# Automated Performance Prediction of Message-Passing Parallel Programs

Robert J. Block*, Sekhar Sarukkai**, Pankaj Mehra**

* Department of Computer Science
University of Illinois
Urbana, IL 61801
rblock@ncsa.uiuc.edu

** Recom Technologies
NASA Ames Research Center
Moffett Field, CA 94035-1000
{sekhar,mehra}@ptolemy.arc.nasa.gov

**Keywords:** performance analysis, performance prediction, scalability analysis, parallel program performance, automated modeling, performance tools, performance debugging

**Abstract**

The increasing use of massively parallel supercomputers to solve large-scale scientific problems has generated a need for tools that can predict scalability trends of applications written for these machines. Much work has been done to create simple models that represent important characteristics of parallel programs, such as latency, network contention, and communication volume. But many of these methods still require substantial manual effort to represent an application in the model's format. The MK toolkit described in this paper is the result of an on-going effort to automate the formation of analytic expressions of program execution time, with a minimum of programmer assistance.

In this paper we demonstrate the feasibility of our approach, by extending previous work to detect and model communication patterns automatically, with and without overlapped computations. The predictions derived from these models agree, within reasonable limits, with execution times of programs measured on the Intel iPSC/860 and Paragon. Further, we demonstrate the use of MK in selecting optimal computational grain size and studying various scalability metrics.

# 1 Introduction

With the common use of powerful microprocessors as building blocks for high performance computing and with the imminent use of high-speed interconnection networks (local or wide) to communicate among these processors, network characteristics and processor configurations significantly effect delivered program performance. During program development, it is impractical to reserve hundreds of processors for each round of debugging and performance tuning or to utilize the networking resources that may be at ones disposal during large-scale executions. Typically, only small-scale runs (involving, say, up to 32 processors) are feasible. Programmers need tools that will rapidly (and with sufficient accuracy) characterize large-scale performance of production runs using only the information available from small-scale test runs. The development of such tools involves **performance prediction** and **scalability analysis**, which seek to characterize, respectively, a program's completion time and its asymptotic speedup as functions of problem size (N) and the number of processors (P).

The performance of parallel programs on parallel and distributed environments can be characterized in terms of their computation and communication complexities. The complexity of computation is determined largely by the activities that occur within each processor whereas that of communication, by the message-transmission time between processors. It therefore makes sense to make a distinction between computation phases and communication phases. Frequently, however, computation and communication overlap; in that case, the complexities of computation and communication cannot be considered separately. Therefore, modeling of message-passing programs requires a representation scheme that not only allows us to model the computation and communication phases but also captures the relationships among them.

In comparison with data-parallel paradigms, the communication patterns and interprocessor data dependencies of message-passing parallel programs are not as evident to the compiler or from examination of source code; these factors complicate the modeling and analysis of communication complexities. It is perhaps because of this difficulty that there are no automated tools for performance prediction and scalability analysis of message-passing programs. There are, however, a number of tools for profiling communication activity and visualizing it post mortem [5, 12]. The problem is that each profile is tied to particular (typically small) values of N and P, whereas one needs to characterize complexity as a function of N and P. One of the key problems in performance prediction and scalability analysis, then, is the inference of scale-independent communication patterns from communication pro-

files of small-scale runs.

The initiation of the project to generate automated models of programs was motivated by our inability to hand-build models of realistic parallel CFD applications in reasonable time. Generating models of parallel programs could be very time consuming and possibly more complex than the actual coding of the application, leading to little or no use of model analysis tools. In this paper we present our approach in providing a first step towards automatically generating performance models of message-passing parallel programs and its application in accurately predicting execution characteristics of a number of different applications.

The three key issues in building automated tools for performance prediction and scalability analysis are: (i) choice of a scheme for representing a program's control structure and communication patterns; (ii) development of algorithms for automated model building using only the information available either in the source code or in execution profiles from small-scale runs; and (iii) development of algorithms for model-based analysis of complexity. This paper describes our approach to these issues in the context of MK (Modeling Kernel), a tool for automated modeling of message-passing programs.

MK uses two key data structures for representing models: Augmented Parse Trees (APTs) and Communication Phase Graphs (CPGs). Briefly, an APT captures information about the syntactic control constructs of interest for complexity analysis (subroutines, loops, branches and communication calls) and augments this parse-tree-like structure with symbolic information about loop bounds, branch frequencies and instruction counts/timings. Also briefly, a CPG represents an application's communication pattern as a collection of phases; each phase includes symbolic information about participating processors and about APT constructs whose computation might overlap with the communications in that phase.

MK makes extensive use of two existing toolkits during model building. It uses the Sage compiler front-end toolkit [4] to analyze the program's control and data flows, and to insert instrumentation for collecting certain information not available at compile time. It uses the Automated Instrumentation and Monitoring System **(AIMS)** tool [12] to gather small-scale execution profiles containing a variety of information: timing information about basic blocks, sources and destinations of messages, and data generated by the aforementioned instrumentation. MK then converts the numeric information from many different small-scale execution profiles, each for a different value of N and P, into symbolic expressions in N and P. In order to accomplish this, it makes use of

statistical regression.

The final component of MK is an analysis layer that employs well-known techniques for analysis of iteration spaces for determination of computation complexity. The analysis of communication begins by characterizing individual phases as either pipelined or loosely synchronous; the appropriate complexity expression is then generated. The analysis proceeds bottom up in the APT; when a node contains more than one communication phase, MK determines whether those phases could have been overlapped. Overall complexity is thus calculated by appropriately combining the complexities of lower-level nodes.

From a source code analysis perspective, programs with data-dependent loop bounds and/or message sizes are hard to analyze. Consequently, only direct solvers are completely amenable to automated analysis using MK. However, with some user intervention that specifies data-dependencies or the number of iterations needed to converge, iterative solvers can be tackled in almost the same manner as direct solvers. High accuracy of prediction is neither achievable nor sought by MK, because accurate prediction requires detailed modeling of memory accesses and cache behavior, both of which are difficult to characterize in a scale-independent fashion and are time-consuming to simulate.

Its model-building stage distinguishes MK from other research in performance prediction [1, 3, 2]. While MK automatically extracts communication characteristics from a message-passing parallel program, the performance-prediction systems described in [1, 3] either require manual specification of a communication model or assume that the program is described in a data-parallel language. Crovella [2] on the other hand, describes an approach for shared-memory systems and starts from default models of performance predicates. Another unique contribution of our work is a systematic, language- and scale-independent representation of parallel-program models – namely, APTs and CPGs – which can support a variety of analysis functions.

MK is an ongoing project that currently handles only data-independent, deterministic computations executed in a homogeneous, distributed environment. A natural extension of the toolkit towards a more heterogeneous environment, considering a dynamically changing network load, is under investigation. The modules that make-up MK are flexible enough to answer a number of "what-if?" questions with minimal user effort. Any detail of model not yet incorporated into MK can easily be added, since MK provides a framework (including the APT and CPG) that can be easily augmented with any level of detail.

The rest of this paper is organized as follows. Section 2 provides an

overview of MK's operation and describes the construction and annotation of augmented parse trees. Sections 3 and 4 explain how communication phase graphs are built and used in complexity analysis. Section 5 describes our applications and presents results of MK's automatic analysis in generating models of their execution times. Section 6 concludes the paper.

# 2 MK: Modeling Kernel

MK combines static compiler-assisted analysis, execution profiling and statistical regression to obtain a closed-form analytic expression for a parallel program's execution time in terms of its problem size (N), and the number of processors (P). The goal is to capture the salient execution characteristics of a program without modeling every detail. (The problem size may in fact be determined by multiple input parameters, and MK allows any number of parameters to be used, but we will refer to a single variable N for simplicity.)

Several features of MK distinguish it from other prediction tools and make its use largely automatic:

- Automatic derivation of model structure from program structure: A program dependence graph (PDG) is generated using the Sage compiler front-end toolkit [4]. MK then selects the subgraph of the PDG induced by the inclusion of major control points in the program - such as loops, function headers and calls to subroutines - to form the skeleton of an **augmented parse tree (APT)** of the program. The APT's nodes are augmented with performance-related information useful for both analysis and simulation [8] of large-scale performance.

- Use of static source-level analysis to identify program variables affecting performance: It is difficult to infer the statistical relationship between the execution time for a complete program and the independent variables (N and P) due to the presence of many independent terms. On the other hand, characterizing the values of individual variables, such as loop bounds and message lengths, is usually straightforward because these variables are typically affine functions of N and P.

- Detection and evaluation of communication phases using trace analysis and domain-based graph partitioning: Careful analysis

of communication domains based on execution traces and trace-based determination of computations overlapping communication are the key to construction of communication phase graphs (CPGs), which identify the different communication phases and contain a variety of information about the participating processors and their pattern of communication.

- Generation of complexity and scalability expressions: Expression for program completion time is derived via a bottom-up traversal of the APT. The expression can be used to compute data rapidly for a variety of scalability metrics.

## 2.1 APT Generation

A comprehensive representation of the program's control and data-dependence graphs is essential for modeling program behavior. MK uses APTs, which consist of a subset of nodes derived from the parse tree of a program. Regions of code corresponding to a node in the parse tree can be analyzed by considering the corresponding node in the APT.

Each node in the APT is of type {ROUTINE,IF,LOOP,CALL,COMMN}. Once the APT skeleton has been extracted from the Sigma program graph, its nodes are annotated with attributes related to the program's execution time. The attributes include loop bounds, branch probabilities, message sizes, and basic-block execution times. In subsequent stages, nodes are augmented with communication-phase information that is used to determine the program's overall complexity.

To express the sizes of loop bounds and messages in terms of the program's input parameters, MK uses flow analysis to perform constant propagation and variable substitution wherever possible, and prompts the user to supply the symbolic expressions corresponding to the remaining variables. In many cases, aggressive analysis of the program's def-use chain can drastically reduce the amount of user input required. Evaluation of branch probabilities is done by post-mortem trace analysis. To reduce overhead, an **IF**-node is analyzed only if at least one of its branch subtrees contains additional control or communication nodes.

In a program APT **T**, a **computation phase** is a subtree of **T** rooted at node **u**, type(**u**) = LOOP, containing no communication nodes. By timing these loop bodies and determining their loop bounds, MK derives first-order estimates of computation complexities. The static analysis component of MK currently recognizes rectangular and triangular itera-

tion spaces only; it is being enhanced to analyze loops with more general structures and dependencies. The most important factors contributing to a parallel program's first-order execution time are the complexities of its computation and communication phases, and the amount of overlap between the two. The next two sections discuss how communication phase information is automatically inferred by MK using CPGs, and inserted into the APT to complete the analysis.

# 3 Modeling of Communication Phases

As noted in Section 1, an important step in modeling message-passing parallel programs is the recognition of communication phases. There is a list of important constraints under which our approach for automatically recognizing communication phases should work:

- There is no prior knowledge of the communication pattern generated.
- Recognized communication phases are architecture- and source syntax-independent.
- Communication phases can overlap with each other.
- Access to a global clock will not be assumed.

In this section we define an algorithm that meets the above constraints and show how this information augments the APT, towards use by model analysis. A communication phase can be defined as a set of inter-related communication operations that act in concert to transfer data across a set of processors. These phases can take many specific forms in an application, but they can be roughly divided into two categories: **synchronous** and **asynchronous**. Synchronous phases typically involve **exchanges** of data, such as boundary communication patterns, but also include shift patterns (see figure 1.) Asynchronous phases more commonly involve a **directed flow** of data such as pipeline or tree patterns.

The complexity of these phases is straightforward to model analytically, if one knows ahead of time where to look for them in the application. However, recognizing them automatically is complicated by the variety of forms that even a simple pattern can take in the program graph, and by the subtle lexical differences that may distinguish two phases with radically different execution times. As noted earlier, the fundamental data structure in MK that contains information about communication

patterns is the communication phase graph (CPG). We describe the information contained in CPGs and how it can be inferred from execution traces.

Since communication patterns cannot be determined by static analysis of programs, some run-time information is necessary to form a CPG. Execution traces for CPG construction are obtained using the Automated Instrumentation and Monitoring System (AIMS) toolkit. A trace is a time-ordered sequence of events generated during execution as the thread of control passes through instrumented program constructs. By default, MK instruments all communication points and all APT constructs (subroutines, loops and branches). Each event **e** contains information about its time of occurrence **t(e)**, the processor on which it occurred **p1(e)**, and an identifier **c(e)** for the instrumented APT construct whose execution caused that event. If **e** is a send or receive event, it also contains the other processor involved **p2(e)**, and the specified message tag **m(e)**.

The inputs to the CPG construction algorithm are the APT, containing the program's communication call points, and a trace file containing the send and receive events from one execution. The trace file is initially sorted by event-time. The complete algorithm is listed in figure 2; its steps are summarized here:

1. Read the trace file, find matching send and receive events, and add communication edges between the corresponding call points in the APT.
2. Identify communication phases as connected components (with communication edges) of the resulting graph.
3. In a second pass through the trace, record which processors send and receive messages in each phase, and in what order.
4. Compute the APT ancestor of each communication phase.
5. Identify each phase as synchronous or asynchronous, and for pipeline phases, compute the length of the longest processor chain.

The following data structures are associated with each phase **g** created in step 2 above:

- **SL**(g): list of processors that send messages in a phase
- **RL**(g): list of processors that receive messages in a phase
- **pairs**(g): time-ordered list of send-receive endpoints

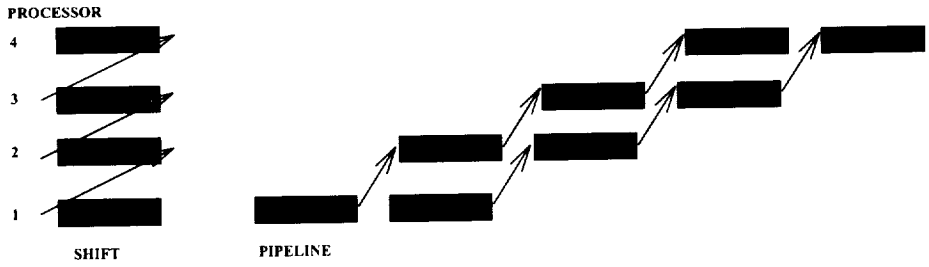


Figure 1: Two illustrative communication patterns.

- **async**(g): flag indicating whether receive was posted before send

We illustrate the algorithm by comparing two different communication patterns with similar lexical structures. Consider the two code skeletons below: a loosely synchronous shift-right communication pattern, and a pipeline, for the same set of 4 processors.

**Shift pattern**

**do** $i = 1$ to $n$
  if (me $<$ 4) send(me+1)
  **compute**
  if (me $>$ 1) recv(me-1)
**enddo**

**Pipeline pattern**

**do** $i = 1$ to $n$
  if (me $>$ 1) recv(me-1)
  **compute**
  if (me $<$ 4) send(me+1)
**enddo**

Figure 1 illustrates the communication patterns. In both cases, processor 1 sends data to processor 2, 2 to 3, and 3 to 4. The send list in each case is {1,2,3} and the receive list, {2,3,4}. The crucial difference between the two patterns is captured by the ordering of communication events on processors 2 and 3. In the shift pattern, each processor can send data before it receives data from another processor. In the pipeline, processors 2, 3 and 4 must wait for data from their predecessors before passing data onto their successors. Step 3 of the algorithm catches the receive-before-send sequence in the pipeline, marks it as an asynchronous phase, and computes that its chain depth is 3. The shift pattern is categorized as loosely synchronous, by default.

The next section shows how the time complexity of these communication phases can easily be characterized once we can distinguish pipelines from loosely synchronous patterns.

# 4 Automated Scalability Analysis using APTs and CPGs

In this section, we will show how the complexity expression for a message-passing program is generated from its APT and CPG. We assume that the component expressions for computation phases are known, and demonstrate how analogous expressions can be derived for the communication phases that were identified and classified during the CPG construction.

The complexity $T(V)$ of APT node $V$, is defined recursively in terms of its children's complexities, and any communication phases anchored at $V$. $T(V)$ has two components:

$$T(V) = T_s(V) + T_p(V),$$

where $T_s$ is the complexity exclusive of communication phases, and $T_p$ is the aggregated phase time. (For nodes that are not phase ancestors, $T(V) = T_s(V)$.) The method for computing $T_s$ was discussed in [6], and is summarized in the table below, for the various APT node types.

| $type(V)$ | $T_s(V)$ |
|---|---|
| **ROUTINE** | sum of children's complexities |
| **IF** | sum of children's complexities, weighted by their branch probabilities |
| **LOOP** | sum of children's complexities, multiplied by the width of the loop |
| **CALL** | complexity of called routine |
| **GLOBAL** | global communication cost (machine specific) |

An important point to note is that the ancestor node of a communication phase may contain computation, as well as communication, in its subtree (see figure 3,) and the algorithm must determine how much computation can overlap in time with communication.

## 4.1 Communication Phases

If $V$ is a phase ancestor, the time component $T_p(V)$ must account for overlap between successive phases. Two phases $g_i$ and $g_{i+1}$ are **depen-**

Let $v$ be the set of all communication points in the program.
Let $G(V, E)$ be the CPG of the program.
Let $Q$ be a queue of send/receive events.
Initialize $G$: $V \leftarrow v, E \leftarrow \{\}$.

**Step 1: (First pass through trace)**

Initialize $Q \leftarrow \{\}$.
for each event $e_i$
    Search $Q$ for prior matching event $e_j$, such that
        $m(e_i) = m(e_j), p1(e_i) = p2(e_j), p2(e_i) = p1(e_j)$
    if $e_j$ found, Add edge $(c(e_i), c(e_j))$ to $G$
    else add $e_i$ to $Q$

**Step 2:**

**Partition $G$ into connected components $g_1 \ldots g_n$.**

For each $g$, initialize $SL(g), RL(g), pairs(g) \leftarrow \{\}$.

**Step 3: (Second pass through trace)**

For each event $e_i$
    Find $g_k$ containing $c(e_i)$
    Set $p \leftarrow p1(e_i)$ (processor issuing event)
    if $c(e_i)$ is **SEND**, and $p \notin SL(g_k)$,
        and $p \in RL(g_k)$, set $async(g_k) \leftarrow true$.
    Add $p1(e_i)$ to $SL(g_k)$ or $RL(g_k)$, depending
        on whether $c(e_i)$ is **SEND** or **RECV**.
    Add $(p1(e_i), p2(e_i))$ to $pairs(g_k)$.

**Step 4:**

For $k = 1$ to $n$ do
    Find APT node $Anc(g_k)$ which is nearest common ancestor of
        all call points in $g_k$

**Step 5:**

For $k = 1$ to $n$ do
    if $async(g_k) = true$, mark this phase as a pipeline and
        compute $D(g_k)$ = depth of longest chain
        embedded in $pairs(g_k)$.

Figure 2: **CPG Construction Algorithm**

**dent**, if a processor that receives data in $g_i$ also sends data in $g_{i+1}$. Consecutive phases with no processor dependencies overlap, and the aggregate completion time is the maximum of their complexities. Let $Ph(V) = (g_1, g_2, \ldots g_M)$ denote the set of communication phases attached to node $V$. The following algorithm computes the overall completion time $T(V)$. [1]

```
Complexity(V)
{
    if V is a computation phase
        if V was included in a communication phase
            return 0 { avoid multiple inclusions }
        else
            return T_s(V)
    else if Ph(V) = ∅
        return T_s(V)
    else
        T_p ← 0
        for i = 1 to M − 1
            if ((RL(g_i) ∩ SL(g_{i+1})) = ∅)
                T_p ← T_p + max(T(g_i), T(g_{i+1}))
                i ← i + 1
            else
                T_p ← T_p + T(g_i)
        return(T_s + T_p)
}
```

The following procedures evaluate the completion times of synchronous and pipeline communication phases:

**Synchronous phases:**

In a loosely synchronous phase $g$, each processor issues a send followed by a receive. If there is no computation performed in between, the phase completion time is simply

$$T(g) = \tau(m_g) + s + r,$$

where $\tau(m)$ is the transmission time of a message as a function of its size $(m)$, $s$ is the minimum overhead for sending a message, $r$ the minimum overhead for receiving a message, and $m$ is expressed in terms of $N$ and

[1] The complete algorithm has been summarized here, with some special cases removed for simplicity. For example, two communication phases whose ancestors are adjacent siblings in an APT are independent if their processor domains are disjoint (this may occur in the case of multiple independent pipelines).

$P$. However, if computation does occur after return from the send but before the following receive, then $\tau(m_g) + r$ time units of the message transmission time in the sending node effectively **overlap** with useful work. In addition, if each node is equipped with a message co-processor, then potentially all time for communication can be overlapped with useful computation. Let $W$ be the sum of the overlapped computation phases, $W = \sum_i w_i$. Then,

$$T(g) = max(\tau(m_g), W) + s + r.$$

**Pipeline phases:**
In asynchronous communication phases such as pipelines, each processor in the chain posts a receive before it sends, and typically executes some work in between. In this case, the completion time is divided into two stages: $T(g) = Fill(g) + Remain(g)$, where $Fill$ is the time to fill the pipeline, and $Remain$ is the completion time of a processor after it has received its first message. The fill time is determined by the depth of the pipeline, the communication time, and the amount of computation done at each stage:

$$Fill(g) = (D(g) - 1) \times (\tau(m_g) + s + r + W).$$

Here, $W$ is the sum of all computation phases between the receive and the corresponding send. The remaining time for the pipeline to complete depends on the width of the enclosing loop ($B$), the computation time ($W$), send ($s$) and receive ($r$) overheads and other system overheads ($o$) such as interrupt handling time [10]:

$$Remain(g) = B \times (s + r + o + W).$$

When generating complexity expressions for either phase type, the algorithm needs to determine which loops $w_i$ are executed in the course of a communication phase $g$. To obtain this information, MK instruments each loop corresponding to a computation phase as an event, and subsequently scans the trace output to determine which processors record instances of that event. Recall that $t(e)$ and $p(e)$ are the time of event $e$ and the processor on which it occurs. The necessary condition for including $w$ in $T(g)$ is that a processor executes $w$ between two consecutive communication calls in $g$:

$$(t(c1) < t(w) < t(c2)) \wedge (p(c1) = p(w) = p(c2)).$$

In synchronous phases, $c1$ is SEND and $c2$ is RECEIVE; in pipelines, it is the opposite.

Having derived the complexity expressions for an APT's communication and computation phases, in addition to its loop bounds and branch probabilities, we can consolidate the results to compute the completion time of the APT.

## 4.2 An Illustrative Example

To illustrate the analysis algorithm, we consider the back-substitution stage of a two-way pipelined Gaussian-elimination algorithm (PGE2), whose APT is shown in Figure 3. The APT nodes consist of a complex control-structure that includes { **IF, DO** }. Without processor information, the skeleton of the APT is not enough to determine the communication phases.

In order to illustrate how the analysis progresses, the figure also shows the processors that satisfy various conditionals. On analysis of a trace file with 8 processors, communication edges between the communication calls are introduced into the APT. These edges are shown as directed, dashed edges in Figure 3. By analyzing all connected communication calls, four distinct communication phases are observed.

The four computation phases are distinguished by bold boxes, communication phases are illustrated by arrows connecting the nodes in each set, and participating processors are shown in curly braces. The CPG algorithm proceeds as follows:
**Steps 1 and 2:** Four distinct communication sets are identified; they are labeled **S1, S2, P1** and **P2** in the figure.
**Step 3:** The send and receive lists are computed for each phase. (The lists turn out to be identical here; that is not always the case.)

| | **S1** | **S2** | **P1** | **P2** |
|---|---|---|---|---|
| **send list** | 4 | 5 | 1-4 | 5-8 |
| **recv list** | 4 | 5 | 1-4 | 5-8 |

**Step 4:** The top-level routine node **(PGE2)** is found to be the ancestor of all four sets; these sets are placed into the node's **phase list**.
**Step 5:** Set **P1** is recognized as a pipeline, because processors 2 and 3 both receive before sending. Set **P2** is handled similarly. Sets **S1** and **S2**, on the other hand, are labeled synchronous, since processors 4 and 5 both send before receiving.

In the subsequent communication phase analysis, loops **C3** and **C4** are found to contribute work to **P1** and **P2** respectively, since they are executed by processors between receive and send operations. Phases **S1** and **S2** are not annotated with any computation since no computation is found between the send and receive nodes.

Let **B** be the width of the loops in the APT (which in this example are identical), **m1** the size of the messages transmitted in the synchronous phase, and **m2** the size of the pipelined messages. The branch probabilities **B1** and **B2** are extracted from the trace file and approximated

as 1/2, and the other two as 0 (which are exact in the limit $P \rightarrow \infty$), and the pipeline depth is easily found to be P/2 by using a sequence of small executions and performing regression on the depth of the pipeline for different processor numbers.

The complexities of the four communication phases, as derived automatically by the above method, are listed below. In the pipeline cases, the complexity term automatically considers the overlap between communication and computation.

| | |
|---|---|
| **S1** | $\tau(m1) + s + r$ |
| **S2** | $\tau(m1) + s + r$ |
| **P1** | $(P/2 - 1) \times (\tau(m2) + T(C3)) + B \times (s + r + T(C3))$ |
| **P2** | $(P/2 - 1) \times (\tau(m2) + T(C4)) + B \times (s + r + T(C4))$ |

Returning to the top level node, we can now assemble the completion time for this stage of **PGE2**. The serial time $T_s$ is

$$T_s = 0 \times T(C1) + 0 \times T(C2) = 0,$$

since loops **C3** and **C4** are part of communication phases and thus are not included in the serial time. Phases **S1** and **S2** overlap because there is no conflict between their send and receive lists. The two pipeline phases also overlap for the same reason. The additional time $T_p$ attributable to communication is therefore

$$T_p = max(T(S1), T(S2)) + max(T(P1), T(P2)),$$

and the total completion time for this stage of **PGE2** is

$$\begin{aligned} T(PGE2) &= T_s + T_p \\ &= \tau(m1) + (B+1) \times (s+r) \\ &\quad + (\frac{P}{2} - 1) \times (\tau(m2) + T(C3)) + B \times T(C3), \end{aligned}$$

since $T(C3) = T(C4)$.

It is important to note that in generating the execution time $T(PGE2)$, no assumptions regarding the syntactic structure of the program were made. As illustrated, overlapping computation regions are also naturally handled by our general analysis methodology. In addition, overlapping communication phases are accurately determined and the analysis methodology uses this information appropriately.

# 5 Experimental Results

We implemented the algorithms described above in MK, and tested its analytical capability on two applications: a tri-diagonal linear system

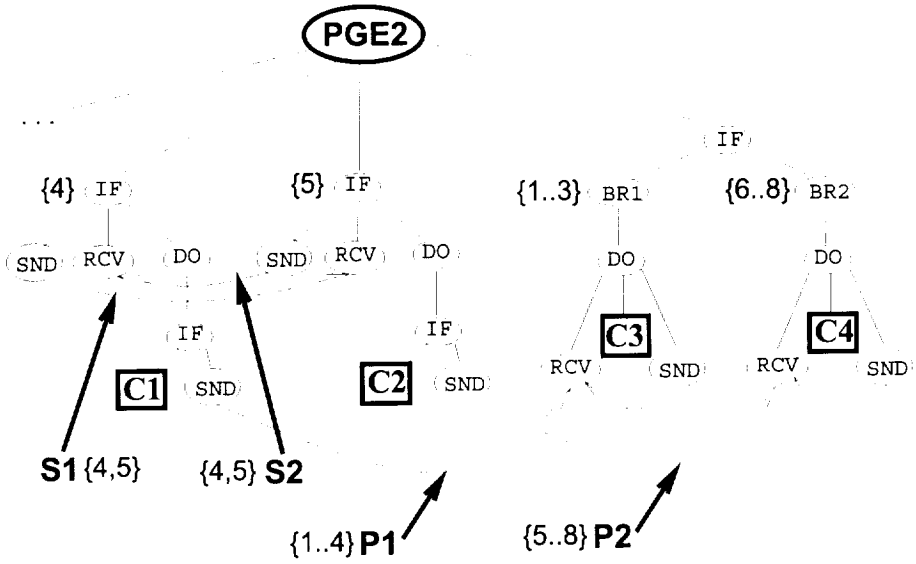


Figure 3: APT for the backward substitution stage of PGE2, annotated with communication phase information. Communication phases are indicated by dashed arrows. Processor chains of phases are listed in brackets, assuming an eight node partition.

solver using pipelined Gaussian elimination (PGE1, PGE2,) and a two-dimensional atmospheric simulation (ATM2D). We discuss the analysis that MK performed on each application and compare predicted versus actual execution times on the target machines.

## 5.1 Tri-diagonal Solver Using Pipelined Gaussian Elimination

The numerical solution of many problems reduces in part or fully to various matrix operations such as multiplication of matrices and vectors, LU decomposition, matrix transposes and the determination of eigenvalues and vectors. The matrix itself may be full, banded (with nonzero elements clustered around the diagonal), or sparse. Loosely synchronous programs are easier to analyze, resulting in good accuracy of prediction[6, 9]. In this section, we will consider programs with multiple communication phases, including both pipelines and exchanges.

This kernel operation is commonly used in a number of computational fluid dynamics computations such as a block tridiagonal solver(one of the NAS parallel benchmarks ) used to solve multiple, independent systems of block tridiagonal equations. This code was written by S. K. Weeratunga at NASA Ames Research Center. In this version of

the code the Gaussian elimination phase of the computation is parallelized. A pipelined Gaussian elimination algorithm is used for this purpose. Pivot elements of each column are determined in sequence and these values are used for updating succeeding columns. Processor **i** has the columns $\frac{(i-1)\times n}{p}$ to $\frac{i\times n}{p}$. Processors are arranged in a ring, so that each processor receives the updated pivot element from its left neighbor, updates its own column, and then transmits the data to its right neighbor. Two different versions of the pipeline program are considered. The first version (PGE1) uses a one-way pipelined Gaussian elimination algorithm while the second version (PGE2) uses a two-way pipeline.

As stated earlier, the primary goal of generating first-order performance information is to project overall scalability trends rapidly without extensive simulations, and thus to shorten the performance tuning and debugging cycle. Since analytically predicted execution times can be generated much more quickly than those for simulations or actual runs, the prediction model can be a valuable data generator for performance visualization tools. The predicted data can also be used to choose optimal values for parameters such as computation granularity, as we demonstrate below.

### 5.1.1 PGE1

MK identifies two communication phases in PGE1: one pipeline communication phase in the forward-elimination phase of computation and another pipeline phase in the backward-substitution phase. Each pipeline is determined to have a processor chain of depth **P-1**. In addition, the analysis correctly determines that the two communication phases identified cannot proceed in parallel due to processor dependencies between the two phases. Finally, the predicted execution times of the program using the closed-form symbolic expression generated by MK, compare favorably with experimental values.

### 5.1.2 PGE2

The steps taken by MK in analyzing the complexity of PGE2 were already discussed in the previous section. MK successfully identified the six communication phases in the program's execution: two pipelines in the forward-elimination stage, two pipelines in the backward-substitution stage, and two independent exchange communication phases in between. Each pair of phases overlaps due to the absence of processor

dependencies.

For each version of the program, MK used 4- and 8-processor traces to estimate loop computation times, build the CPG, and generate the complexity expressions. Predicted and actual times are compared in figure 4. The mean prediction error is 17% for PGE1, and 21% for PGE2.

## 5.2 Two-Dimensional Atmospheric Simulation

Our second test application, ATM2D, is a two-dimensional atmospheric simulation written by Bob Wilhelmson and Crystal Shaw at the National Center for Supercomputing Applications. ATM2D computes the temperature change over the simulated region by iteratively solving the planar diffusion and advection equations at each time step.

The execution platform for ATM2D was the 226-node Intel Paragon located at NASA Ames Research Center, running NX on top of OSF/1. The Paragon uses a packet-switched mesh interconnection network with a communication coprocessor at each node; its network parameters are 120 $\mu s$ latency and 30MB/s bandwidth.

The computational domain in ATM2D is partitioned by mapping the storage matrices across a logical mesh of processing nodes, so that each node performs its portion of the computation locally and trades boundary values with the four adjacent nodes. If the nodes are configured in a rectangular grid of size $P_x \times P_y$, and the entire modeled domain has dimensions $N_x \times N_y$, then each node's domain has area $N_x/P_x \times N_y/P_y$. To simplify the analysis, we will assume square processor domains of length $\lambda = N_x/P_x = N_y/P_y$. To prevent communication deadlock, each processor is labeled **red** or **black** (as on a checkerboard,) and executes the following (simplified) code:

```
do i = 1 to nsteps
    if (node is red) then
        send_and_receive(north, buffer, 2λ)
        send_and_receive(south, buffer, 2λ)
        send_and_receive(east, buffer, 2λ)
        send_and_receive(west, buffer, 2λ)
    else
        send_and_receive(south, buffer, 2λ)
        send_and_receive(north, buffer, 2λ)
        send_and_receive(west, buffer, 2λ)
```

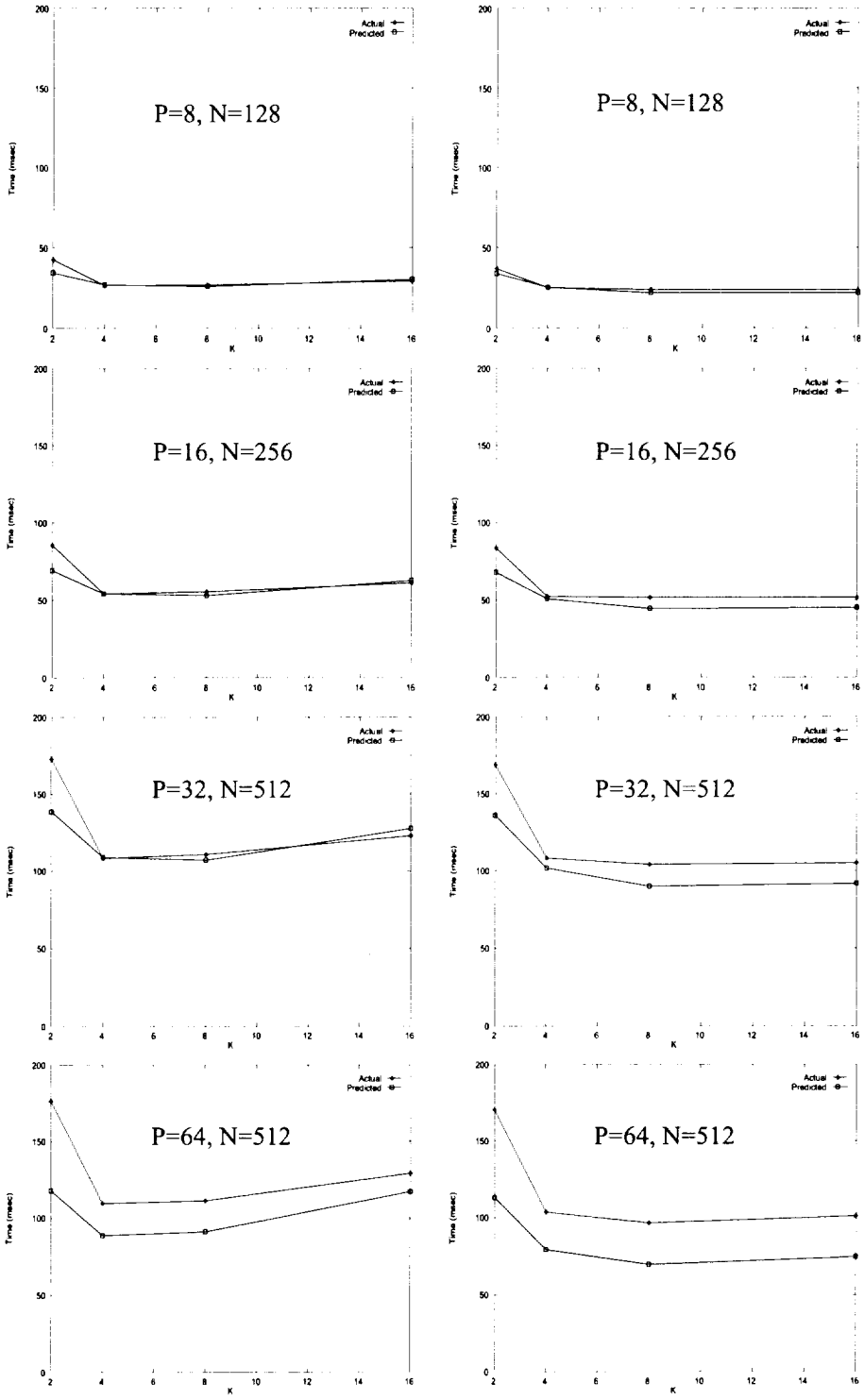


Figure 4: Predicted and actual execution times for PGE1 (left) and PGE2 as a function of block size, using various N and P values.

```
        send_and_receive(east, buffer, 2λ)
    endif
    compute diffusion, advection, and temperature
        change in local domain
enddo
```

Four synchronous communication phases take place between time steps, with message sizes proportional to $\lambda$, while the work at each time step is $\Theta(\lambda^2)$. These synchronous phases were easily detected and analyzed by MK, and the complexity function it generated is a quadratic function of $\lambda$:

$$T(N, P) = a + b\lambda + c\lambda^2$$

An implication of this prediction is that if the problem size is scaled proportionately to the mesh size (constant $\lambda$), the completion time will remain constant. This idealization is not realized exactly in practice, as we discuss shortly.

As mentioned earlier, one objective of MK is to provide a flexible basic framework (the APT/CPG structures) that can be improved as needed by adding lower-level details specific to the machine or application. In this case, we sought to model the scalability of synchronous communication phases more carefully, as this is the dominant factor in the overall scalability of ATM2D. Since all the nodes operate independently, any imbalance in their progression increases the receive-waiting time of the nodes that complete their work sooner. As the likelihood of skew is higher for larger partitions, one would expect the net communication time to be similarly increased. To estimate the effect of increasing the mesh size on communication time (with constant $\lambda$), we removed all computation from the program and re-timed the execution on partitions with 4 to 80 nodes. As figure 5 shows, execution time varies roughly as $\sqrt{P}$, with a constant factor independent of $\lambda$. Additional nonlinear effects are visible for larger values of $\lambda$ and P, due to larger messages and irregular mesh allocation, which produce some contention. This observed feature was added to MK's analysis of synchronous phases, with the result

$$T(phase) = \tau(m) + \alpha\sqrt{P},$$

where $\alpha$ was found experimentally to be 30 $\mu s$.

Predicted and actual times for different values of N and P, are compared in table 1. The predicted times were estimated by MK using only the smallest trace ($P = 4, \lambda = 128.$) The mean error over all 18 runs was 9.14%, and only 12.75% for P = 128.

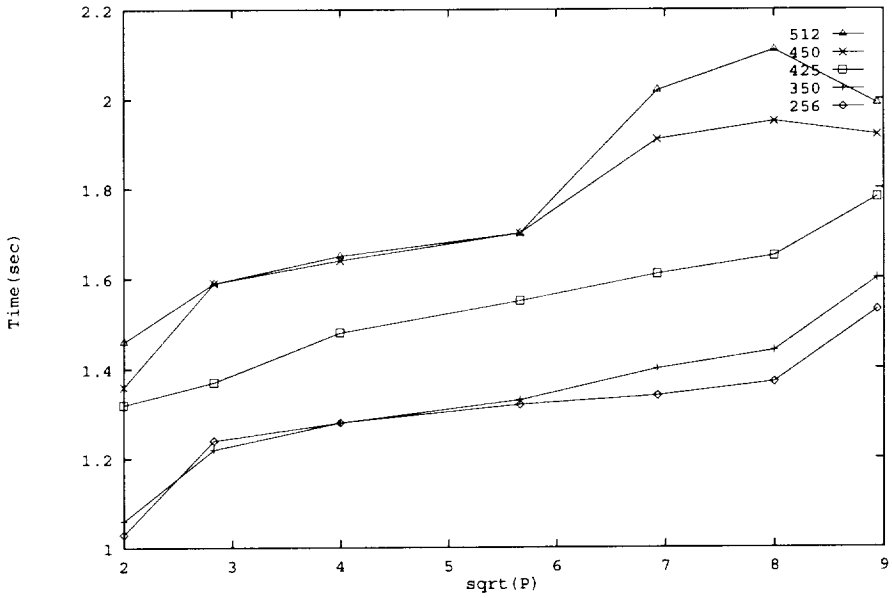


Figure 5: Communication time in ATM2D versus partition size, for different values of $\lambda$.

| | P | T(Actual) | T(Predicted) | % Error |
|---|---|---|---|---|
| $\lambda = 128$ | 4 | .817 | .769 | 5.88 |
| | 8 | .839 | .771 | 8.10 |
| | 16 | .861 | .776 | 9.87 |
| | 32 | .892 | .782 | 12.3 |
| | 64 | .937 | .791 | 15.6 |
| | 128 | .925 | .803 | 13.2 |
| $\lambda = 256$ | 4 | 3.11 | 3.00 | 3.54 |
| | 8 | 3.15 | 3.01 | 4.44 |
| | 16 | 3.21 | 3.01 | 6.23 |
| | 32 | 3.31 | 3.02 | 8.76 |
| | 64 | 3.46 | 3.03 | 12.4 |
| | 128 | 3.45 | 3.04 | 11.9 |
| $\lambda = 512$ | 4 | 12.49 | 11.91 | 4.64 |
| | 8 | 12.57 | 11.91 | 5.25 |
| | 16 | 12.80 | 11.91 | 6.95 |
| | 32 | 13.13 | 11.92 | 9.22 |
| | 64 | 13.71 | 11.93 | 13.0 |
| | 128 | 13.75 | 11.94 | 13.2 |

Table 1: Predicted vs. Actual Execution Times for ATM2D on the Intel Paragon.

## 5.3 Analysis of Prediction Error

The execution time of a parallel program depends on many contributing factors, of which MK models only a subset. For example, MK assumes that the timing of sequential loops depends only on the iteration space and a constant factor, while in reality there are nonlinear effects due to cache utilization. In general, the predicted times approximate measured times fairly well for midrange values of iteration counts, but tend to underestimate when the granularity is outside a certain range. Another source of error, as mentioned earlier, is the lack of timing information for sequential blocks outside of pure computation phases. These timings can, of course, be added at the expense of generating more trace data. Since timing information is obtained from trace files for relatively small **N**, memory effects in larger problems are hard to predict.

It turns out that a large fraction of the errors in PGE1 and PGE2 is not attributed to modeling computation phases, but rather to the nonlinear behavior of pipelined communications. Briefly, the non-linear behavior arises due to interrupt overheads when messages arrive at destination processors. This effect is particularly severe for small granularities and on the first processor in the pipeline. The nonlinear recurrence relation describing the effect of this phenomenon is beyond the scope of this paper. Incorporation of this more sophisticated pipeline model into our analysis is straightforward, but has not been implemented as of this writing.

## 5.4 Predicting Optimal Grain Size

In pipelined computations such as in PGE1 and PGE2, the choice of strategy for sending the array values through the processor chain affects program performance. One strategy involves packing the pivot elements from all the arrays into a single message, thereby reducing the number of messages. The number of pivots from each array (i.e. the **block size**, denoted by **K** in the above tables) ranges from 1 to 256. The number of messages is inversely proportional to the block size; the maximum block size of 256 corresponds to all pivots being sent in a single message. Clearly the choice of **K** affects the amount of computation performed between message sends. If the granularity is very fine, too many small messages are sent, and if it is too coarse, the pipeline initiation time may incur too much overhead. Selecting the optimal grain size that achieves both good load balance and low communication overhead is critical in obtaining good performance of pipeline algorithms.

One useful result of obtaining a closed form complexity expression **T** is

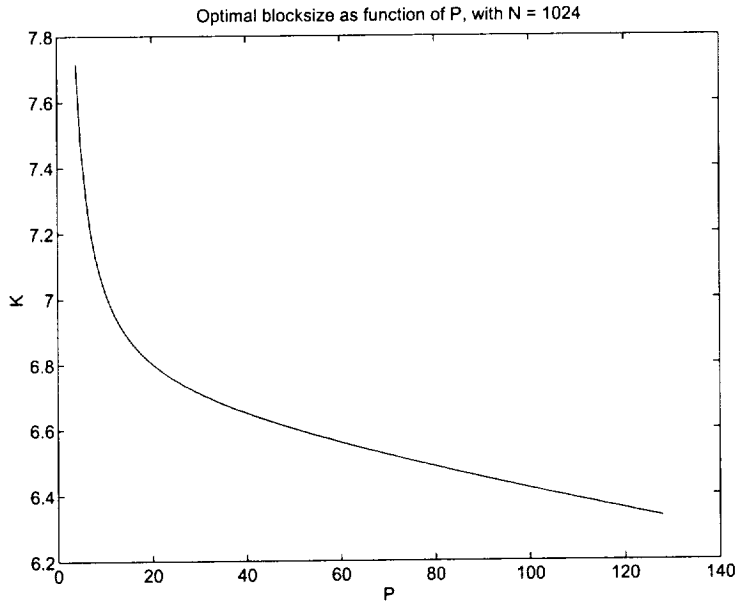


Figure 6: Optimal block size for PGE1 computed by solving $\frac{\partial T}{\partial K} = 0$.

that if grain size (**K**) is one of the independent parameters, the optimal grain size can be derived as the solution to $\frac{\partial T}{\partial K} = 0$. The complexity expression for PGE1 was differentiated using Mathematica's symbolic algebra functions. Figure 6 contains the solution plot for **N** = 1024 and **P** ranging from 4 to 128. The experimental data indicate that the optimal grain size is between 4 and 8 for different problem sizes. Comparison of results in Figure 6 show that the predictions accurately reflect this observation.

## 5.5 Computing Scalability Metrics

Two commonly used metrics for evaluating parallel performance are **speedup**, which measures **T**(1)/**T**(P) for a given problem size, and the **constant time** curve, which indicates how problem size must be scaled with system size to maintain constant completion time. A significant advantage of a closed-form approximation expression is the ease and quickness with which these metrics can be generated. Using actual data would require a large number of executions and simulations. We generated speedup and constant time curves using Mathematica [11] for both routines in less than a minute, and the graphs are displayed in Figures 7 and 8, respectively. Since the optimal block size is close to **8** for all parameter values tested, this value of **K** is used in the speedup plots.

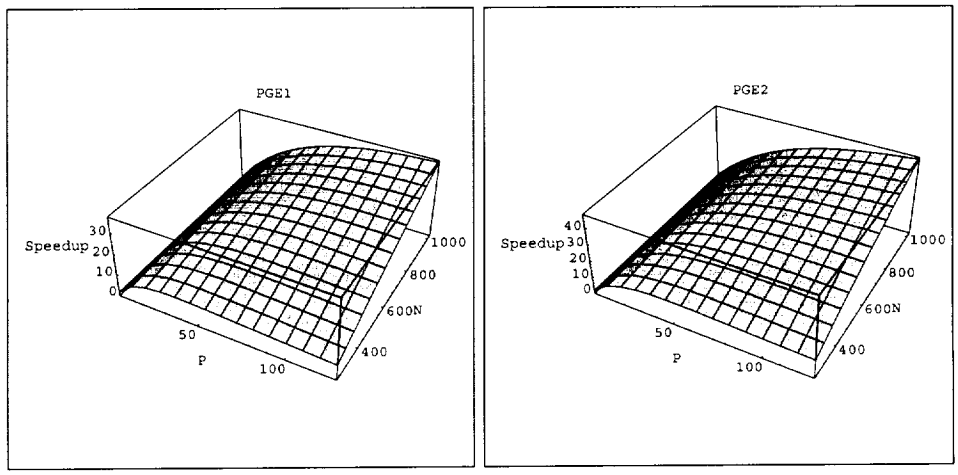


Figure 7: Speedup curves for one-way (left) and two-way pipeline Gaussian elimination, using nearest optimal block size $\mathbf{K} = 8$.

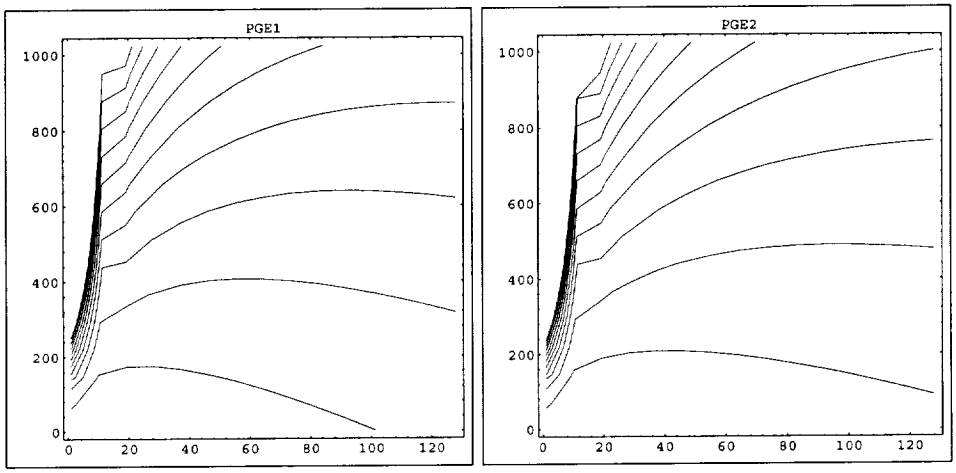


Figure 8: Constant-time curves for one-way (left) and two-way pipeline Gaussian elimination, using nearest optimal block size $\mathbf{K} = 8$.

The speedup curves illustrate the diminishing returns of increasing the system size for a fixed problem size, and where the saturation points occur. The threshold appears to be a roughly linear function of N in both routines, although as expected, the two-way pipeline algorithm exhibits about 20% better speedup in the parameter ranges under consideration.

The constant time curve provides valuable information about applications with scalable problem domains, because it answers questions such as: how much more accuracy can be achieved, or how many more molecules can be simulated, etc., in the same amount of time if the number of available processors is doubled? In the case of pipelined algorithms, the presence of an *O(P)* term corresponding to pipeline fill-time, indicates that **N** must be sufficiently large to minimize this startup overhead; figure 8 visualizes this phenomenon.

# 6 Conclusions

In this paper we have illustrated our approach for automated modeling of parallel programs. We have presented detailed algorithms for building two crucial data-structures (the APT and CPG) and the analysis of these data structures in determining program execution times as functions of **N** and **P**. The use of static compiler front-end tools and run-time information in the form of traces is vital for the goal of automated performance modeling.

We have considered a number of complex examples with multiple communication phases and overlapping computations. The results show that programs with overlapped computation and pipelined communication patterns can be automatically modeled and analyzed. In particular, our predictions are within approximately 10% of observed execution times for the purely synchronous application (ATM2D), and within 20% for the pipelined examples (PGE1,PGE2.) Further, the models are accurate in predicting the optimal grain size of the pipelined programs, and useful for obtaining scalability metrics on different network environments.

Future research in this area will extend the static analysis to consider more complex iteration spaces, and explore methods for characterizing programs with data-dependent complexities. Plans are also in progress to develop an integrated framework that can analyze HPF as well as explicit message-passing programs.

# References


[1] M.J. Clement and M.J. Quinn, "Analytical Performance Prediction on Multicomputers," **Proc. Supercomputing 93**, IEEE, pp. 886-893, November 1993.

[2] M. Crovella and T. LeBlanc, "Parallel Performance Prediction Using Lost Cycle Analysis," **Proc. Supercomputing 94**, IEEE, pp. 600-609, November 1994.

[3] T. Fähringer, "Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers," Ph.D. Thesis, Inst. for Software Technology and Parallel Systems, Univ. Vienna, Austria, September 1993.

[4] D. Gannon, *et al.* " Sigma II: A toolkit for Building Parallelizing Compilers and Performance Analysis Systems," **Proc. Programming Environments for Parallel Computing Conference**, April 1992.

[5] M. Heath, "Recent Developments and Case Studies in Performance Visualization using ParaGraph", Performance Measurements and Visualization of Parallel Systems, 1993, pp. 175-200.

[6] P. Mehra, M. Gower and M.A. Bass, "Automated Modeling of Message-Passing Programs," **Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**, pp. 187-192, January 1994.

[7] P. Mehra, C. Schulbach, J. Yan, "A Comparison of Two Model-Based Performance-Prediction Techniques for Message-Passing Parallel Programs," **ACM Sigmetrics Proceedings**, May 1994, pp. 181-190.

[8] S.R. Sarukkai, P. Mehra and R. Block, "Automated Scalability Analysis of Message-Passing Parallel Programs," to appear in **Transactions on Parallel and Distributed Systems**, Fall 1995.

[9] S.R. Sarukkai, "Scalability Analysis Tools for SPMD Message-Passing Parallel Programs," **Proc. Int'l. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems**, pp. 180-186, January 1994.

[10] R. Wijngaart, S. Sarukkai, P. Mehra, "Analysis and Optimization of Software Pipeline Performance on MIMD Parallel Computers," submitted to the **Journal of Parallel and Distributed Computing.**

[11] S. Wolfram, "Mathematica", Second Edition, Addison Wesley, 1991.

[12] J. Yan, "Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers," **Proceedings of the** $27^{th}$ **Hawaii International Conference on System Sciences,** Hawaii, January 1994.