# Using High-Level Performance Prediction in Compiling for Distributed Systems

Arjan J.C. van Gemund
http://dutepp0.et.tudelft.nl/~gemund

Department of Electrical Engineering
Delft University of Technology
P.O.Box 5031, NL-2600 GA Delft, The Netherlands

**Abstract**

*In cost-driven program optimization performance feedback is either based on a model of the algorithm or on a model of the actually generated machine code. Especially in the case of a distributed-memory system, the difference in abstraction is large. In this paper we study the trade-off between prediction at high (program) level and low (machine) level in the context of automatic optimization for message-passing architectures. We present a prediction technique based on modeling the various optimizations in terms of resource contention. Despite the abstraction, we show that high level modeling yields more reliable predictions provided this technique is used. We illustrate this result by deriving various optimizations of a line relaxation kernel for distributed-memory machines.*

## 1 Introduction

Performance prediction is crucial in the design and the compilation of parallel programs. This especially applies to automatic program optimization for distributed-memory machines considering the impact of data layout in overall performance. The utility of performance prediction techniques greatly depends on the abstraction level at which they are applied and the (internal) analysis technique that is used. Organized in terms of the abstraction level, performance prediction approaches are characterized by two levels, namely the *feedback* level, being the level at which the performance feedback is used, and the *modeling* level, being the code abstraction level which is used as a modeling basis (see Figure 1). With respect to feedback, we distinguish between the *user* level and the *compiler* level. In the latter case, performance feedback is used to guide automatic optimization, involving decisions on pipelining, data partitioning, etc. With respect to modeling, we distinguish between two

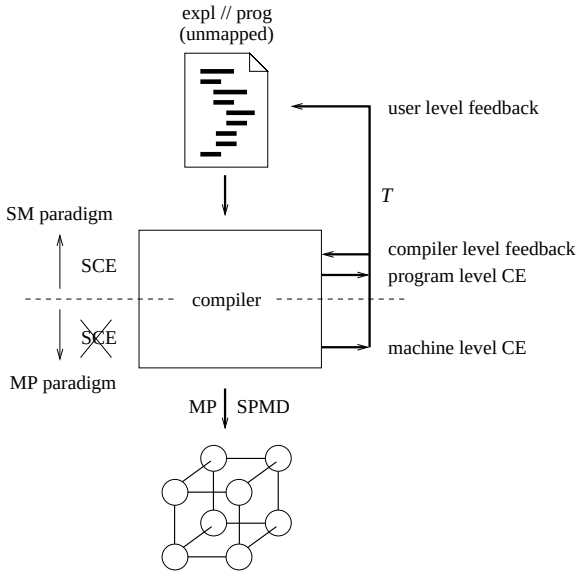


Figure 1: Various abstraction levels in automatic program optimization based on performance feedback.

levels, namely program level and machine level. At program level, modeling is based on an intermediate representation of the program under optimization, still expressed in terms of a shared-memory programming model. At machine level, modeling is based on the actual compiler-generated output which is expressed in terms of the message-passing machine architecture.

At the compiler feedback level, but also at user level, *symbolic* cost estimation techniques (denoted SCE in the figure) can play an extremely important role. Especially techniques capable to deliver minimum-complexity, *closed-form*, algebraic cost estimates are important because they allow the compile-

1060-3425/98 $10.00 (c) 1998 IEEE

time optimization process to be optimized itself. The reason is that generic optimization schemes will essentially involve cost estimate *comparisons*, the code of which we shall denote *optimization logic*. When the cost estimates are closed-form expressions, the comparisons themselves can be significantly reduced (partly at meta-compile-time, i.e., the time at which the compiler itself is compiled for a specific target machine). This opens up the real possibility of a "model of compilation" based on *efficient* optimization logic that, depending on program parameterization, is evaluated either at compile-time or at run-time, being an integral part of the object code.

Returning to the above taxonomy, a principle question is what modeling level to choose. This question is especially topical when optimizing for message-passing machines. In this paper we consider this matter for some task or data parallel language (e.g., an HPF host language extended with some coordination mechanism) being compiled for *message-passing* architectures. At the compiler level the programming model is still procedure-oriented [2] which indeed allows the use of static techniques in which the source code can be compiled into a symbolic performance model [7]. However, the abstraction from the actual message-passing machine implies that the machine model used is based on assumptions regarding the actually generated message-passing output produced by a machine-dependent SPMD code generation phase which might introduce performance surprises. In contrast, at the machine level, the programming model does conform to the actual message-oriented [2] machine interface which avoids the need to predict the performance of the low level code generation model. Consequently, the inherent accuracy potential of machine-level prediction is high. A trade-off in the case of message-passing architectures, however, is that the message-oriented paradigm of the code inhibits the ability to apply symbolic prediction techniques[1].

In summary, the choice of prediction level concerns the trade-off between modeling accuracy and symbolic, closed form predictions. Clearly, this choice depends on the feedback level that is of primary interest. At user level the trade-off may well be in favor of machine-level modeling which allows the use of simulation techniques [15, 17, 19, 20]. For compiler level feedback, however, the choice is effectively limited to program-level cost modeling [3, 5, 6, 7, 10, 16, 18, 21].

This paper addresses the trade-off between performance modeling at program and machine-level in the context of automatic, compile-time optimization for message-passing architectures. More specifically,

- We show that, despite the abstraction at program-level, (symbolic) predictions can be derived that are capable to account for the most significant performance effects that occur in the actual message-passing SPMD code. We demonstrate the use of these models in the automatic derivation of optimization logic by discussing vectorization as well as data (re)mapping for a line relaxation kernel on a distributed-memory machine.

- We show that the choice for machine-level modeling inhibits the use of symbolic prediction techniques. This fundamental aspect is illustrated by discussing the compilation of the line relaxation example. The generated message-passing code does not properly reveal the effects of data partitioning whereas our abstract program-level symbolic model indeed produces the correct prediction.

Especially, the last result is somewhat counter-intuitive, and demonstrates the large potential of program-level prediction, provided the right cost modeling methodology is used.

In Section 2 we present this modeling methodology, which is based on a concept called *contention modeling*. We also show how our cost modeling technique provides a framework for the automatic compilation of optimization logic. In Section 3 we demonstrate the effectiveness of our approach through a case study in which we derive a number of optimizations for the line relaxation algorithm on a message-passing machine. Finally, in Section 4 we summarize our work.

## 2 Cost Estimation

In this section we introduce our cost modeling approach which is based on the PAMELA (PerformAnce ModEling LAnguage) methodology [9]. More specifically, we focus on the PAMELA subset for which symbolic, closed-form cost estimations can be derived. Though the formalism and associated cost estimation technique is described at length elsewhere [8], for convenience of reference we will briefly summarize the technique in Section 2.1. In Section 2.2 we describe our specific modeling approach called "contention modeling". In Section 2.3 we present the rationale behind our approach. In Section 2.4 we describe how our technique is used to derive optimizations.

[1] Unless the communication architecture is restricted to *collective* communications [4]. However, this implies that the model is in principle procedure-oriented rather than message-oriented.

## 2.1 Modeling Algebra

The language subset we consider comprises a process algebra based on sequential, parallel, and conditional composition operators. The algebra features binary infix operators to describe sequential composition (';'), and fork/join-style parallel composition ('‖'). Sequential and parallel replication are expressed by **seq** and **par** prefix operators, defined by **seq** $(i = a, b)\ L_i = L_a\ ;\ \ldots\ ;\ L_b$ and, similarly, **par** $(i = a, b)\ L_i = L_a \parallel \ldots \parallel L_b$. The algebra also features **if** and **while** operators which are not used in this paper.

While the *condition synchronization* [2] (CS for short) provided by the above operators allows for the expression of any series-parallel (SP) computation structure, *mutual exclusion* [2] (ME for short[2]) can be specified by the **use** construct, like in $\mathbf{use}(r, \tau)$ where the invoking process exclusively acquires resource $r$ (FIFO without preemption, non-deterministic conflict resolution) for $\tau$ *time* (excluding possible queuing delay). In fact, any time delay associated with spending cycles (i.e., work) in a computation is expressed (i.e., charged to some resource) by **use** statements. A resource $s$ can have a multiplicity, denoted $|s|$, that may be larger than 1. Like in queuing networks, it is convenient to define a resource $\rho$ such that $|\rho| = \infty$, usually called infinite server. Instead of $\mathbf{use}(\rho, \tau)$ we will simply write $\mathbf{delay}(\tau)$. Unlike the **use** operation, a **delay** operation will never entail additional queuing delay on top of its programmed delay.

As a simple example, consider the PAMELA model (by convention denoted $L$), of some parallel computation described by the following process expression $L = \mathbf{par}\ (p = 1, P)\ \mathbf{use}(cpu_{f(p)}, \tau_p)$ where $cpu_{f(p)}$ denotes a processing resource and $\tau_p$ models its workload. If $cpu_{f(p)}$ is unique (e.g., a mapping $f(p) = p$), $L$ represents a time delay equal to $T = \tau_1 \max \ldots \max \tau_p$, as each **use** statement runs in parallel. In contrast, however, if $f(p) = c$ (i.e., constant, each process is mapped to the same CPU), it follows $T = \tau_1 + \ldots + \tau_p$, as a result of the serialization of the $N$ parallel requests due to ME. Although PAMELA features more operators, essentially, our approach to modeling parallel computation will be expressed in terms of **par**, **seq**, and **use**.

A PAMELA model $L$ can be directly executed which corresponds to performance simulation. However, the highly structured operators for both CS (**par**, **seq**) and ME (**use**) enable a simple compile-time analysis technique through which PAMELA models can be compiled into symbolic cost models through a completely mechanical procedure. Due to the non-determinism that arises with ME, the time cost of computations in which ME plays a role is typically stochastic. Aimed to provide a low-cost, analytic (i.e., deterministic) estimate, the analysis algorithm computes a lower bound which in some cases entails a prediction error. The specific advantage of the estimation algorithm compared to conventional techniques (static analysis, complexity analysis), however, is that the estimation error of $T$ is bounded. Theory and experiments show that the average estimation error due to the synchronization effects is less than a constant factor 2, *regardless* of the type of parallel computation (as long as it can be expressed in terms of our process algebra), while in most cases the average error is well within tens of percents. Although applied in the sequel, due to space considerations the cost estimation algorithm itself is not described in the paper. Details can be found in [8].

## 2.2 Contention Modeling

A classical example in performance modeling is the machine repair model (MRM) [14] in which $P$ clients either spend a mean time $\tau_l$ on local processing, or request service from a server $s$ ($s = 1$), with mean service time $\tau_s$, for a total cycle count of $N$ iterations. Both time delays are typically stochastic. The PAMELA model of the MRM is given by

$$
\begin{array}{l}
L = \mathbf{par}\ (p = 1, P) \\
\qquad \mathbf{seq}\ (i = 1, N) \\
\qquad\quad \{\ \mathbf{delay}(\tau_l)\ ;\ \mathbf{use}(s, \tau_s)\ \}
\end{array}
$$

in which the exclusive service is expressed by the **use** operation applied to the passive resource $s$ that represents the server. Note that in our modeling approach the server is a passive *resource*. The ME arising from sharing the resource is modeled in terms of *contention*, rather than "communication". (In a message-oriented approach $s$ is modeled by a reactive *process*.) Hence, we have coined our modeling approach *contention modeling*[3]. We will discuss the disadvantage of the alternative, message-oriented approach later on.

Despite the advantages in analytical sense, as described earlier, it would seem that the constraints imposed by the highly structured synchronization operators entail a drastic reduction in modeling power. For

[2] CS represents the static form of process synchronization, while ME represents dynamic process synchronization ("contention").

[3] The original terminology *material*-oriented modeling, and its dual, *machine*-oriented modeling, stem from the domain of simulation of, e.g., plant production lines [13]. We feel that the application of the material-oriented paradigm in the specific domain of parallel programming justifies using the distinct name "contention modeling".

example, the SP restriction with respect to CS would make it impossible to model a fundamental parallel computation schedule such as *pipelining*. However, the use of contention modeling does allow pipelining to be expressed in terms of our cost modeling framework. Consider a pipelined computation involving $N$ data sets processed by an $S$ stage pipeline (e.g., vector unit, packet-switched communication pipeline, software pipeline). In a message-oriented paradigm each pipeline unit would map to a process that would synchronously receive a data set, process it, and send it to the next unit. In contention modeling, however, the entire computational process is expressed for each data set. Each data process is executed in parallel and *contends* for each unit in the course of its propagation through the pipeline. The PAMELA model is given by

$$L = \textbf{par}\ (i = 1, N)\ \textbf{seq}\ (s = 1, S)\ \textbf{use}(u_s, \tau_c)$$

where $u_s$ denotes the resource corresponding to stage $s$, and $\tau_c$ denotes the associated processing time (cycle time) per unit. The above model correctly accounts for both startup delay as well as the bandwidth of the pipeline. Note that, while the absolute order in which data is processed is left undetermined, the *time cost* prediction is valid. Application of the earlier estimation algorithm (including optimizations [9]) yields the exact result $T = (S + N - 1)\tau_c$.

Notice that in our contention modeling paradigm the pipeline can be conveniently expressed as an SP model based on the use of ME, whereas a usual description of all the task precedences merely in terms of CS would necessitate a non-SP description (which is not amenable to our cost compilation algorithm). Hence, the example is a typical illustration of the modeling power of the contention modeling technique. It should be stressed that contention modeling is not some contrived way of describing parallel computation. It essentially expresses (and preserves) the *potential* parallelism that exists within the *algorithm* independent of the actual machine *implementation*, where ME models the specific resource limitations. Thus it offers a *portable* way of modeling.

### 2.3 Rationale

The choice for the high-level, contention-oriented modeling paradigm in PAMELA is motivated by the fact that message-oriented models are not amenable to symbolic cost estimation. This is why symbolic cost modeling is best applied at program level where the programming model is still a shared-memory model instead of at the message-passing machine level. We illustrate the fundamental problems involved with analyzing message-passing models with a simple example. Recall the MRM. In a message-oriented paradigm, both clients and server would map to processes that 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 [11] using synchronous **send** and **recv** operators combined with a selective communication construct ('$\Box$') to achieve scheduling nondeterminism. Let $L_p$, $p = 1, \ldots, P$ denote the $P$ client processes and let $S$ denote the server. The MRM is modeled by the following set of process equations

$$\begin{aligned}
L = &\ S \parallel \textbf{par}\ (p = 1, P)\ L_p \\
L_p = &\ \textbf{seq}\ (i = 1, N)\ \{ \\
&\quad \textbf{delay}(\tau_l)\ ;\ \textbf{send}(S)\ ;\ \textbf{recv}(S) \\
&\}\\
S = &\ \textbf{while}\ (\textbf{true})\ \{ \\
&\quad \textbf{recv}(L_1)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_1)\ \Box \\
&\quad \textbf{recv}(L_2)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_2)\ \Box \\
&\qquad \ldots \\
&\quad \textbf{recv}(L_P)\ \rightarrow \textbf{delay}(\tau_s)\ ;\ \textbf{send}(L_P) \\
&\}
\end{aligned}$$

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

In contrast to the contention model, for the above CSP-type solution there exists no general, automatic analysis scheme that produces a useful symbolic cost estimate. Especially when message-passing models become complex, it is impossible to efficiently deduce the critical synchronization path that now actually runs across multiple processes. The problem is due to the use of a *low-level* non-determinism operator $\Box$, in combination with providing *separate* constructs for sending and receiving which might lead to non-SP structures. In contention modeling, CS and ME are kept orthogonal, both in terms of *single*, high-level operations. Consequently, the need to first "reverse engineer" to a higher level model (impossible in a mechanized scheme) is completely avoided.

This property of the message-oriented modeling paradigm implies that program code expressed in terms of a message-passing architecture can not be symbolically analyzed in terms of closed-form expressions. This is the essential rationale for choosing a program-level contention modeling approach for deriving program optimization logic. Another striking example of the problems associated with analyzing message-passing code is described in Section 3.

### 2.4 Optimization

In order to illustrate the use of contention modeling in optimization problems in this section we discuss a simple example that shows how the issue is addressed in the case of vectorization.

Consider the following scalar loop

```
forall (i = 1 .. N) x[i] = a * x[i];
```

The vectorization decision typically depends on whether $N$ is long enough to sufficiently amortize the startup overhead. In the following we will derive this decision. Ignoring loop overhead and memory traffic for simplicity, the operation can be modeled by

$$L = \mathbf{par}\ (i = 1, N)\ flop$$

where $flop$ models the floating point multiplication.

In our process-algebraic approach towards describing algorithms and machines, execution on a *scalar* processor is modeled by the equation

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

where $\tau_f$ denotes the instruction delay of the scalar floating point unit $s$. By substitution it follows

$$L_s = \mathbf{par}\ (i = 1, N)\ \mathbf{use}(s, \tau_f)$$

where the $s$ subscript denotes the scalar design alternative. The estimation algorithm compiles $L_s$ directly into $T_s = N\tau_f$ which effectively models the sequential execution resulting from the contention principle.

On the other hand, execution on a *vector* processor would be modeled by using an $S$ stage pipeline (see Section 2.2) according to

$$flop = \mathbf{seq}\ (s = 1, S)\ \mathbf{use}(u_s, \tau_c)$$

where $u_s$ represents stage $s$ of the vector unit and $\tau_c$ denotes the cycle time[4]. It follows

$$L_v = \mathbf{par}\ (i = 1, N)\ \mathbf{seq}\ (s = 1, S)\ \mathbf{use}(u_s, \tau_c)$$

where the $v$ subscript denotes the vectorization alternative. The estimation algorithm generates $\tau_s + N\tau_c$ where $\tau_s = (S-1)\tau_c$ denotes the startup cost [9]. Consequently, the (boolean) vectorization decision becomes $T_v < T_s$ which equals $\tau_s + N\tau_c < N\tau_f$. Because of the fact that our cost modeling approach produces closed-form expressions, this comparison can be reduced at the time the compiler optimization module itself is being compiled for a specific target architecture. It follows

$$T_v < T_s = \ldots = N \geq \left\lceil \frac{\tau_s}{\tau_f - \tau_c} \right\rceil = N \geq N^*$$

where $N^*$ is the cross-over value usually hard-wired in vectorizing compilers.

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, merely by discussing alternative $flop$ machine models while using the *same* algorithmic description. (Note that an explicit *sequential implementation* of the algorithm in the scalar case has *not* been considered.) Second, it shows how the optimization problem is addressed in terms of low-complexity cost models. If $N$ is known at compile-time, the optimization logic (in this simple case comprising the integer test $N \geq N_v$) can be numerically evaluated. Even when $N$ is only known symbolically, the logic can be still compiled and evaluated at run-time at negligible expense. The decision to make at compile-time merely reduces to the question if it is worth-while to compile the additional integer test $N \geq N^*$ as part of the target code.

In general, as long as the optimization test complexity is small compared to the optimization yield (i.e., the task complexity), it pays off to compile a partial decision tree as part of the generated code. Except for fine-grain operations this relative difference is typically at least an order of magnitude, as illustrated by the above example as well as the future examples in the paper. For brevity, the above discussion did not include the performance effects of the memory system (e.g., memory pipelines based on parallel memory bank access). However, we stress the fact that this in no way complicates the optimization scheme other than adding some syntactic complexity.

## 3 Case Study

In this section we will study a somewhat more complex distributed-memory application kernel in which we derive optimization logic for the optimal data partitioning which involves a trade-off between remapping and processor pipelining. The application kernel is a simplified ADI line relaxation algorithm fragment of which the optimal implementation has been described at length in [1]. We derive the same solutions, yet in terms of our generic cost estimation framework.

[4] 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. However, the above linear 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 $\tau_c$ satisfy (fit) the performance measurements.

### 3.1 Machine-level Modeling

Before deriving our optimization solutions, we will first demonstrate the problems associated with the choice of *machine* level as the basis for performance feedback. Consider the simplified line relaxation algorithm fragment [1] 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 (followed by a sweep in the $j$ direction which is considered later on). In the parallelization for a $P$ processor distributed-memory machine we shall 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 from a human point of view, see Fig. 2).

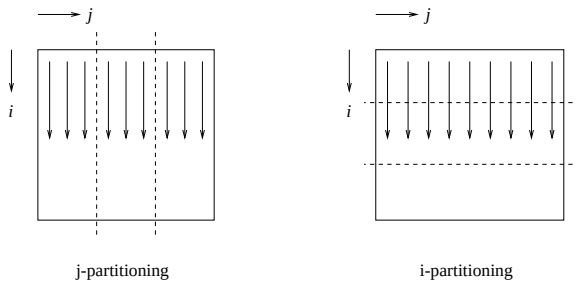


Figure 2: Partitioning choices, vertical phase.

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 resulting 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 ($U(p) - L(p) = \mathcal{O}(N/P)$). Note that the original (shared) data indexing is used for readability.

The above code clearly reveals the speedup potential. While this might also seem for the alternative partitioning, the $i$ axis partitioning not yield any speedup in reality. The SPMD code is characterized by the following pseudo code (following the usual "owner-computes" convention including a number of trivial index optimizations)

```
if p > 0
   send(p-1,A[L(p)][:]);
if p > 0                ! get A[L(p)-1][:]
   recv(p-1,tmp_l);
else
   tmp_l = A[0][:];
if p < P                ! get A[U(p)+1][:]
   recv(p+1,tmp_u);
else
   tmp_u = A[N-1][:];
for i = L(p) .. U(p) { ! update partition
    if i = L(p)
       for j = 0 .. N-1
           A[i][j] = tmp_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]+tmp_u[j];
}
if p < P
   send(p+1,A[U(p)][:]);
```

Again, note that the code is expressed in terms of the original (shared) data structure for readability. In this example, we will ignore the fact that the $j$ loops can be vectorized.

While the previous SPMD code can be easily compiled into a symbolic performance model, the above SPMD code illustrates the difficulties involved when compiling generated SPMD code into a performance model. Although the local bounds on the $i$ loop are also reduced by a factor $P$ as in the $j$-partitioning, the message-passing scheme totally *serializes* the entire computation. As discussed in Section 2.3, however, especially in a symbolic analysis it is generally hard to deduce that the the critical path now runs through each SPMD process.

### 3.2 Program-level Modeling

Clearly, the critical path that is obscured in the above message-passing implementation is nothing but the result of the explicit sequential $i$ loop at program level. Consequently, it is much more advantageous to consider a modeling approach at *algorithm* level (program level) than at *implementation* level (machine 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, we will used the same

"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 vertical phase 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

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

where $\tau_f$ denotes the computation time associated with the update of $A_{ij}$. Note, that this represents a "processor contention model".

Based on this model, the evaluation of both partitionings is straightforward. For the $j$ axis partitioning $\mu(i,j) = j/B$ where $B = N/P$ denotes the block size (for simplicity assume $P|N$). It follows

$$\textbf{seq}\ (i = 1, N-2)\ \textbf{par}\ (j = 0, N-1)\ \textbf{use}(cpu_{j/B}, \tau_f)$$

Application of our mechanical cost estimation technique yields (after a few intermediate symbolic reductions, described in [9]) $T = (N-2)N\tau_f/P$ corresponding to the speedup found earlier.

For the $i$ axis partitioning $\mu(i,j) = i/B$. It follows

$$\textbf{seq}\ (i = 1, N-2)\ \textbf{par}\ (j = 0, N-1)\ \textbf{use}(cpu_{i/B}, \tau_f)$$

which, in contrast to the actual SPMD code, directly reveals the algorithm's sequential nature. Indeed, the cost estimate directly compiles to $T = (N-2)N\tau_f$. Thus by exploiting the algorithm's inherent sequential semantics present in its description, from a simple cost estimation it directly follows that an $i$ axis partitioning will not yield any speedup[5]. 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 require simulation.

### 3.3 Remapping

The choice of modeling at program level for the purpose of optimization makes it easy to reason about much more optimizations than just the choice of partitioning and/or vectorization. In the following we will consider the possibility of remapping between both phases of the ADI algorithm corresponding to the example discussed in [1]. In the first phase the line relaxation is swept along the $i$ axis, after which the relaxation is swept along the $j$ axis. The two-phase algorithm is given by

```
for i = 1 .. N-2   ! ver. phase
    forall j = 0 .. N-1
        A[i][j] = A[i-1][j] + A[i+1][j];
for j = 1 .. N-2   ! hor. phase
    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 and let $\mu'(i,j) = i/B$ denote the layout after remapping. Let $L_r$ denote the PAMELA model of the remapping procedure. Given $L_v$, $L_r$, and $L_h$, the mapping decision involves the comparison $T_v(\mu) + T_r + T_h(\mu') < T_h(\mu)$ where $T_x$ denotes the lower bound time estimate of $L_x$, $x \in \{v,h,r\}$. In the following we will derive the optimization logic. To avoid describing unnecessary complicated expressions we will use order terms only.

We start with $L_v$. In terms of PAMELA the vertical sweep phase is modeled by

$$\begin{array}{l}\textbf{seq}\ (i = 1, N-2)\\ \quad \textbf{par}\ (j = 0, N-1)\ \{\\ \quad\quad move(\mu(i-1,j), \mu(i,j));\\ \quad\quad move(\mu(i+1,j), \mu(i,j));\\ \quad\quad flop(\mu(i,j))\\ \quad \}\end{array}$$

where $flop$ is defined as before and $move(s,r)$ represents the (scalar) data transfer from sending processor $s$ to receiving processor $r$ where the update takes place (i.e., $r = \mu(i,j)$). Thus 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). In the following we will only account for non-local data transfers. Let

$$move(s,r) = \textbf{use}(l_s, [s \neq r]\tau_m) \parallel \textbf{use}(l_r, [s \neq r]\tau_m)$$

where $l$ models the communication link (or service) that is needed by a processor to communicate non-locally, and where the $[\ldots]$ construct denotes Iverson's operator defined by $[false] = 0$ and $[true] = 1$. Thus data movement charges workload to communication resources at both sender and receiver. Note that this is a fairly abstract model. However, it does account for

[5] Later on, we will consider a modified version of the algorithm in which $i$ axis partitioning does yield speedup.

the fact that a node can neither send nor receive data in parallel without (proportionally) increasing the associated time cost. As even this simple (bandwidth) model can already introduce a potential bottleneck we refrain from modeling other sources of network contention (see [9] for more detailed communication models). Note that we also assume that any additional condition synchronization (e.g., due to the message-passing *implementation*) is negligible as a result of low-level compiler optimizations. (Considering the fact that the application-level source of sequentialization is already accounted for, this is a safe assumption. This point will be further discussed at the end of this section.)

As $\mu(i,j) = j/B$ for the vertical phase $L_v$ it follows by substitution

$$
\begin{array}{l}
\textbf{seq}\ (i = 1, N-2) \\
\quad \textbf{par}\ (j = 0, N-1)\ \{ \\
\qquad \{\ \textbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\ \|\ \\
\qquad\ \ \textbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\ \}; \\
\qquad \{\ \textbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\ \|\ \\
\qquad\ \ \textbf{use}(l_{j/B}, [j/B \neq j/B]\tau_m)\ \}; \\
\qquad \textbf{use}(cpu_{j/B}, \tau_f) \\
\quad \}
\end{array}
$$

which simply yields $T_v = \mathcal{O}(N^2/P)\tau_f$. The horizontal phase is modeled by

$$
\begin{array}{l}
\textbf{seq}\ (j = 1, N-2) \\
\quad \textbf{par}\ (i = 0, N-1)\ \{ \\
\qquad move(\mu(i,j-1), \mu(i,j)); \\
\qquad move(\mu(i,j+1), \mu(i,j)); \\
\qquad flop(\mu(i,j)) \\
\quad \}
\end{array}
$$

which equals

$$
\begin{array}{l}
\textbf{seq}\ (j = 1, N-2) \\
\quad \textbf{par}\ (i = 0, N-1)\ \{ \\
\qquad \{\ \textbf{use}(l_{(j-1)/B}, [(j-1)/B \neq j/B]\tau_m)\ \|\ \\
\qquad\ \ \textbf{use}(l_{j/B}, [(j-1)/B \neq j/B]\tau_m)\ \}\ ; \\
\qquad \{\ \textbf{use}(l_{(j+1)/B}, [(j+1)/B \neq j/B]\tau_m)\ \|\ \\
\qquad\ \ \textbf{use}(l_{j/B}, [(j+1)/B \neq j/B]\tau_m)\ \}\ ; \\
\qquad \textbf{use}(cpu_{j/B}, \tau_f) \\
\quad \}
\end{array}
$$

which generates $T_h = \mathcal{O}(PN)\tau_m + \mathcal{O}(N^2)\tau_f$ (see [9]). Clearly, it is interesting to consider remapping. Remapping $A$ implies a transposition $L_r$ modeled by

$$\textbf{par}\ (i = 0, N-1)\ \textbf{par}\ (j = 0, N-1)\ move(\mu_{i,j}, \mu_{j,i})$$

Given the mapping $\mu(i,j) = j/B$ it follows

$$
\begin{array}{l}
\textbf{par}\ (i = 0, N-1) \\
\quad \textbf{par}\ (j = 0, N-1) \\
\qquad \{\ \textbf{use}(l_{j/B}, [j/B \neq i/B]\tau_m)\ \|\ \\
\qquad\ \ \textbf{use}(l_{i/B}, [j/B \neq i/B]\tau_m)\}
\end{array}
$$

which yields $T_r = \mathcal{O}(N^2/P)\tau_m$. As $T_h(\mu') = T_v(\mu)$, the optimization logic is given by

$$\mathcal{O}(\frac{N^2}{P})\tau_m + \mathcal{O}(\frac{N^2}{P})\tau_f < \mathcal{O}(PN)\tau_m + \mathcal{O}(N^2)\tau_f$$

Clearly there exist values for $P$, $\tau_f$, and $\tau_m$ for which remapping is justified.

### 3.4 Pipelining

Thus far, we have considered remapping while assuming that the same algorithm would have to be used for the horizontal phase as for the vertical phase. However, for the horizontal phase the algorithm can be optimized by a *loop exchange* which 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 [1]. Unlike in the remapping case, we ignore data communication for simplicity. 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 $L_h$ according to

$$\textbf{seq}\ (j = 1, N-2)\ \textbf{par}\ (i = 0, N-1)\ \textbf{use}(cpu_{j/B}, \tau_f)$$

For the purpose of a future algorithm transformation we will consider the following equivalent model

$$\textbf{seq}\ (p = 0, P-1)\ \textbf{par}\ (i = 0, N-1)\ \textbf{use}(cpu_p, B\tau_f)$$

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 expressed in a single **use** statement). As discussed earlier, with the original mapping it follows $T_h = (N-2)N\tau_f$. The algorithm transformation we consider is *exchanging* the $i$ and $j$ loop, in terms of the PAMELA model $L_{h'}$ according to

$$\textbf{par}\ (i = 0, N-1)\ \textbf{seq}\ (p = 0, P-1)\ \textbf{use}(cpu_p, B\tau_f)$$

Note that this produces a pipeline like in the examples discussed earlier (see Fig. 3). Indeed, instead of running sequential, each $i$ loop is now pipelined such that the next processor executes a different $i$ loop instance

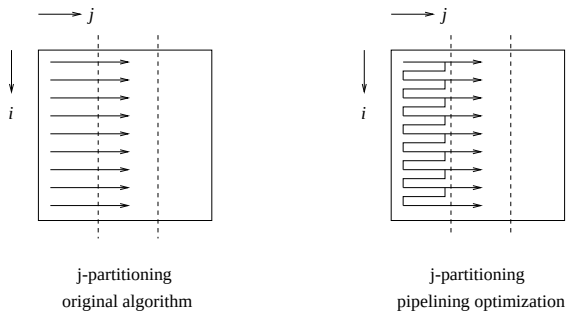


Figure 3: Original partitioning vs. pipelining in horizontal phase.

concurrently, yet obeying the $j$ sequence [1]. 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). The reason for the slight modification to the original model $L_h$ is that a loop exchange in this model, i.e.,

$$\textbf{par}\ (i = 0, N-1)\ \textbf{seq}\ (j = 1, N-2)\ \textbf{use}(cpu_{j/B}, \tau_f)$$

actually would not model the pipelining behavior because of the scheduling fairness of the **use** statement[6]. Applying the transformation on the modified model, however, does produce the intended schedule.

The loop exchange has a great impact on performance as it immediately follows $T_{h'} = \mathcal{O}(N^2/P)$. Thus, by loop exchange, the same (order) of performance can be achieved as in the first phase, without remapping. The optimization logic is given by $\mathcal{O}(N^2/P)\tau_f < \mathcal{O}(N^2)\tau_f$ which is reduced at compile-time in favor of processor pipelining. Of course, $L_{h'}$ ignores the additional pipeline startup delay as well the additional communication overhead as some communication is still required. Especially when multiple sweeps are performed in both phases remapping may still be appropriate (see [9] for more details).

The line relaxation case study illustrates the advantage of the contention modeling approach when applied at program level in order to express compile-time (i.e., symbolic) optimizations. The analysis permits the various optimization decisions to be expressed in terms of an overall symbolic expression that can be reduced considerably before being solved (by, e.g., 0-1 integer programming [12]). In many cases (like in the above examples) low-cost solutions can be derived which implies that the optimization logic can be compiled and evaluated at run-time.

Of course, there are cases where the choice to abstract from the actual implementation may induce a considerable error as the low-level code generation model is not included in the cost model. For instance, a naive message-passing code generation model could completely sequentialize an inherently parallel operation[7]. Clearly, the above prediction method will not take this into account (although inherent sequentializations at *algorithmic* level are accounted for, of course). As mentioned earlier, however, a code generation model should be assumed that does not introduce these extremely pathological schedules (note that a simple inspector/executor implementation already solves the problem just mentioned in the footnote).

Of course, as mentioned in Section 2, the symbolic cost estimation technique itself introduces a certain *in*accuracy with respect to which optimization will actually yield better performance. On the other hand, its limited inaccuracy [8] guarantees that if a wrong optimization is applied, the performance decrease can be expected to be limited as well. One should realize 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. In this context, a cost estimation technique that guarantees the correct, order-type of prediction can be considered appropriate[8].

## 4 Conclusion

In this paper we have studied the trade-off between modeling at program level and machine level in the context of automatic performance optimization for message-passing architectures. Despite the abstractions involved, we have shown that program level modeling is to be preferred for compile-time cost estimation provided this method is used in conjunction with a modeling technique called "contention modeling". In this technique, the parallelism (or sequentialism) implicitly or explicitly present at program level is expressed in terms of parallel (or sequential) processes

[6] Because of the conflict arbitration fairness, the $(i, j)$ accesses would still be scheduled according to a column-major scheme rather than the intended row-major scheme. Thus, the model would effectively execute in the same order as the original model (without loop reversal).

[7] 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 will completely (and unnecessarily) sequentialize the computation.

[8] Although the cost estimation technique may introduce a (limited) prediction error, it essentially includes all the right order terms in the expression because of the fact that it accounts for *all* contention effects, be it at program level (e.g., critical sections) or machine level (e.g., link, memory, disk contention).

while the effect of machine implementation choices is modeled by introducing resource contention. We outlined this modeling technique in terms of the PAMELA language, which is specifically tailored to this way of modeling, and which automatically produces low-cost, symbolic time cost expressions that guide the optimization process. We have demonstrated the effectiveness of our modeling approach by deriving various optimizations of a line relaxation kernel for distributed-memory machines, in which we have also shown that the message-passing machine level is generally unsuitable as the basis for symbolic cost estimation.

## References


[1] 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*, Fall 1994, pp. 48–58.

[2] G.R. Andrews and F.B. Schneider, "Concepts and notations for concurrent programming," *Computing Surveys*, vol. 266, no. 24, 1983, pp. 132–145.

[3] D. Atapattu and D. Gannon, "Building analytical models into an interactive prediction tool," in *Proc. Supercomputing '89*, pp. 521–530.

[4] V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, "A static performance estimator to guide data partioning decisions," in *Proc. 3rd ACM SIGPLAN Symp. on PPoPP*, Apr. 1991.

[5] M.J. Clement and M.J. Quinn, "Multivariate statistical techniques for parallel performance prediction," in *Proc. 28th HICSS, Vol. II*, IEEE, Jan. 1995, pp. 446–455.

[6] T. Fahringer and H.P. Zima, "A static parameter-based performance prediction tool for parallel programs," in *Proc. ACM ICS'93*, Tokyo, pp. 207–219.

[7] A.J.C. van Gemund, "Compiling performance models from parallel programs," in *Proc. ACM ICS'94*, Manchester, pp. 303–312.

[8] A.J.C. van Gemund, "Compile-time performance prediction of parallel systems," in *Computer Performance Evaluation: Modelling Techniques and Tools* (LNCS 977), Sept. 1995, pp. 299–313.

[9] A.J.C. van Gemund, *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, The Netherlands, Apr. 1996.

[10] M. Gupta and P. Banerjee, "Compile-time estimation of communication costs of programs," in *Proc. Second Workshop on Automatic Data Layout and Performance Prediction (CRPC-TR95548)*, Rice University, Houston, Apr. 1995.

[11] C.A.R. Hoare, "Communicating Sequential Processes," *CACM*, vol. 21, Aug. 1978, pp. 666–677.

[12] 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.

[13] W. Kreutzer, *System simulation, programming styles and languages*. Addison-Wesley, 1986.

[14] S.S. Lavenberg, *Computer Performance Modeling Handbook*. Academic Press, 1983.

[15] 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'94*, May 1994, pp. 181–189.

[16] 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 (CRPC-TR95548)*, Rice University, Houston, Apr. 1995.

[17] 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. ACM SIGMETRICS'93*, May 1993, pp. 48–60.

[18] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, 1989.

[19] A. Sivasubramaniam, A. Singla, U. Ramachandran and H. Venkateswaran, "An approach to scalability of shared memory parallel systems," in *Proc. ACM SIGMETRICS'94*, May 1994, pp. 171–180.

[20] B. Stramm and F. Berman, "Predicting the performance of large programs on scalable multicomputers," in *Scalable HPC Conf.*, 1992, pp. 22–29.

[21] K-Y. Wang, "Precise compile-time performance prediction for superscalar-based computers," in *Proc. PLDI'94*, Orlando, June 1994, pp. 73–84.