Performance Prediction by Trace Transformation

Celso L. Mendes*

Department of Computer Science
University of Illinois
Urbana, Illinois 61801
E-mail: mendes@cs.uiuc.edu

ABSTRACT

Performance stability is an essential feature for the widespread adoption of multicomputers. In this paper, we report the preliminary steps of our research in performance prediction and extrapolation. Performance tuning, guided by extrapolation, may help achieve a substantial fraction of peak performance rates across a broader range of applications while providing guidance for code porting. We introduce a methodology for assessing stability of parallel programs, based on stability of the program execution graph, using time perturbation analysis. For programs with stable behavior, we present a model for performance prediction under architecture variations, by transformation of the execution traces with parameters that reflect the differences in architecture between two systems. We illustrate the use of this transformation with an example of a parallel PDE solver executing on a multicomputer.

*Supported by the Brazilian Institute of Space Research (INPE) and by a scholarship from the Brazilian Ministry of Education, Process CAPES-913/89-2

1 Introduction

Our work focuses on massively parallel, message-passing systems. These systems, also known as *multicomputers*, consist of a collection of autonomous processing nodes interconnected by a high-speed communication network. Scalability is a key feature of multicomputers. Machines have been built with over a thousand processors, and there are no insurmountable technological obstacles that would prevent multicomputers from scaling to sizes allowing multiple teraflop performance.

Despite the performance potential of multicomputers, several factors have limited their widespread adoption. Of these, their *performance variability* is a significant drawback. Execution of some programs may yield only a small fraction of peak system performance, while others approach the system's theoretical peak efficiency. Moreover, the observed performance may change substantially as application and architecture parameters vary.

Because performance tuning of parallel programs is time consuming and costly, and because performance varies with application and architecture parameters, mechanisms for estimating program performance as a function of application and architecture changes would accelerate the use of multicomputers. *Extrapolation* uses performance metrics and system analysis to predict the execution behavior of programs in response to application or architecture variations. Performance extrapolation can be used to help answer several important questions, including:

- Will the time spent porting a program to a a different parallel system yield performance gains that justify the porting costs?
- How will application scale with larger input data sets?
- How will application performance change with system size?

The first of these is a cross-machine performance prediction, and is the main subject of this paper; the others are extrapolations to a different configuration or problem size, and are part of our ongoing research. Our major goal is to develop and evaluate a methodology for prediction and extrapolation. Performance tuning, guided by extrapolation, may help achieve a substantial fraction of peak performance rates across a broader range of applications while providing guidance for code porting.

For a given application running on a parallel machine, we will study its *performance stability* — how the program behavior is affected by the architectural parameters of the underlying machine. Tracing is the basic technique to capture program performance data. Using this data, we will then develop models that allow performance extrapolation, as configuration parameters vary, for stable programs.

In the next section we list the major factors involved in performance prediction, and review related work in the area. In §3 we describe our approach to assessing program stability, which is a basic requirement for good predictions. We introduce our model for prediction under architecture variations in §4, and show an example of application in §5. Finally, we conclude the paper and summarize our future work in §6.

2 Background: The Performance Prediction Problem

The performance prediction problem on massively parallel systems has many possible dimensions, including those related to scalability (e.g., changes in the number of processors), to machine characteristics (e.g., type of processor on the nodes or type of interconnection among nodes), and even to the application problem itself (e.g., size of data sets). These issues can be broadly grouped as those regarding the specific architecture of the machine, and those intrinsically connected to the nature of the application. Along all such dimensions, however, the prediction process follows the same goal: extrapolation of performance results from some basic configuration to a different, target configuration.

2.1 Architecture Effects

The computational power of a multicomputer is primarily dictated by the type of processor used in the processing nodes. Most machines today employ state-of-the-art commercial microprocessors as their computing engines. Comparing two machines that use different processors is, however, a nontrivial task, even for uniprocessors. Several other factors, in addition to the processor type, play an important role in overall performance: cache organization, compiler quality and I/O bandwidth. The SPEC benchmarks [11] are a recent attempt to address this problem: performance data are reported for each of the individual benchmarks in the suite. Users then compare two machines by considering only those benchmarks that most closely resemble their typical application.

In parallel systems, interconnection network differences also affect overall performance. High latency, low bandwidth or network congestion (due to certain patterns of communication in the application) may cause some of the nodes to experience extended idle periods.

Any of these factors can change the order or duration of actions during program execution on different hardware configurations. Intervals with the same duration on one machine might differ on another, and such variations might be sufficient to create distinct execution paths in one of the processes of the parallel program.

2.2 Application Effects

Tuning an application on a specific parallel machine consists of finding the correct balance between computation and communication for a given data set. This balance, also known as *granularity*, must be such that parallelism is achieved (by dividing the computation among the processing nodes) and the overhead imposed by communication is minimized. In absolute terms, the granularity depends on the computation and communication speeds of the underlying hardware and software and on the application input data set. Variations in any of these components can change an application's balance.

In some applications, behavior is highly dependent on the input data; multiple executions of a given program on the same machine, with distinct input data sets, may differ dramatically. Such variability makes the prediction task extremely difficult. Conversely, applications with a deterministic nature are easier to model, making performance prediction simpler and more reliable.

2.3 Related Work

Performance prediction and the interaction of hardware, software and application variations have been widely studied for both sequential and parallel systems. As an example, Saavedra-Barrera and Smith [10] proposed a model for performance evaluation and prediction on uniprocessors. In this model, they identified standard operations and constructs in Fortran and characterized application programs by the number and type of these operations that were executed. By combining the program characterization with measures of machine performance on the standard operations, it was possible to estimate program execution time.

Three major factors prevent the direct extension of this methodology to parallel systems. First, the characterization of a parallel machine is more complex than of a uniprocessor, because of the interactions among processors. Second, parallel programs have more behavior variability than sequential programs, even across different executions on the same machine. The third reason is that, in general, there is a bigger semantic gap between high-level program code and compiled code on parallel systems than in sequential ones.

In another approach, Mak and Lundstrom [7] presented a method for predicting performance of parallel computations. They modeled a parallel computation as a task system with precedence relationships expressed as a series-parallel directed acyclic graph and machine resources as service centers in a queueing network model. On several test cases, they obtained very accurate predictions. However, Adve and Vernon [1] suggested recently that stochastic models may create unnecessary modeling complexity.

Lyon *et al* [6] made another claim against stochastic models, by proposing performance analysis at a macro level, thus ignoring particular details in the systems or in the applications. They inserted synthetic perturbations in a program, and measured their effect on global performance. The major goal was to find locations in the original program where optimization efforts should concentrate.

3 Program Behavior and Stability

Analisys of program stability is an essential step before performance prediction. In this section, we approach this problem by studying performance data captured from program execution under varying conditions.

3.1 Characterization of Program Behavior

We use tracing to characterize program behavior; tracing defines the sequence of activities that occur during execution. A trace consists of a sequence of event records that contains a timestamp and an event identifier that uniquely associates the event with an activity in the program.

3.1.1 Program Model

We define a concurrent program as a group of tasks that communicate by exchanging messages. Every processor executes exactly one of these tasks, and there is no task migration.

Each task consists of a sequence of activities, that can be of three types: computation, message sending and

message receiving, which we denote by C, S and R, respectively. The n^{th} occurrence of an event E on processor i is denoted by E_i^n, where $E \in \{C,S,R\}$.

Our basic model of communication assumes *nonblocking sends* and *blocking receives*. This means that a task can proceed after sending a message, even if the destination processor has not executed the corresponding *receive*. That allows overlaped computation and communication.[1] When the program executes a *receive*, however, it remains blocked until the required message arrives.

3.1.2 Execution Graphs and Partial Orders

We represent a parallel program by a directed graph where each trace event corresponds to a graph node. Directed edges define the event order. Every edge can also be associated with a numeric value, corresponding to some function of its initial and terminal vertices (e.g., the time between the events). We refer to this graph as the *program execution graph*.

A particular execution of the program, represented by the program execution graph, defines a relation on the set of events in the execution. This is Lamport's *happens before relation* [4], denoted by $<$. It has the following properties:

1. If E_i^m and E_i^n are events that occur on the same task i, and E_i^m occurs before E_i^n, then $E_i^m < E_i^n$.
2. If S_i^m is an event corresponding to the end of a message send on task i and R_j^n is an event related to the end of the corresponding message receive on task j, then $S_i^m < R_j^n$.
3. For any events E,F,G (in any tasks), if $E < F$ and $F < G$, then $E < G$. Events E and F are said to be *concurrent* if $E \not< F$ and $F \not< E$.

Thus, $<$ is an irreflexive partial order on the set of all events of the execution.

3.2 Program Stability

As we observed earlier, if the same program is executed on two different machines, it is possible that two different execution graphs will result. As an example, consider Figure 1, which shows a program with three tasks. The second and third tasks compute (represented by modules **A** and **B**, respectively, in Figure 1), and then send a message to the first task. This first task receives messages in whatever order they arrive. If the received message is from module **A**, some additional processing is required at the first task.

Assuming two given machines M_1 and M_2 with different processors, the observed behavior might vary depending on the nature of the computation: the first machine could be able to compute module **A** faster than module **B**, while the second machine could compute **B** faster than **A**. Under this assumption, Figure 2 shows the corresponding execution graphs. The set of events is the same, but there are different partial orderings. Specifically, we have the following relationships on each machine:

[1] Our definition of *nonblocking send* differs from some vendors' nomenclature; for example, by our definition, Intel's *csend* is a nonblocking call, because the sender task can proceed as soon as the message buffer is free, independent of the receiver task' status.

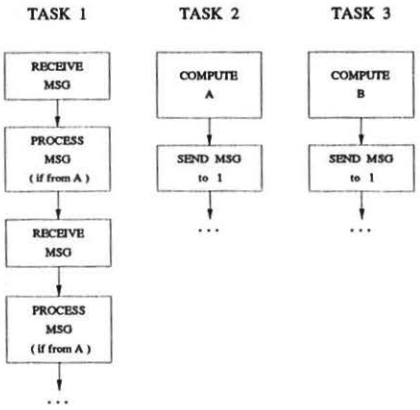


Figure 1: A parallel program fragment

- Machine M_1: $S_2^1 < R_1^1$, $S_3^1 < R_1^2$, $C_1^1 < R_1^2$; R_1^1 and S_3^1 concurrent
- Machine M_2: $S_3^1 < R_1^1$, $S_2^1 < R_1^2$, $R_1^2 < C_1^1$; R_1^1 and S_2^1 concurrent

We can detect such changes in the partial event order by analyzing the corresponding execution graphs: the graphs are equivalent (or, in graph theoretic terms, *isomorphic*) if and only if the partially ordered sets of events are identical.

If we simply assumed that the execution graph of a given program remained the same for every possible machine, our predictions could potentially fail, depending on the program and on the machines. The execution graphs of some programs present the same partial order of events across machines; we call these programs *stable*. Other programs may have different execution graphs even for two executions on the same machine and data set. We call these programs *unstable*. Predicting performance for this last class of programs is much more difficult, and is beyond the scope of this study.

Hence, the first step in performance prediction is evaluating program trace stability.

3.2.1 Message Receives and Stability

Based on the model of §3.1.1, repeated executions of the same program with the same input data can have different event orders only if messages are received and processed in a different order. Our implicit assumption is that multiple messages sent by the same processor are delivered to the receiver in the same order as they are sent by the originating processor.

In turn, the potential instability depends on the semantics and generality of the *receive* call. The most flexible form allows the task to receive any message, from any sender; further processing is required to identify the sender and the characteristics of the message. At another extreme, the receiving task might specify which

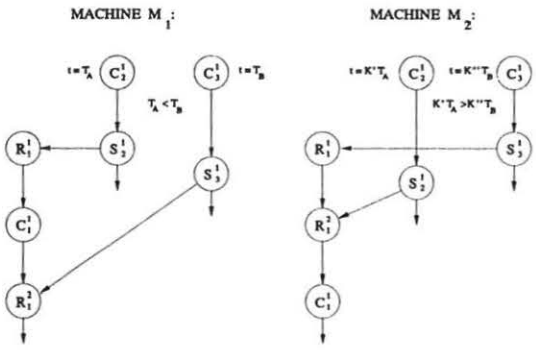


Figure 2: Execution graphs of the same program on two different machines

type of message, and from which sender, it is willing to receive; in this case, arriving messages that do not conform to the specifications are temporarily buffered. The receiving task is kept blocked until the specified message arrives. An intermediate approach specifies some of the parameters for a required message; the receiver may obtain a message from several senders, as long as that message satisfies the specified constraints.

Generality is not without price. As we saw in the example of Figure 2, the use of an unspecified *receive* operation makes the program unstable. Consider, for example, a program slightly different from the one in Figure 1. The only difference is that, now, a certain *type* of message is specified for every *receive* operation. With this constraint, the partial event order is the same for both graphs. The event order of this new program is not sensitive to speed differences between the two machines. Predicting the program's performance on a new system requires only the computation and communication characteristics of the new system; the event sequence is unchanged.

3.2.2 Time Perturbation

Different features on two machines may change the execution graph of a given program. Our problem consists of predicting the program behavior on a *target* machine, based on its behavior on a *base* machine and the architectural features of both machines. If the execution graph does not change significantly across machines, we can confidently use the event order on the first machine as a basis for prediction.

One possible way to assess program stability relies on *time perturbation analysis* [6]. The idea is to perturb the original program and verify the effect of such perturbation. Several instrumented versions of the program are executed, each with a specific set of time delays inserted in the code. By comparing the execution graph obtained in each experiment with the original execution graph, the sensitivity of program behavior to perturbations can be assessed.

The design of the required experiments includes the selection of code locations where delays will be inserted; each such location constitutes a *factor*. For each experiment, factors are set at one of their possible levels (e.g.,

delay or no-delay). In a full factorial design [3], with p locations selected, we must conduct 2^p experiments to determine the influence of each of the p perturbations. However, in practice, a much smaller number is needed, because interactions between distinct perturbations are not always significant.

Consider a program with two factors, and denote the locations of the factors by D_1 and D_2. Implementing a full factorial design requires executing and analyzing the behavior of four versions of the program:

1. $D_1 = D_2 =$ No-delay (regular program)
2. $D_1 =$ No-delay, $D_2 =$ delay
3. $D_1 =$ delay, $D_2 =$ No-delay
4. $D_1 = D_2 =$ delay

3.2.3 Graph Analysis

We can detect variations in the program execution graph, exposed using time perturbations, by testing for isomorphism of the corresponding execution graphs. Although it is not known whether graph isomorphism is an NP-complete problem, no polynomial time algorithm is known [8].

In our case, testing for isomorphism is insufficient — two execution graphs might be similar, but not isomorphic. We need to determine how "similar" they are. In other words, we need a metric to compare graphs. Under this metric, isomorphism means complete similarity. Graphs with high degrees of similarity represent nearly stable behavior.

The metric we propose is based on subgraph isomorphism. Let G_1 and G_2 be two graphs with n vertices. We define the *similarity* s between G_1 and G_2 as the degree of the largest isomorphic graphs H_1 and H_2, where H_1 is an induced subgraph of G_1 and H_2 is an induced subgraph of G_2. We also define the *distance* d between graphs G_1 and G_2 by $d = n - s$.

Under these definitions, the following two statements are equivalent for any graphs G_1 and G_2 with n vertices [12]:

1. There exist isomorphic graphs H_1 and H_2, each with at least $n - d$ vertices, such that H_1 is an induced subgraph of G_1 and H_2 is an induced subgraph of G_2.
2. There exists a graph G with at most $n + d$ vertices having two induced subgraphs G'_1 and G'_2 such that G'_1 is isomorphic to G_1 and G'_2 is isomorphic to G_2.

This means that, having G_1, we need to add at least d more vertices with appropriate edges to obtain graph G, which will also contain G_2 as a subgraph. Use the notation $d(G_i, G_j)$ to represent the distance between graphs G_i and G_j and the symbol $\cong$ to represent graph isomorphism; the following properties hold for any graphs G_i, G_j and G_k of degree n [12]:

- $G_i \cong G_j \Leftrightarrow d(G_i, G_j) = 0$

- $d(G_i, G_j) = d(G_j, G_i)$
- $d(G_i, G_k) \leq d(G_i, G_j) + d(G_j, G_k)$
- $0 \leq d(G_i, G_j) \leq n - 1$

Thus, d is a metric in the set of graphs of degree n. It can be used to compare graphs G_1 and G_2, and provides a quantitative measure of similarity. Algorithms for direct computation of d are known [5]. However, the time complexity of these algorithms is $\mathcal{O}(n!)$ for an n node graph. The large number of nodes expected in most execution graphs makes the cost of exact isomorphism tests prohibitive.

3.2.4 Alternatives for Graph Comparison

Because computing the exact distance between two execution graphs is intractable with current algorithms, we seek approximations that exploit specific features of execution graphs. In particular, they consist of p interconnected linear chains, one for each task.

We are currently investigating the effectiveness of two approaches to the graph comparison problem. The first approach compares two graphs by pairwise comparison of individual task execution traces, one trace from each graph. We look for the maximum possible mapping between vertices of the same type in the two traces, such that the original event order in each trace is preserved. Under our previous assumption that graph vertices can be of types C, S, R, each task trace is a linear string $E^1, E^2, \ldots, E^k$, with $E^i \in \{C, S, R\}$. Finding the maximum match between the executions of each task corresponds to finding the longest common substring between the strings representing each task trace. We approximate the similarity s by the total number of matches from the comparison of all pairs of corresponding task traces.

We can prove that the number of matches found by this trace comparison procedure is always greater than or equal to the number of matches that would be obtained by the exact procedure. If this were not the case, in one of the task traces the number of matches would be strictly smaller than in the exact procedure; this violates the assumption that the procedure finds a *longest* common substring, and thus it can not be true. Hence, the approximated distance is less than or equal to the real distance d.

A second approach to comparing two execution graphs divides both graphs into execution regions. The motivation is that many scientific programs exhibit iterative behavior, which produces patterns in their execution graphs. By detecting and comparing such patterns in the two graphs, we reduce the problem to a series of much smaller graph isomorphism problems. This might allow us to use exact algorithms for calculation of graph distances.

4 A Model for Performance Prediction

After determining that a given program has stable behavior, performance prediction can begin. The goal of building a model is to establish guidelines for the prediction process. The prediction consists of analysing

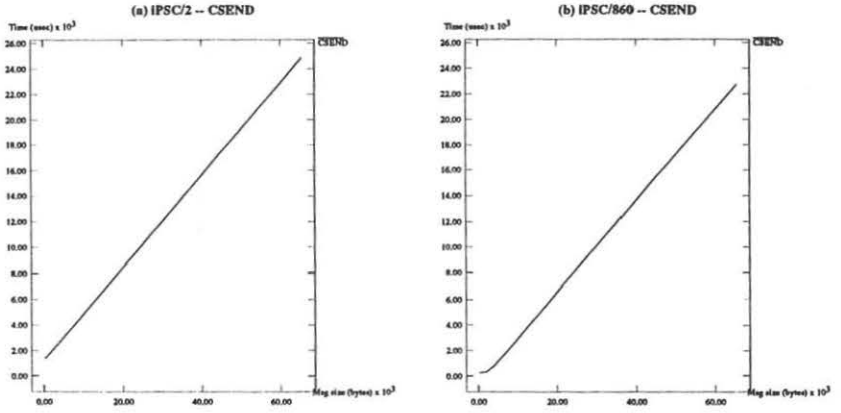


Figure 3: Time to execute a *send* on the Intel iPSC/2 and iPSC/860

program traces on one machine and, by applying the model, generating corresponding traces for the new system. We implicitly assume that the number of tasks is the same on both systems.

4.1 Architectural Parameters

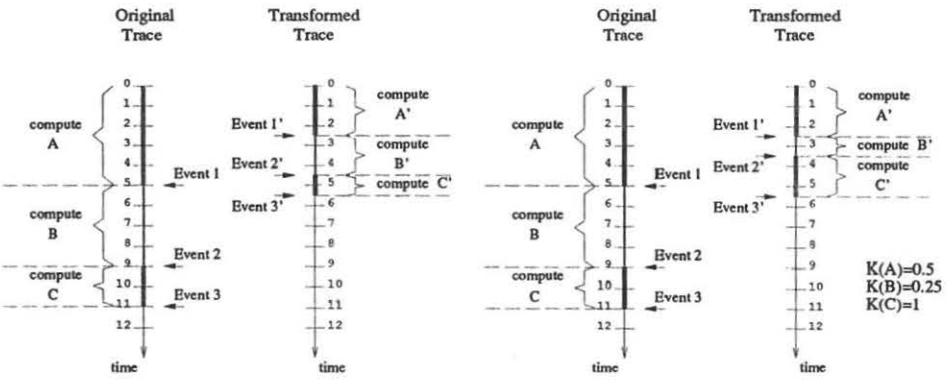
Performance prediction support is based on transformations of each trace event. Event timestamps from the original trace are adjusted to reflect the predicted duration of the corresponding activity on the new system.

For computation activities, the major transformation adjusts the ratio of processor speeds. By assumption this ratio can be a constant or dependent on some aspect of the code. The simplest approximation assumes a single ratio that could be derived either from published performance data for the two processors (e.g., SPEC ratio). or by executing a sequential version of the program on the two systems and computing the ratio of the total execution times.

A better alternative uses a variable ratio. Event records usually contain some information about the type of computation (e.g., procedure or loop identification). If some information about the effectiveness of the processors on different code fragments is known, one can derive a more realistic transformation.

In regard to communication, we take a simplified approach and make no assumptions about the underlying interconnection network or the message passing software — all elements of communication cost are extracted from traces of communication benchmarks. From the benchmark data one can build a model of the time to send or receive messages of a given length.

As an example, consider the Intel iPSC/2 and iPSC/860. Although their interconnection networks are the same, they use different processors, and their software message-passing latencies are different. Figure 3 shows the time to execute a *send* operation, as a function of message length, on the two systems. Given this data and the size of a message, one can transform the times of *send* events observed in application traces.



(a) Constant ratio (b) Variable ratio

Figure 4: Transformation of computation traces

4.2 Prediction Model

Our prediction model specifies transforms for computation, message transmission and message receipt events. The transform is applied on an event-by-event basis. We read event records from the original trace, adjust their timestamps, and generate predicted event records for the new machine.

4.2.1 Computation Events

We transform computation events using the processor speed ratio. Timestamps for the new machine are computed by adjusting the durations of the corresponding intervals. Figure 4a illustrates the transformation of a trace with a computation speed ratio of 0.5. In this case, the transformation consists of computing timestamps for events 1', 2' and 3', based on the timestamps of events 1, 2 and 3.

In general, intervals with computation activity are transformed as follows:

$$C_2 = K\ C_1$$

where C_1 is the activity duration in the original machine, C_2 is the predicted duration in the new machine, and K is the computation speed ratio of machines. In the more general case, K is a function of the kind of computation in the interval. As an example, Figure 4b shows a variable ratio K, which assumes a different value for each computation module.

4.2.2 Communication Events

Usually, traces contain two events for each message transmission: *send begin* and *send end*. *Send begin* events delimit the end of a computation interval and the beginning of a message transmission. Their times of occurrence are transformed using the computation speed ratio of the previous computation interval.

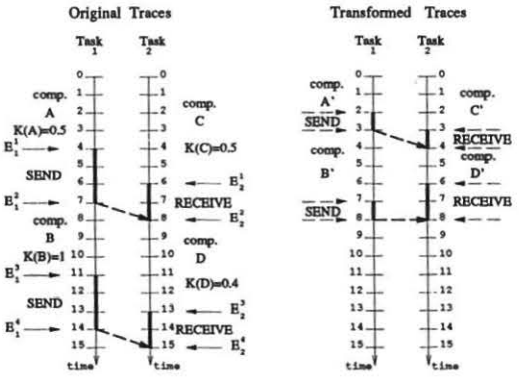


Figure 5: Transformation of message-related events

For *send end* events, like event E_1^2 in Figure 5, the original interval is transformed by a function derived from the message length and the communication characteristics of both machines. Assuming that both messages in Figure 5 have length b, the rule for transformation is

$$S_2 = \frac{f_2(b)}{f_1(b)} \; S_1$$

where S_1 is the duration of the *send* interval on the original system, S_2 is the predicted duration for the new system, and $f_1(b)$ and $f_2(b)$ are the times to send a message of length b on the original and new systems, respectively, obtained from their communication characterizations.

Events related to receiving a message can also be of two types: *receive begin* events and *receive end* events. Events of type *receive begin* are transformed exactly like *send begin* events.

To transform *receive end* events, we assume that communication characterizations are also available for the message *receive* operation. The basic rule for transformation in this case is:

$$R_2 = g_2(b)$$

where $g_2(b)$ is the time to execute a *receive* for a message of length b on the new system; this value can be obtained with a benchmark similar to Figure 3, for the corresponding *receive* operation. Assuming in the example of Figure 5 a value of 1 for $g_2(b)$ on both messages, the interval between events E_2^1 and E_2^2 is reduced in the transformed trace to half of its original value.

For the case of event E_2^4 in Figure 5, direct use of our basic rule for *receive end* events would lead to a causality violation: in the predicted trace, the message would be received before the end of the underlying *send* operation. In situations like this, we must follow the behavior of a real system, by respecting causality dependences. A *receive end* event should not occur before the *send end* event of the same message. Thus, we must delay the predicted time of event E_2^4 until the *send* operation completes, in this case at instant 8. We can now complete our general rule for transformation of *receive end* events, by considering causality effects:

$$\hat{T}_R^{end} = \max(\hat{T}_R^{begin} + R_2 \ , \ \hat{T}_S^{end})$$

where R_2 is the predicted duration for the new *receive* operation, computed by our basic rule; $\hat{T}_R^{begin}$ is the predicted instant for the beginning of the *receive* operation, and $\hat{T}_S^{end}$ is the predicted instant for the end of the corresponding *send* operation.

5 Trace Transformation Example

To illustrate the trace transformation method presented in the previous section, consider a trivial PDE solver, which iteratively computes the heat propagation on a bidimensional metal plate, under fixed boundary temperatures. The problem is discretized into N rows and N columns, and at each time step, N^2 new grid values are computed, based on the grid values from the previous time step. This process repeats until a steady state is reached.

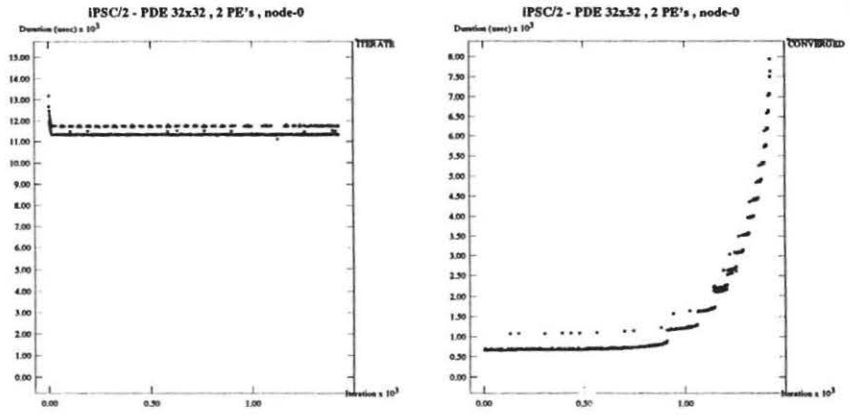
This algorithm has two phases that repeat over time: calculation of new grid values, and convergence checking. One possible parallel implementation of the algorithm assigns a set of contiguous grid rows to each processor; each processor computes the new grid values as well as a local convergence check, and a global convergence check occurs at the end of each time step.

Figure 6 shows the lifetimes of each phase from an instrumented version of this algorithm running on two processors of an iPSC/2. The vertical axis of each graph represents the execution time of each phase, and the horizontal axis represents the iteration. The lifetimes for the grid calculation are quite stable, as the time to compute the N^2 new grid values is relatively independent of the magnitudes of the values. However, the lifetimes for the convergence test have a significant variation across iterations. As execution proceeds, the grid becomes more uniform, and more points reach the steady state. The convergence checking algorithm must examine more points at each iteration to verify that convergence has not been reached. Figure 7 illustrates the execution graph for an iteration of this program with four processors.

The precise values of the procedure lifetimes on a given processor depend, among other factors, on the grid size, on the relative position of the processor in the grid, and also on the specific type of processor being used. We conducted the following prediction experiment: based on the traces obtained from execution of the PDE program, instrumented with the Pablo tracing library [9], on four nodes of an iPSC/2 and with a 64x64 grid, we applied our transformation model to compute the predicted traces for an iPSC/860.

The first step in the prediction was to execute the communication benchmarks on both the iPSC/2 and iPSC/860, to characterize their communication performance. Then we executed a smaller version of the PDE program (32x32 grid size) on reduced configurations of both machines (two processors each). We measured the amount of time spent by the processors on each procedure for this reduced problem, and used these values to derive the computation speed ratios between the machines, shown in Table 1.

After transforming the original iPSC/2 traces, we executed the same program on an iPSC/860, and compared the results with our predictions. For each iteration, we computed the ratio between the predicted and observed lifetimes of each procedure and message-passing function; the average values of such ratios are in Table 2. The



(a) New Grid Calculation (b) Convergence Checking

Figure 6: Iterative PDE solver lifetime traces from parallel execution on two nodes of an iPSC/2

Procedure	node-0	node-1
IterEdge	14.2	14.4
Converged	8.5	7.8
Iterate	21.3	20.9

Table 1: Computation speed ratios between iPSC/2 and iPSC/860 for reduced PDE problem

total execution times were 21.19 sec (observed) and 17.80 sec (predicted), with a prediction error of 16%.

The largest errors found in Table 2 were for the *isend* operation on nodes 0 and 1. A close analysis of the observed traces showed that network contention caused a deviation from regular behavior: in this program, both nodes execute the *isend* operation nearly at the same time (see again the execution graph in Figure 7), and thus one of them succeeeds, while the other must wait for their common network channel to become available. Also, the real behavior under such conditions for the iPSC/860 is different from the iPSC/2.

6 Conclusion and Planned Work

In this paper, we have reported preliminary steps of our research in performance prediction and extrapolation, namely the modeling of performance under architecture variations. We are currently complementing this preliminary study with models for scalability of both machine and problem sizes.

Our next step is to implement the required tools to extract the execution graphs from traces, and analyze their structure in terms of similarity as indicated in §3. We will use these tools to study the behavior of time perturbed versions of programs, in which delays are inserted in systematic patterns. In addition to the PDE

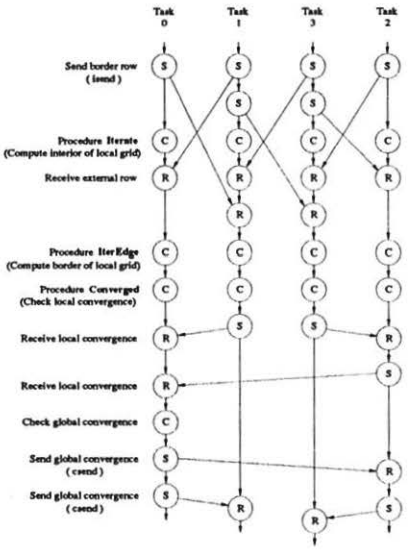


Figure 7: Execution graph for an iteration of the PDE program with four processors

program, we will be using message-passing codes from the HPF/Fortran D Benchmark Suite [2], a ray-tracing program, which has a more dynamic behavior and thus becomes a potential candidate to instability, and a variety of other parallel codes. For the programs which present stable behavior, we will perform all the three types of extrapolations mentioned before: cross-machine prediction, extrapolation to more processors and to a larger problem.

We will run these experiments on available Intel's (iPSC/2, iPSC/860 and Paragon XP/S) and Thinking Machines's (CM-5) multicomputers. With such machines, we can run cross-machine tests where only the processor is changed (iPSC/2 and iPSC/860), the interconnection network is changed (iPSC/860 and Paragon), or both

Function	node-0	node-1	node-2	node-3
IterEdge	1.19	1.11	1.19	1.10
Converged	1.30	0.99	1.02	0.97
Iterate	1.01	0.99	0.86	0.73
isend	0.29	0.48	0.82	0.57
csend	0.57	0.78	0.85	0.78
crecv	0.75	1.22	1.00	1.22

Table 2: Average ratios between predicted and observed lifetimes across iterations of the PDE program

processor and network vary (iPSC/2 and Paragon, or Paragon and CM-5). For the scalability tests, we will be using the faster machines (Paragon or CM-5), which also comprise a higher number of processors.

After running the experiments, and quantitatively comparing the predictions with actual results, we will perform a critical evaluation of our models. The goal in this phase is to quantify the accuracy of the models across a range of application codes. We will be looking for possible weaknesses in the models, or unforeseen sources of anomalies in the extrapolation process. If appropriate, the models will be refined, and new experiments conducted.

References

[1] ADVE, V. S., AND VERNON, M. K. The influence of random delays on parallel execution times. In *Sigmetrics Proceedings* (San Diego, May 1993).

[2] A.G.MOHAMED, G.C.FOX, VON LASZEWSKI, G., M.PARASHAR, T.HAUPT, K.MILLS, Y.LU, N.LIN, AND N.YEH. Application benchmark set for Fortran-D and High Performance Fortran. Tech. Rep. SCCS-327, Northeast Parallel Architectures Center, 1992.

[3] JAIN, R. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, New York, 1991.

[4] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July 1978), 558–565.

[5] LEVI, G. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *Calcolo 9* (1972), 341–352.

[6] LYON, G., SNELICK, R., AND KACKER, R. Synthetic-perturbation tuning of MIMD programs. Tech. Rep. NISTIR 5131, National Institute of Standards and Technology, February 1993.

[7] MAK, V. W., AND LUNDSTROM, S. F. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems 1*, 3 (July 1990), 257–270.

[8] READ, R. C., AND CORNEIL, D. G. The graph isomorphism disease. *Journal of Graph Theory 1* (1977), 339–363.

[9] REED, D. A., AYDT, R. A., MADHYASTHA, T. M., NOE, R. J., SHIELDS, K. A., AND SCHWARTZ, B. W. *The Pablo Performance Analysis Environment.* University of Illinois at Urbana-Champaign, 1992.

[10] SAAVEDRA-BARRERA, R. H., SMITH, A. J., AND MIYA, E. Performance prediction by benchmark and machine characterization. *IEEE Transactions on Computers 38*, 12 (December 1989), 1659–1679.

[11] SPEC. SPEC benchmark suite release 1.0. *SPEC Newsletter 2*, 2 (1990), 3–4.

[12] ZELINKA, B. On a certain distance between isomorphism classes of graphs. *Casopis pro pěstování matematiky 100* (1975), 371–373.