Neumann architecture. In this paper we describe how P^3T+ models communication caused by Fortran 90 array assignments in the context of regular data distributions. Predicting communication based on Fortran 77 array references has been described in detail in [15].

In what follows, we briefly sketch how *VFC* generates parallel code for Fortran 90 array assignments. Then, we outline the computation of the communication parameters for Fortran 90 array assignments based on a modified VFC runtime system and associated communication libraries

4.2.1 Modeling Fortran 90 Array Assignment Statements (*VFC*)

Distributed arrays, when referenced in a Fortran 90 array assignment statement, can introduce a considerable amount of communication, depending on the data distribution of the arrays involved in the assignment, access patterns implied by array subscripts, and problem and machine size chosen.

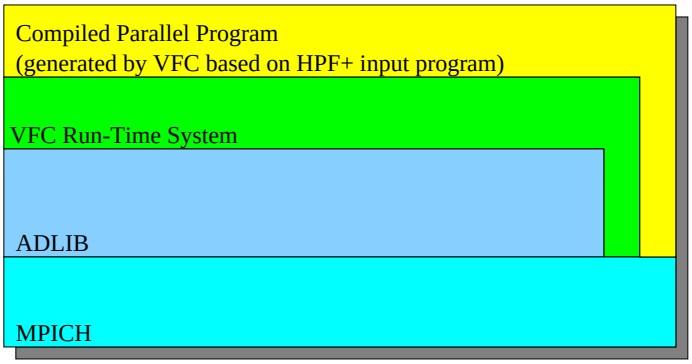

Figure 3. The structure of the compiled parallel program

As shown in Figure 3, a parallel program generated by *VFC* contains calls to the *VFC* Run Time System (*RTS*) which manages distributed data structures (including redistribution of arrays) and provides an interface to communication libraries such as Adlib library [6]. A VFC generated parallel program contains calls to the *RTS* for any kind of communication. *RTS* requires allocation of a runtime descriptor (RD) for every array in a program. The RD is updated during runtime, for instance, when changing the shape of an array or its distribution. Let an array assignment statement S consist of a left-hand side array reference (LHS_ref) and several right-hand side array references (RHS_ref). VFC compiles Fortran 90 array assignment statements as follows:

1. For every array reference in S, a section descriptor (SD) is allocated and initialized. SD describes the array elements (specified by an array section with lower, upper bound and stride for every array dimension) that are touched by a given array reference.

2. Communication buffers are allocated for every different distributed RHS_ref of S

3. For every distributed RHS_ref of S, a call to a RTS routine is inserted with the following parameters: RD and SD of LHS_ref and RHS_ref, and a communication buffer of RHS_ref. The RTS routine is responsible to transfer non-local data to the communication buffer by invoking an Adlib library call which is implemented on top of MPI [24].

4. The subscript expressions of RHS_ref are modified so as references to non-local data are redirected to their associated communication buffers.

For further information on the parallelization of Fortran 90 array assignments in *VFC*, the reader may refer to [2].

4.2.2 Modifying VFC *RTS* and Adlib library

P^3T+ estimates the communication behavior of VFC generated distributed or parallel programs by simulating the behavior of the VFC RTS and associated Adlib library calls on a von Neumann architecture. This means that at compile-time P^3T+ partially executes calls to the VFC RTS and Adlib library for every processor suppressing any actual communication. Only those code sections that compute the sending processor and size of messages are executed. This is achieved by integrating P^3T+ with a modified VFC RTS and Adlib library (see Figure 4) and executing them for every processor of the parallel program at compile-time on a von Neumann architecture.

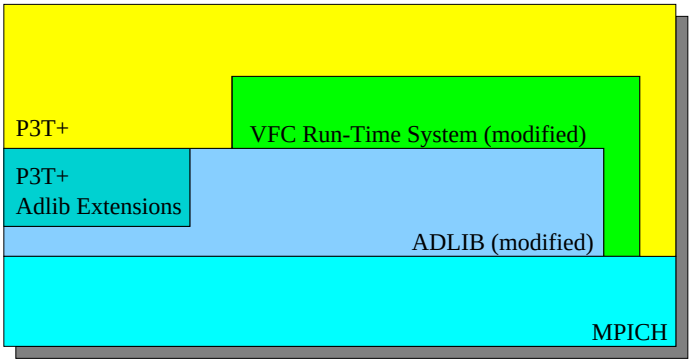

Figure 4. Modified VFC RTS and Adlib library as part of P^3T+.

In *RTS* we suppressed initialization code where the number of processors available on a given parallel architecture is compared with the number of processors requested by the parallel program. The Adlib library has been modified as follows:

- Three global variables have been introduced: *NoOfProcessors* holds the number of processors onto which the parallel program is being executed (as defined by the HPF PROCESSORS directive). *CurrentProcessor* ($1 \leq$ CurrentProcessor $\leq$ *NoOfProcessors*) defines the identification of the current processor that is being simulated by P^3T+ . *CurrentLineNumber* holds the line number of the currently analyzed source code line.
- All calls to functions MPI_COMM_SIZE and MPI_COMM_RANK are, respectively, replaced with a reference to NOOFPROCESSORS and CURRENTPROCESSOR.
- A new data structure – P3T_COMM_SEQUENCE – is introduced which records all SEND operations of a unique statement S. Every entry in P3T_COMM_SEQUENCE holds information about a unique SEND operation by specifying the size of the message in bytes, the sending processor, and the number of the currently analyzed source code line.
- All $send$ operations are suppressed except computation of their parameters which are used to update P3T_COMM_SEQUENCE.
- All $receive$ and $wait$ operations are suppressed.

We use a preprocessor together with conditional code in VFC RTS and Adlib library thus both VFC and P^3T+ can use the same sources. The conditional code is only activated for P^3T+ .

4.2.3 Computing P^3T+ Communication Parameters

In order to estimate the communication behavior of all Fortran 90 array assignments in a parallel program, P^3T+ proceeds as follows:

1. Invoke VFC to generate message passing code based on input program.
2. Traverse VFC generated message passing code and execute pre-compiled communication code – based on modified VFC RTS and Adlib library – for every call to a communication routine R of VFC RTS.
   (a) Update P3T_COMM_SEQUENCE for every processor p that executes R.

3. Compute communication parameter for all code regions (e.g. statements, loops, procedures, and program) of interest based on P3T_COMM_SEQUENCE entries.

Some of the input parameters of RTS and Adlib library calls may require static program analysis or user interaction. For instance, in order to determine number of processors, array shape information, and program unknowns appearing in array subscript expressions, we employ symbolic evaluation as described in [18, 17]. If our symbolic evaluation fails then we request the user for realistic values of input parameters.

Definition 4.4 Communication parameters for an array assignment

Let $\mathcal{S}$ denote the set of array assignments in a program and $\mathcal{F}(S)$ the set of procedures referenced within a statement $S \in \mathcal{S}$. Furthermore, let $\mathcal{K}(S)$ denote the set of communication records (stored in P3T_COMM_SEQUENCE*) associated with S. Then the number of transfers $ntS(S)$, the amount of data transfered $tdS(S)$, and the transfer time $ttS(S)$ for S statement are defined as*

$$ntS(S) = freq(S) * \left(|\mathcal{K}(S)| + \sum_{q \in \mathcal{F}(S)} \frac{ntE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

$$tdS(S) = freq(S) * \left(\sum_{k \in \mathcal{K}(S)} data(k) + \sum_{q \in \mathcal{F}(S)} \frac{tdE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

$$ttS(S) = freq(S) * \left(\sum_{k \in \mathcal{K}(S)} (\alpha + data(k) * \beta) + \sum_{q \in \mathcal{F}(S)} \frac{ttE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

where $freq(S)$ is the execution frequency of S, $calls(q)$ denotes the set of statements calling procedure q in the program, $ntE(q)$ denotes the overall number of transfers for a procedure q, $card(q,S)$ is the number of calls to procedure q in S, $data(k)$ is the amount of data (in bytes) transfered by a message k, and α and β, both measured on the target machine, denote the message startup time and the transfer time per message byte respectively.

Definition 4.5 Communication parameters for a loop nest

Let L denote a loop at the nesting level i, $\mathcal{S}_L$ the set of all statements (excluding nested loop statements and their bodies) appearing in the body of L. Furthermore, let $\mathcal{L}_L$ denote the set of all loops at the nesting level $i+1$, occurring in the body of L. Then the number of transfers $ntL(L)$, the amount of transfered data $tdL(L)$, and the transfer time $ttL(L)$ for L are recursively defined as

$$ntL(L) = \sum_{s \in \mathcal{S}_L} ntS(s) + \sum_{l \in \mathcal{L}_L} ntL(l)$$

$$tdL(L) = \sum_{s \in \mathcal{S}_L} tdS(s) + \sum_{l \in \mathcal{L}_L} tdL(l)$$

$$ttL(L) = \sum_{s \in \mathcal{S}_L} ttS(s) + \sum_{l \in \mathcal{L}_L} ttL(l)$$

Definition 4.6 Communication parameters for a procedure or a program

Let E be a procedure or an entire program, $\mathcal{L}_E$ the set of loop nests with nesting level 0 (correspond to loop nests without enclosing loop) in E. Furthermore, let $\mathcal{S}_E$ denote the set of statements (excluding loop nests) in E, outside of loops. Then, number of transfers $ntE(E)$, amount of transfered data $tdE(E)$, and transfer time $ttE(E)$ implied by all statements $S \in \mathcal{S}_E$ and loop nests $L \in \mathcal{L}_E$, are defined as

$$ntE(E) = \sum_{s \in \mathcal{S}_E} ntS(s) + \sum_{l \in \mathcal{L}_E} ntL(l)$$
$$tdE(E) = \sum_{s \in \mathcal{S}_E} tdS(s) + \sum_{l \in \mathcal{L}_E} tdL(l)$$
$$ttE(E) = \sum_{s \in \mathcal{S}_E} ttS(s) + \sum_{l \in \mathcal{L}_E} ttL(l)$$

4.3 Computation Times

The computation time parameter reflects the time required by a processor to execute local computations of the program excluding communication. By local computations we mean those computations assigned to a processor according to the SPMD programming model and the "owner computes paradigm" (see Section 2). This parameter can be useful to

- analyze the communication/computation relationship by incorporating also communication parameters described in Section 4.2
- evaluate whether there is enough computation contained in a code region, thus parallelizing the code region may be effective.
- identify the most time-consuming code regions of the program (*hot spots*) which are often hard to isolate without the help of a profiling tool.

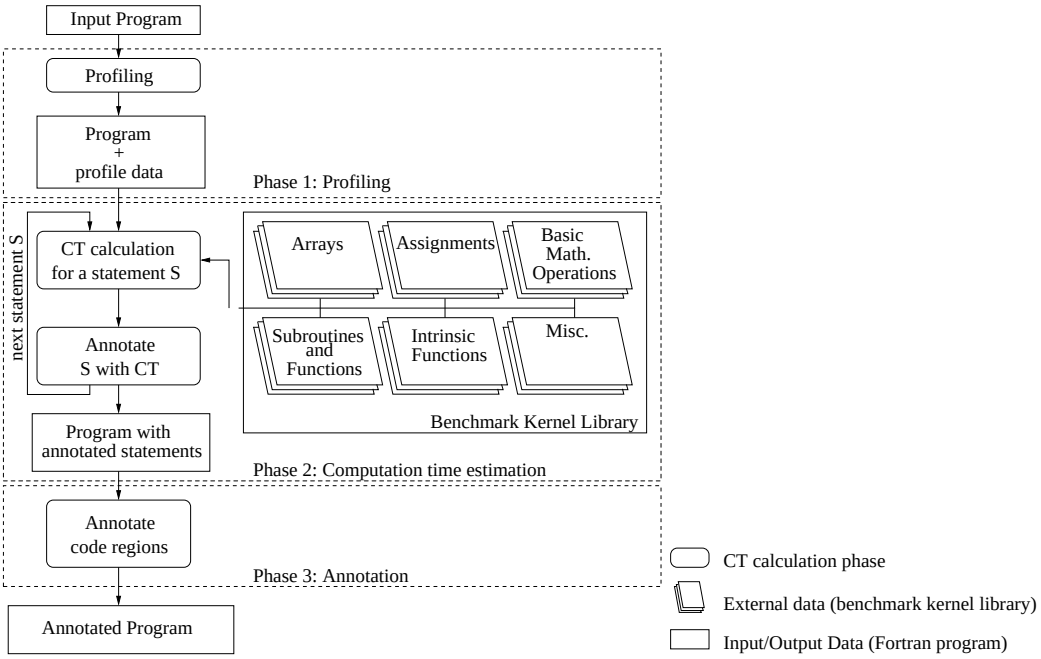

Figure 5. Estimating computation times under P^3T+

Our method for predicting computation times employs statement execution frequencies and branching probabilities as well as pre-measured kernel codes. Pre-measured kernel codes are used to associate statements and small code sections in the input program with pre-measured execution times for a specific target machine. A large set of kernel codes are pre-measured for every target machine of interest and stored in a benchmark kernel library.

Figure 5 shows the architecture of the CT parameter. Given the Fortran program and the profiling information for a specific set of input data, the computation time parameter is estimated for each statement separately by pattern matching against pre-measured kernels stored in the benchmark kernel library.

In what follows we describe the set of kernels upon which our techniques are based on. Then, we will discuss the training phase of the benchmark kernel library which measures all kernels once for every different target machine of interest. Finally, we describe how to estimate computation times based on pre-measured kernels and profiling data.

4.3.1 Benchmark Kernel Library

The benchmark kernels of the computation time parameter can be classified as follows:

1. **Assignments**
   Scalar assignment operations considering several cases where the data types of left-hand side and right-hand side scalars are identical or different. Different data types may cause additional overhead due to type conversion.

2. **Basic mathematical operations**
   Basic mathematical operations, such as +, -, *, /, **.

3. **Procedures (subroutines and functions)**
   Subroutine call and function reference overheads for varying numbers of parameters.

4. **Intrinsic functions**
   Standard intrinsic functions, like SIN, COS, MOD, LOG, etc. and implicit reduction functions included in Fortran such as MIN, MAX, SUM, and INDEX.

5. **Arrays**
   Kernels for array reference address calculations.

6. **Miscellaneous**
   All other kernels comprising, for instance, boolean operations, IF-THEN-ELSE constructs, loop headers, etc.

4.3.2 Training Phase

The performance estimator has to be trained once for all different target machines of interest in order to determine computation times for each different kernel in the kernel library. This is achieved by a training phase. Primitive statements and most primitive operations - except array operations - are measured for different data types and stored in the benchmark kernel library as numeric values. Computation times for array operations and intrinsic functions depend not only on the data types involved but also on the data size of arrays and the number of parameters which are considered by the measurements. Based on these measurements a set of functions describing the computation times for different access patterns, data types, and problem sizes is constructed using the chi-square fit method and stored in the benchmark kernel library.

4.3.3 Estimating computation times

Obtaining computation times for a program essentially involves 3 phases as shown in Figure 5.

1. The input program is instrumented and executed once on a von Neumann architecture. The profile data is used to annotate the program with execution frequencies and branching probabilities.

2. For every statement, a kernel pattern matching in combination with a performance evaluation algorithm is invoked. Primitive operations, primitive statements and intrinsic functions are simply detected by their syntax tree node representation. The computation times for every statement are then weighted by their execution frequencies or branching probabilities (in case of conditional statements) which yields the overall execution time for a statement. Every statement is annotated with the estimated computation time as obtained from this phase.

3. Estimated computation times for larger code regions (e.g. loops, procedures, and programs) are obtained by summing up the corresponding computation times of all statements in this region. Larger code patterns (e.g. matrix multiplication) may require more advanced pattern matching techniques such as those mentioned in [10]. The current implementation of our pattern matcher handles all kernels in the benchmark kernel library except code patterns. The output of phase 3 is the program annotated with computation times for all code regions.

In the following, we define the computation time for a single statement, loop nest, procedure and entire program.

Definition 4.7 Computation time for a program statement

Let $\mathcal{S}$ denote the set of statements of a program and $\mathcal{F}(S)$ the set of procedures referenced within a statement $S \in \mathcal{S}$, then the accumulated time $ctS(S)$ for this statement is defined as

$$ctS(S) = freq(S) * \left(ctS_{simple}(S) + \sum_{q \in \mathcal{F}(S)} \frac{ctE(q) * card(q,S)}{\sum_{S' \in calls(q)} freq(S') * card(q,S')} \right)$$

where $ctS_{simple}(S)$ denotes the computation time required by the single instantiation of S excluding the computation time required by any procedures referenced in that statement. The set of statements referencing procedure q in program is given by $calls(q)$. $ctE(q)$ denotes the overall computation time of a procedure q, $freq(S)$ the execution frequency of the statement S, and $card(q,S)$ the number of references to procedure q in S.

Definition 4.8 Computation time for a loop nest

Let L denote a loop at the nesting level i, $\mathcal{S}_L$ the set of all statements (excluding nested loop statements and their bodies) appearing in the body of L. Further, let $\mathcal{L}_L$ denote the set of all do loops at the nesting level $i+1$ occurring in the body of L. Then the computation time for L denoted by $ctL(L)$ is defined as

$$ctL(L) = \sum_{s \in \mathcal{S}_L} ctS(s) + \sum_{l \in \mathcal{L}_L} ctL(l)$$

Definition 4.9 Computation time for a procedure or a program

Let E be a procedure or an entire program and $\mathcal{L}_E$ the set of loop nests with nesting level 0 in E. Further, let $\mathcal{S}_E$ denote the set of statements in E outside of loops. Then the accumulated computation time $ctE(E)$, implied by all statements $S \in \mathcal{S}_E$ and loop nests $L \in \mathcal{L}_E$, is defined as

$$ctE(E) = \sum_{s \in \mathcal{S}_E} ctS(s) + \sum_{l \in \mathcal{L}_E} ctL(l)$$

4.4 Number of Cache Misses

It is well known [16, 38, 31, 29, 22] that inefficient memory access patterns and data mapping into the memory hierarchy (data locality problem) of a single processor cause major program performance degradation. P^3T+ estimates the number of accessed cache lines which correlates with the number of cache misses. This parameter is derived for loops, procedures, and entire programs.

The main idea of our estimation approach for cache misses is that array references are grouped into array access classes such that all arrays in a specific class exploit reuse of array elements in the same set of array dimensions. The definition of array access classes is based on a specific number of innermost loops of a not necessarily perfectly nested loop L. Two array references are in the same array access class for a loop nest if they actually access some common memory location in the same array dimensions and reuse occurs in L. The common accesses occur on either the same or a different iteration of L.

In the following we define the number of cache misses for a loop nest, procedure, and entire program.

Definition 4.10 Number of cache misses for a loop nest

Let P define the set of processors executing the loop nest L and $\mathcal{F}(L)$ the set of procedures referenced within L. Furthermore, let $cmL^p(L)$ define the number of cache misses induced by a single instantiation of L with respect to a processor $p \in P$, excluding the cache misses implied by procedure calls within L. Then the overall number of cache misses induced by L with respect to all processors in P is defined as

$$cmL(L) = freq(L) * \left(\frac{1}{|P|} \sum_{p \in P} cmL^p(L) + \sum_{q \in \mathcal{F}(L)} \frac{cmE(q) * card(q, L)}{\sum_{S \in calls(q)} freq(S) * card(q, S)} \right)$$

where $calls(q)$ denotes the set of statements calling procedure q in the program, $cmE(q)$ denotes the accumulated number of cache misses implied by procedure q (see Definition 4.11), $freq(S)$ and $freq(L)$ denote the execution frequency of statement S and loop nest L respectively, and $card(q, S)$ is the number of calls to procedure q in S.

The first sum in Definition 4.10 describes the mean value of cache misses implied by a single instantiation of L across all processors in P executing L. The second sum is explained as follows: in order to take procedure calls into account, the parameter outcome for a single procedure call instantiation is supposed to be independent of the call site. This means that the parameter outcome at a particular call site is the same as the parameter outcome of the procedure over all call sites, which is a common assumption made for performance estimators. The estimated number of cache misses for every specific loop is weighted by its execution count ($freq$) in order to reflect its impact on the overall program performance.

All call graphs are supposed to be acyclic. Note that Definition 4.10 is also applicable to a sequential program iff $|P| = 1$.

Extending the cache cost function to a procedure or a program is straight forward:

Definition 4.11 Number of cache misses for a procedure or a program

Let E be a procedure or an entire program, and $\mathcal{L}_E$ and $\mathcal{S}_E$, respectively, denote the set of loop nests and statements with procedure calls at nesting level 0 in E. Furthermore, let $\mathcal{F}(S)$ denote the set of procedures referenced within a statement $S \in \mathcal{S}$. Then the number of cache misses induced by all loop nests $L \in \mathcal{L}_E$ and statements $S \in \mathcal{S}_E$ is defined as follows:

$$cmE(E) = \sum_{l \in \mathcal{L}_E} cmL(l) + \sum_{S \in \mathcal{S}_E} \sum_{q \in \mathcal{F}(S)} \frac{cmE(q) * card(q, S)}{\sum_{S' \in calls(q)} freq(S') * card(q, S')}$$

where $calls(q)$, $freq(S)$ and $card(q, S)$ are defined as in Definition 4.10.

The first sum in Definition 4.11 corresponds to the loops contained in E. The second sum models procedure calls outside of loops in E. It is assumed that the same cache behavior is implied by every instantiation of L. A more accurate modeling of cmE requires separate values regarding $freq(L)$ for every instantiation of L at the price of a considerably larger computational effort.

More details about our cache modeling approach can be found in [14].

5 Conclusions

In this paper, we have described P^3T+, a performance prediction tool for parallel and distributed programs. In contrast to most other performance estimators P^3T+models programs, code transformations, and parallel and distributed architectures. P^3T+ computes a variety of performance parameters including work distribution, number of transfers, amount of data transferred, transfer times, computation times, and number of cache misses. Several novel technologies are employed to compute these parameters: loop iteration spaces, array access patterns, and data distributions are modeled by employing highly effective symbolic analysis. Communication is estimated by simulating the behavior of a communication library used by the underlying compiler. Computation times are predicted through pre-measured kernels on every target architecture of interest. We carefully model most critical architecture specific factors such as cache lines sizes, number of cache lines available, startup times, message transfer time per byte. P^3T+ has been implemented and is currently evaluated by several application developers. In the future, we will work on performance prediction for object-oriented multi-threaded programs that cover both data and task parallelism.

References

[1] S. Benkner. HPF+: High Performance Fortran for advanced industrial applications. *Lecture Notes in Computer Science*, 1401, 1998.

[2] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1998.

[3] J. Bleymüller, G. Gehlert, and H. Gülicher. *Statistik für Wirtschaftswissenschaftler*. Verlag Vahlen, München, 1985. WiSt Studienkurs.

[4] J. Brehm, M. Madhukar, E. Smirni, and L. Dowdy. PerPreT — A performance prediction tool for massively parallel systems. *Lecture Notes in Computer Science*, 977, 1995.

[5] B. Carlson, T. Wagner, L. Dowdy, and P. Worley. Speedup properties of phases in the execution profile of distributed parallel programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, pages 83–95, 1992. (Ed.) R. Pooley and J. Hillston.

[6] B. Carpenter. Adlib: A Distributed Array Library to Support HPF Translation. In *Proc. of the 5th Workshop on Compilers for Parallel Computers, Malaga, Spain*, June 1995.

[7] B. Chapman, T. Fahringer, and H. Zima. *Automatic Support for Data Distribution*, pages 184–199. Springer Verlag, Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing, 6th International Workshop, (Ed.) U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Portland, Oregon, Aug. 1993.

[8] M. Clement and M. Quinn. Symbolic Performance Prediction of Scalable Parallel Programs. In *Proc. of 9th International Parallel Processing Symposium*, St. Barbara, CA, April 1995.

[9] M. J. Clement and M. J. Quinn. Dynamic performance prediction for scalable parallel computing. Technical Report 95-80-04, Oregon State University.

[10] B. DiMartino. Algorithmic Concept Recognition Support for Automatic Parallelization: A Case Study for Loop Optimization and Parallelization. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, to appear in March 1998.

[11] D. Eager, J. Zahorjan, and E. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408 – 423, March 1989.

[12] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. Technical Report CS1999-0619, University of California, San Diego, Computer Science and Engineering, May 18, 1999.

[13] T. Fahringer. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer*, 28(11):47 – 56, November 1995.

[14] T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, Boston, USA, ISBN 0-7923-9708-8, March 1996.

[15] T. Fahringer. Compile-Time Estimation of Communication Costs for Data Parallel Programs. *Journal of Parallel and Distributed Computing, Academic Press*, 39(1):46–65, Nov. 1996.

[16] T. Fahringer. Estimating cache performance for sequential and data parallel programs. In *Proc. of the International Conference and Exhibition on High-Performance Computing and Networking (HPCN'97), Vienna, Austria*, pages 840–850. Lecture Notes in Computer Science, Springer Verlag, 1997.

[17] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3):227–252, May 1998.

[18] T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the 11th ACM International Conference on Supercomputing*, pages 261–268, Vienna, Austria, July 1997. ACM Press.

[19] T. Fahringer, B. Scholz, and M. Pantano. Execution-Driven Performance Analysis for Distributed and Parallel Systems. Technical Report, Institute for Software Technology and Parallel Systems, University of Vienna, Liechtensteinstr. 22, A-1090 Wien, June 1999.

[20] T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Proc. of the 7th ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993. ACM Press. best paper award.

[21] W. Fang, E. W. Felten, and M. Martonosi. Contention and queueing in an experimental multicomputer: Analytical and simulation-based results. Technical Report TR-508-96, Princeton University, Computer Science Department, Jan. 1996.

[22] J. Ferrante, V. Sarkar, and W. Trash. On estimating and enhancing cache effectiveness. In *Proc. of the 4th Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, Aug 1991.

[23] D. Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

[24] M. P. I. Forum. *Document for a Standard Message Passing Interface*, draft edition, Nov. 1993.

[25] High Performance FORTRAN Language Specification. Technical Report, Version 2.0.δ, Rice University, Houston, TX, October 1996.

[26] J. Brehm et al. A Multiprocessor Communication Benchmark, Users Guide and Reference Manual. *Public Report of the ESPRIT III Benchmarking Project*, 1994.

[27] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley Professional Computing, 1991.

[28] W. K. Kaplow and B. K. Szymanski. Program optimization based on compile-time cache performance prediction. *Parallel Processing Letters*, 6(1):173–184, Mar. 1996.

[29] K. Kennedy and K. McKinley. Optimizing for Parallelism and Data Locality. In *International Conference on Supercomputing 1992*, pages 323–334, Washington D.C., July 1992.

[30] M. Kumar. Measuring parallelism in computation intensive scientific/engineering applications . *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[31] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked Algorithms. In *In Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara,CA, April 1991.

[32] A. van Gemund's *PAMELA* project webpage. http://dutepp0.et.tudelft.nl/~gemund/Pamela/pamela.html.

[33] K.-H. Park. *Dynamic Processor Partitioning for Multiprogrammed Multiprocessor Systems*. PhD thesis, Vanderbilt University, Nashville, TN, Aug 1990.

[34] K. Sevcik. Characterization of parallelism in applications and their use in scheduling . *Performance Evaluation Review*, 17(1):171–180, 1989.

[35] C. Siddhartha. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, October 1991.

[36] A. van Gemund. *Performance Modeling of Parallel Systems*. Delft University Press, 1996.

[37] S. Venugopal and V. K. Naik. SHAPE: a parallelization tool for sparse matrix computations. Research report rc 17899, IBM Research Division, T.J. Watson Research Center, Yorktwon Heights, NY 10598, July 1992.

[38] M. Wolf and M. Lam. A data locality optimizing algorithm. In *In Proceedings of the SIGPLAN 91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[39] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6), 1999.