Performance Extrapolation of Parallel Programs

Kesavan Shanmugam, Allen D. Malony and Bernd Mohr

CIS-TR-95-14
May 1995

Department of Computer and Information Science
University of Oregon

Performance Extrapolation of Parallel Programs†

Kesavan Shanmugam, Allen D. Malony, Bernd Mohr

Department of Computer and Information Science,
University of Oregon, Eugene OR 97403, USA
{kesavans,malony,mohr}@cs.uoregon.edu

Abstract. Performance Extrapolation is the process of evaluating the performance of a parallel program in a target execution environment using performance information obtained for the same program in a different execution environment. Performance extrapolation techniques are suited for rapid performance tuning of parallel programs, particularly when the target environment is unavailable. This paper describes one such technique that was developed for data-parallel *C++* programs written in the *pC++* language. The technique uses high-level event tracing of a *n*-thread *pC++* program run on a uniprocessor machine together with trace-driven simulation to predict the performance of the program run on an *n*-processor machine. Our results show that even with high-level events, performance extrapolation techniques are effective in isolating critical factors affecting a program's performance and for evaluating the influence of architectural and system parameters.

Keywords: performance prediction, extrapolation, object-parallel programming, trace-driven simulation, performance debugging tools, and modeling.

1 Introduction

One of the foremost challenges for a parallel programmer is to achieve the best possible performance for an application on a parallel machine. The most common motivation for developing high-performing programs is that parallel machines are expensive resources that must be utilized to their maximum potential to justify their costs. However, the process of *performance debugging* (the iterative application of performance *diagnosis* [11] and *tuning*) invariably requires access to the target parallel platform, since the majority of the parallel performance tools are based on the measurement and analysis of actual program execution. As a consequence, during performance debugging, a parallel system is typically running less than optimal codes, often with the additional overhead of execution monitoring. The dependence on actual machine access for performance debugging also restricts the parallel programmer to consider optimization issues only for physically available machines. For parallel programs intended to be portable to a variety of parallel platforms, and scalable across different machine and problem size configurations, undertaking performance debugging for all potential cases is usually not possible.

The ideal way to address issues of machine access and efficiency of use is to provide the user with a performance evaluation environment that would require only limited access (if any) to the target system for effective performance debugging to take place. The environment would measure only what performance data was necessary from the execution environment and use high-level analysis to evaluate different program alternatives under different system configuration scenarios. In this manner, the environment would enable performance-driven parallel program design where algorithm choices could be considered early in the development process [21]. Two main performance evaluation directions have been pursued to approximate this ideal situation. Static performance prediction techniques use statistical performance models of a program which are evaluated analytically or through simulation. The models

† This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135 and Fort Huachuca contract ARMY DABT63-94-C-0029. The research is also supported by a NSF National Young Investigator (NYI) award.

can represent different levels of execution detail and can include parameters and components that allow alternative problem and system test cases to be studied. Some parameters may be derived from performance measurements. The work by [5,7,15] are representative of such static prediction approaches. Although it is possible to achieve surprisingly accurate estimates of global performance statistics (e.g. total execution time), the problems that arise in static performance prediction concern the inability to account for the performance effects of dynamic program behavior.

The alternative to static analysis is dynamic performance prediction based on trace-driven or direct execution simulation, where the architectural features of the target system and their interplay with the dynamic program execution can be more accurately represented. Although the work on the Proteus system [3,4] and the Wisconsin Wind Tunnel [19] has considerably advanced the efficiency and effectiveness of these dynamic techniques for architectural studies, the overheads are prohibitively high to warrant their use for rapid and interactive performance debugging.

In this paper, we describe a performance prediction technique that combines high-level modeling with dynamic execution simulation to facilitate rapid performance debugging. The technique is one example of a general prediction methodology called **Performance Extrapolation** that estimates the performance of a parallel program in a target execution environment by using the performance data obtained from running the program in a different execution environment. Our goal is to demonstrate that performance extrapolation is a viable process for parallel program performance debugging that can be applied effectively in situations where standard measurement techniques are restrictive or costly. From a practical standpoint, this implies that performance extrapolation methods must address the problem of how to achieve the comparative utility and accuracy of measurement-based analysis without incurring the expense of detailed dynamic simulation, but at the same time retaining the flexibility and robustness of model-based prediction techniques. Section 2 describes the basic concept of performance extrapolation. Section 3 presents the performance extrapolation tool, *ExtraP*, that we have developed for data-parallel *C++* programs written in the *pC++* language. The experimental results of applying *ExtraP* to *pC++* benchmark codes are discussed in Section 4. Section 5 describes how the performance extrapolation technique employed in *ExtraP* can be used in other environments. The paper concludes with a section on future work.

2 Performance Extrapolation

We believe that performance extrapolation is a concept that has not been explicitly identified or directly studied in the past. Hence, we begin with some definitions.

An **execution environment** is a collection of compiler, runtime system, and architectural features that interact to influence the performance of a parallel program. A complete execution environment would include the following important factors.

1. *Architecture*: Interconnection network topology, memory hierarchy, number of processors, and CPU architecture.
2. *Compiler*: Optimization strategies, calling conventions, storage management policies, and source code transformation algorithms.
3. *Runtime system*: Message passing conventions, data distribution policies, thread management, and scheduling policies.

It is also useful to consider the features of the input problem as part of the execution environment, since factors like problem size and distribution of the input values can influence performance behavior considerably.

The performance of a parallel program should always be specified with respect to a particular execution environment. An execution environment provides a reference framework for interpreting a program's performance on a single run as well as comparing the performance of different versions of the same program. A given program can also be evaluated under different execution environments and the environment best suited to that program chosen.

We define a **performance metric** as a measure of the quality of a parallel program. Total execution time is one of the most commonly used performance metrics for a parallel program. Other metrics like resource utilization and computation / communication ratio are also used often to evaluate parallel programs. Performance metrics are

important because they assist the user during performance debugging to identify performance bottlenecks and investigate how to resolve them. Because performance metrics are derived artifacts of program execution, they should be always specified with respect to an execution environment.

Performance metrics are obtained by analyzing the static and dynamic **performance information** about a parallel program's execution. Trace data obtained by running a parallel program is an example of performance information from which the execution time of the program and other metrics can be calculated. In some cases, the performance metric is trivially derived from the performance information. Sometimes more detailed analysis of the performance information is required. Because performance information depends on a measurement of a program's execution, it, too, must be associated with a particular execution environment.

The importance of execution environment is evident in the definition of performance extrapolation. **Performance extrapolation** is the process of obtaining the performance information PI_1 of a parallel program for an execution environment E_1 and using PI_1 to predict the performance information PI_2^p (the superscript p indicates a predicted quantity) of the same program in a different execution environment E_2. The performance information PI_2^p is then used to compute the predicted performance metrics of the program in E_2, PM_2^p. This process can be considered as a translation or extrapolation of PI_1 to PI_2^p using the knowledge about E_1 and its similarities to and differences from E_2. Figure 1 portrays the performance extrapolation process. PM_1 and PM_2 represent the performance metrics

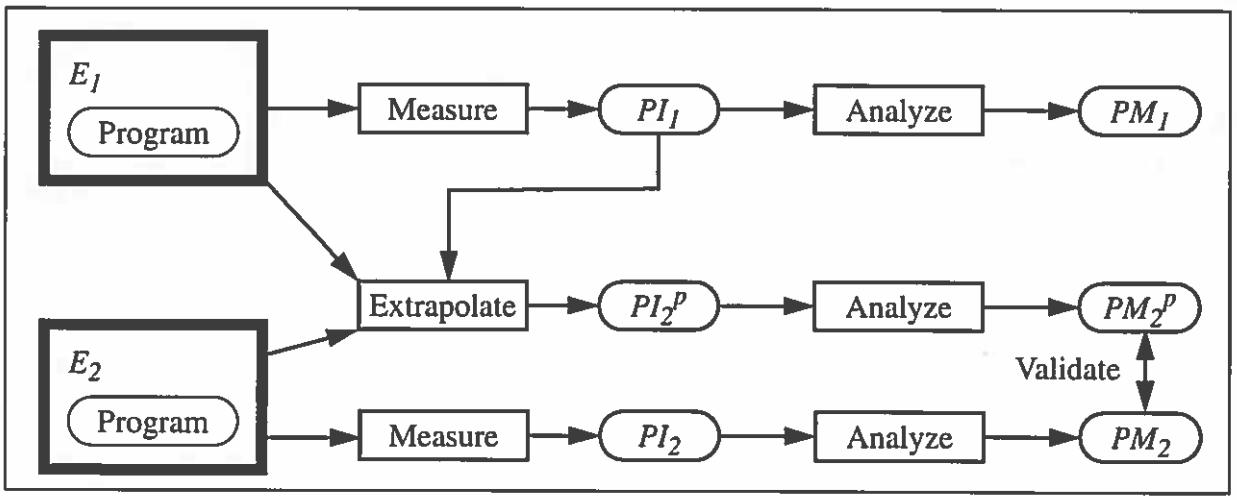


FIGURE 1. Performance Extrapolation

derived from measured performance information, PI_1 and PI_2, in the environments E_1 and E_2, respectively, and PM_2^p represents the predicted performance metrics derived from PI_2^p.

In principle, performance can be extrapolated across various execution environment parameters: number of processors, processor types, data distributions, different interconnect network topologies, processor mappings, etc. However, there are several problems that can arise during extrapolation that could affect the validity of the performance metrics produced. For instance, the analysis to derive a performance metric must often take into account the uncertainty in performance information and its effect on the accuracy of the metric. The work on perturbation analysis [14] is an example of how performance measurement intrusion issues can be addressed during post-mortem program trace analysis. Sometimes problems of performance information accuracy can be dealt with by deriving performance metrics *in vivo* of a system, as in direct execution simulation [6,18]. However, one of the most difficult problems involves non-deterministic program behavior between two execution environments. The performance extrapolation methodology makes the implicit assumption that the measured performance information in one execution environment is useful to predict the performance information in another environment. The major benefit of extrapolation is that measurements do not have to be made in the target execution environment to derive various performance metrics under different environment parameters. But this could also be a major shortcoming. In applying performance extrapolation, one must be aware of situations where the performance information in the measured

environment may be too incomplete or the program behavior may be too unpredictable to guide the extrapolation process.

3 A Performance Extrapolation Technique for *pC++*

We have developed a performance extrapolation technique that allows performance information and metrics to be predicted for data-parallel programs written in the *pC++* language [1,8,9,16]. In particular, we investigated the problem of extrapolating from a 1-processor execution of a *n*-thread parallel program to a *n*-processor execution for an environment where certain architectural and system parameters are configurable.

In general, our approach is to execute a *n*-thread *pC++* program on a single processor using a non-preemptive threads package. Important high-level events are recorded during the program run in a trace file. The events are then sorted on a per thread basis, adjusting their timestamps to reflect concurrent execution. The resulting set of trace files look as if they were obtained from a *n*-thread, *n*-processor run, except that they lack certain features of a real parallel execution. A trace-driven simulation using these trace files attempts to model those features and predict the events as they would have occurred in a real *n*-processor execution environment. The extrapolated trace files are then used to obtain various performance metrics related to the *pC++* program. The technique, depicted in Figure 2, is explained in detail in the following sections.

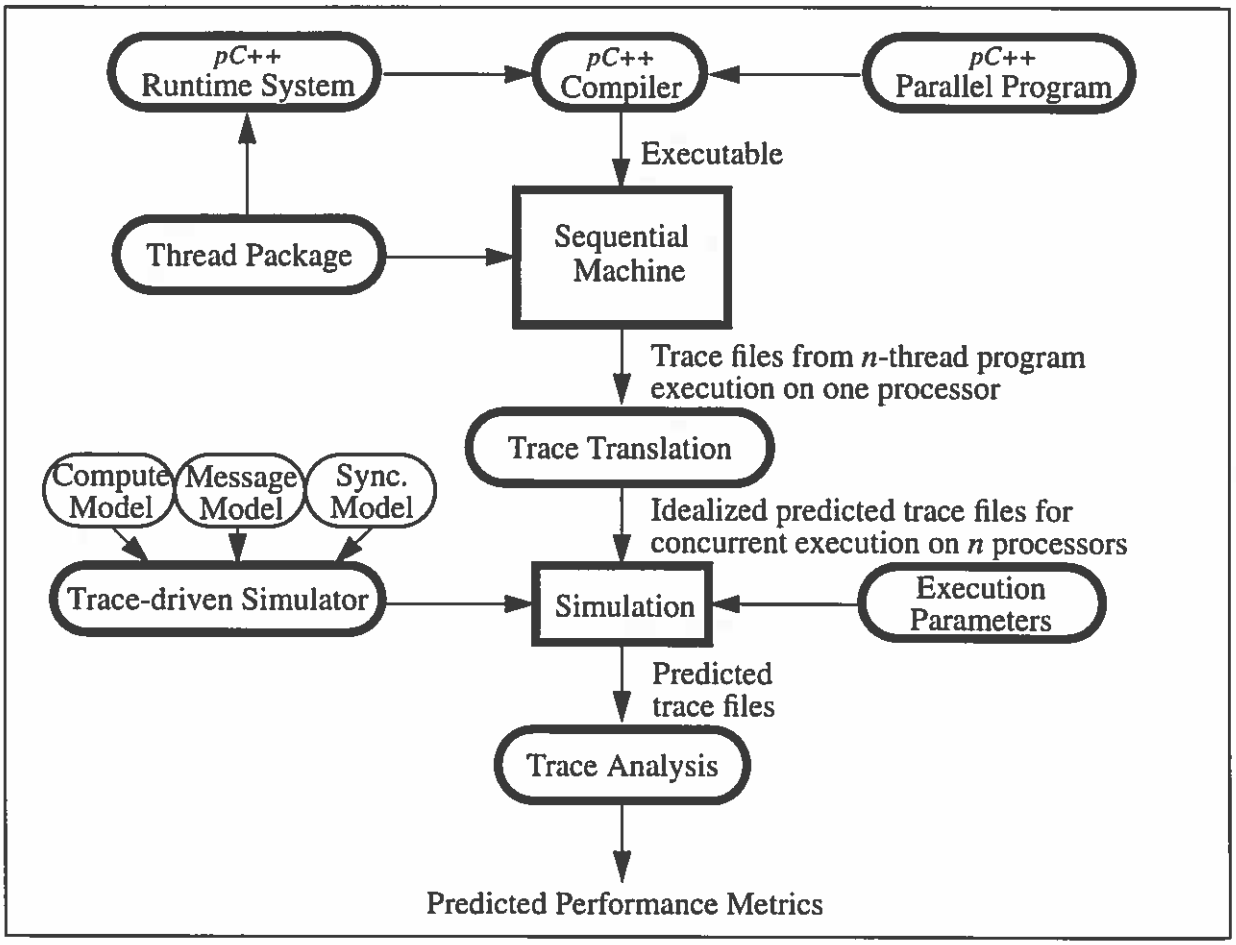


FIGURE 2. A Performance Extrapolation Technique for *pC++*

3.1 *pC++*: The Language, Compiler, and Runtime System

pC++ is a language extension to *C++* that supports an *object-parallel* execution model [1,8,9]. Under this model, a *collection* of objects can be distributed across a set of threads, in much the same way as arrays are distributed in HPF [2,12]. The objects which make up the collection are called the *elements* of the collection. The collection inherits certain member functions of its elements, so that when such a member function is called, it is called for every element in the collection. This parallel method invocation for collections of elements is the main source of parallelism in *pC++*.

The compiler accomplishes a parallel method invocation by generating code so that each thread calls the method for all its local elements. At the end of each parallel method invocation, the threads are synchronized by a global barrier. In addition, when a thread wants to access an element which it does not own, it generates a *remote element* request to be serviced by the thread that owns the element. The runtime system provides facilities for creating the threads, synchronizing the threads using a barrier and for accessing remote elements [1].

One of the important things to notice about *pC++* is that currently it does not provide protected accesses to remote elements. Though this will change in the future, we must point out that this particular characteristic of *pC++* has made our extrapolation technique simpler. Even though unprotected remote element accesses can lead to race conditions, in this paper we are considering only programs which do not have that problem. This issue is discussed further in Section 5.

3.2 Instrumentation and Trace Translation

For the purposes of our performance extrapolation work, we modified the *pC++* runtime system so that all n threads of a parallel program are executed on a single processor (in a virtual parallel manner) using a non-preemptive threads package [10]. The elements of a collection are allocated in a global space accessible by all the threads. When a thread requires access to a remote element, it gets it directly from the global space. Thus, under this runtime system, remote accesses are indistinguishable from local accesses (in terms of timing characteristics) and thread switches happen only at barrier entry and exit points.

Since the only interactions between threads in the *pC++* programming model occur during barrier synchronizations and remote element accesses, the runtime system was instrumented to record all such interactions. The result of a 1-processor performance measurement is a trace file which contains barrier entry, barrier exit, and remote access events from all the threads in the *pC++* program. These high-level events form the basis for the extrapolation and the time between the events reflect the computation times of the threads.

The trace translation algorithm takes in the trace file produced by the n-thread, 1-processor run of a *pC++* program and creates n trace files each containing events from one thread. The timestamps are adjusted to reflect the ideal parallel execution of the threads' computation on a n-processor machine. This is accomplished by retaining the time between two consecutive events for a thread and by enforcing the semantics of the barrier synchronization events. For example, if e_1 and e_2 are two consecutive events from the same thread with timestamps t_1 and t_2, and if e_1's timestamp was adjusted to time t_1', then e_2's timestamp will be adjusted to $t_2-t_1+t_1'$. This kind of adjustment is done only for non-synchronization events. For synchronization events, the timestamps are adjusted so as to preserve barrier behavior. For example, the translated timestamp of a barrier exit event from a thread, T_i, will be set to the timestamp of the barrier entry event from the last thread that entered that barrier, T_{last}. Notice that the trace translation algorithm relies on the fact that the threads are scheduled only at synchronization boundaries (i.e., the threads are not preempted until they encounter the barrier). That is the reason why a non-preemptive threads package must be used for the n-thread, 1-processor run. The trace translation algorithm is easily modified to handle the overhead for recording the events, flushing the event buffer, and switching the threads.

The translated trace files capture the program execution times between events under the assumptions of instant remote accesses, instant barrier synchronization (threads exit a barrier as soon as the last thread comes in), and unperturbed thread computation. Although these assumptions are idealized, the claim that we make is that *pC++* performance extrapolation can now be done for an n-processor execution by using the high-level events in the traces to drive simulations where models attempt to capture the execution realities of these performance factors (i.e., cost of

remote accesses, synchronization overheads, processor performance) in the target environment. Because of the data-parallel *pC++* execution model (and the assumptions that we make about pC++ programs, see Section 5), the high-level event sequence for each thread between the 1-processor execution and the predicted *n*-processor execution is deterministic, and extrapolation requires only the simulation of target system behavior for remote data access and barrier synchronization.

3.3 Simulation Architecture and Models

The trace-driven simulation is the heart of the *pC++* performance extrapolation. The simulation system consists of three main components:

- Processor model
- Remote data access model
- Barrier model

These components correspond to the three main types of information captured by the trace files: computation time between events, remote collection element accesses, and data-parallel synchronization barriers. Each component has several parameters that can be chosen to reflect the environment for which the prediction is to be done. Composed together, the components form a high-level model of the target system. Indeed, the simulation architecture was designed in an object-oriented manner such that the components (and their sub-components) interact through time-based objects (e.g., messages). The importance of this design approach is that we can easily chose the model type and level of functional analysis for each component, and flexibly substitute models to trade off issues of efficiency, accuracy, and detail in the simulation, depending on the performance debugging needs at the time. The simulation components for *pC++* extrapolation that we have implemented are explained in detail in the following sections.

3.3.1 Processor Model

As mentioned earlier, the main characteristic of any performance extrapolation technique is that it reuses information from one platform. In our case, it is the computation time between events on each thread and the event sequence recorded in the trace files that are reused. However, the computation times are clearly dependent on the processor performance. If the performance of the target system's processor is different from the measured machine, the difference must be addressed during extrapolation. For *pC++* extrapolation, we use a simple ratio of processor speeds to scale the computation time between events appropriately for the target machine. Such a ratio can be easily obtained by measuring the MFLOPS (or other processor performance) ratings of the target machine and the machine on which the experiments are performed. For example, the experiments described in this paper were all performed on a sun4 machine with a MFLOPS rating of 1.1360, as determined by a simple floating point benchmark. On a CM-5, the (scalar) MFLOPS rating was computed as 2.7645. So a ratio of 1.1360/2.7645 = 0.41 would be used during simulation for extrapolating the performance from a sun4 platform to a CM-5 platform.

In addition to processor speed scaling, the processor model represents certain operational aspects of the *pC++* runtime system. One important aspect is the policy about how remote data accesses are serviced and what message handling functions are performed for the chosen policy. The following remote data access policies are currently supported:

No interrupt: In this case, no messages are handled during the time between events. Messages are processed only when a thread waits for a barrier release or a remote data access reply.

Interrupt: In this case, an arrival of a message for a particular thread interrupts its computation. After the message is processed, the thread resumes its computation.

Poll: The scaled computation time between events is split into smaller chunks, and at the end of each chunk, the thread processes messages that have been received during that time.

The runtime system also determines how threads are assigned to processors, affecting issues of data locality and processor sharing. The results reported below reflect the current *pC++* runtime system which allocates each thread to a separate processor (hence, our focus on *n*-processor extrapolation) and targets either a shared memory or distributed

memory system architecture. However, an advantage of extrapolation is that we can approximate other runtime system behaviors, as long as the per-thread event sequence is unmodified. We have implemented a processor model (with an approximated runtime system) that supports multithreading on processors and shared memory clustering between processors. This has allowed us to explore the performance effects of these changes that are now being implemented for the *pC++* system.

3.3.2 Remote Data Access Model

The simulation models each remote access in the program as a remote request for data from one thread to the thread that "owns" the data (*pC++* follows the "owner computes" model [9,12]). The owner thread services the request and returns the data to the requesting thread. This is equivalent to how the *pC++* system operates in distributed memory environments. Hence, messages are the natural representation for the remote access protocol in the simulation. Figure 3 graphically depicts the how the remote data accesses are processed in the simulation using messages. During

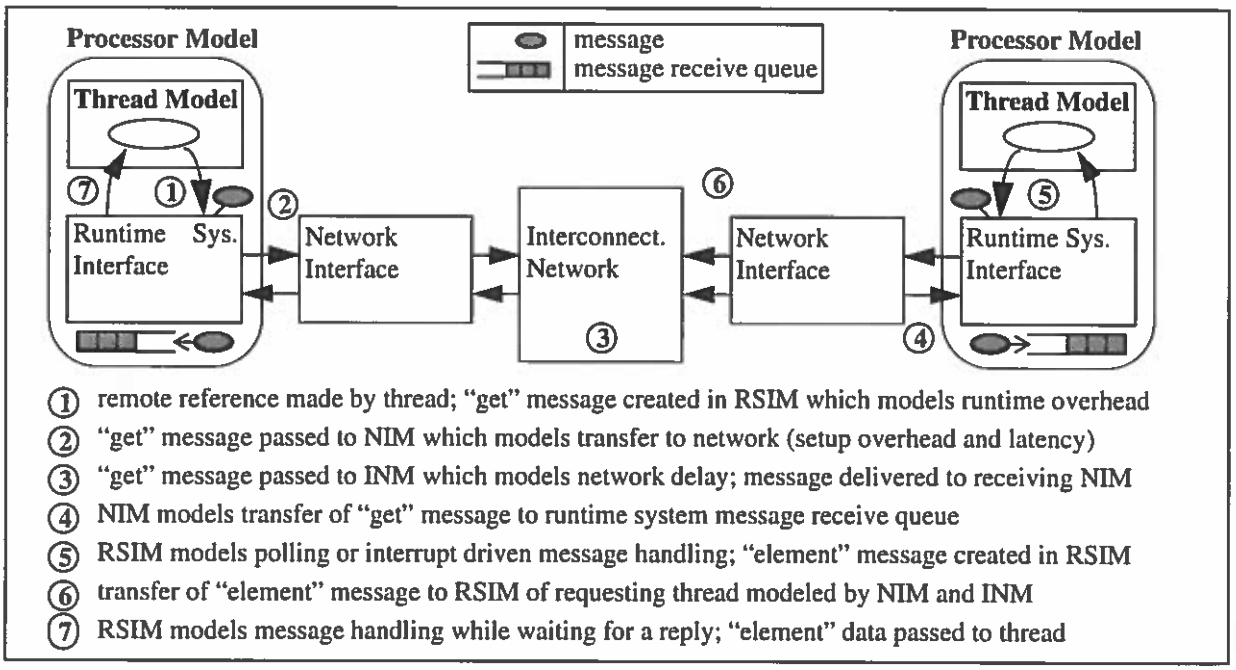


FIGURE 3. Remote Data Access Model

this processing, various performance characteristics of a distributed memory, message passing environment are modeled, as described in the figure. To simulate the performance of remote accesses in a shared memory system, the same protocol structure is used, but the parameters of the component sub-models (e.g., the network interface model and the interconnection network model) are changed to reflect shared memory data transfer. Because we are simulating performance at a high-level, we feel that this general approach is appropriate. Again, the simulation architecture allows a more specific and detailed shared memory model to be substituted if it is necessary to increase simulation realism and accuracy. Also, representing remote accesses generically by messages allows us to easily accommodate a multi-clustered system with shared memory access within a cluster and message passing between clusters.

The remote data access model includes parameters to represent: communication start-up overheads, communication bandwidth, message types and sizes, message construction overhead, network topology, and network contention. Aside from the operational aspects of the sub-models, the remote access performance estimates produced in the *pC++* extrapolation are mostly analytical; a full description of the performance equations is given in [20]. However, modeling contention requires time-based state analysis. That is, the contention models we developed include parameters based on the intensity of concurrent use of shared system resources (e.g., the interconnection network)

during the simulation. Except for concurrent access to message receive queues, we did not simulate the low-level contention behavior directly (e.g., the allocation of network links to contending messages). Instead the contention models were analytical expressions of remote access delay involving the contention factors calculated from the simulation state. More detailed simulation of contention would severely impact the speed of performance extrapolation.

3.3.3 Barrier Model

The current barrier model is based on a linear, master-slave barrier synchronization algorithm. Thread 0 acts as the master thread while all the other threads are slaves. Every slave thread entering a barrier sends a message to the master thread and waits for a release message from the master thread to continue to the next data-parallel phase.The master thread waits for messages from all the slaves and then sends release messages to all of them. For distributed memory systems, the *pC++* runtime system must continue to service remote data access messages that arrive at a processor even when the threads that run on that processor have reached the barrier. This is also true in the simulation.

The barrier model has several parameters that can be adjusted to match the actual barrier implementation on the target machine. Table 1 lists these parameters and briefly describes their operation. Hardware barriers or barriers

Parameter	Description	Example
EntryTime	Time for each thread to enter a barrier.	5.0 µsec
ExitTime	Time for each thread to come out of the barrier after it has been lowered.	5.0 µsec
CheckTime	Delay incurred by the master thread every time it checks if all the threads have reached the barrier.	2.0 µsec
ExitCheckTime	Delay incurred by a slave thread every time it checks to see if the master has released the barrier.	2.0 µsec
ModelTime	Time taken by the master thread to start lowering the barrier after all the slaves have reached the barrier.	10.0 µsec
BarrierByMsgs	1 - use actual messages for barrier synchronization. The message transfer time will contribute to the barrier time. 0 - do not use actual messages for barrier synchronization.	1
BarrierMsgSize	Size of a message used for barrier synchronization.	128

TABLE 1. Parameters for the Barrier Model

implemented through shared memory are easily represented. The linear barrier model delivers an upper bound on barrier synchronization times. We can easily substitute other barrier algorithms (e.g. logarithmic) if a more accurate simulation of barrier operation is required.

4 Experimental Results

To evaluate the concept of performance extrapolation and, in particular, the efficacy of the *ExtraP* tool, we performed several extrapolation experiments on codes in the *pC++* benchmark suite. These codes represent a wide range of execution behaviors, reflecting different degrees of computation and communication; Table 2 briefly describes the benchmarks used. Our goal in these experiments was two-fold. First, we wanted to establish that the extrapolation methodology could be applied in an actual parallel programming context where performance debugging is an

Benchmark name	Description
Embar	NAS "embarrassingly parallel" benchmark
Cyclic	Cyclic reduction computation
Sparse	NAS random sparse conjugate gradient benchmark
Grid	Poisson equation on a two dimensional grid
Mgrid	NAS multigrid solver benchmark
Poisson	Fast Poisson solver
Sort	Bitonic sort module

TABLE 2. *pC++* Benchmark Codes used for Extrapolation Studies

important component [16]. Second, we wanted to verify that modifying simulation parameters of interest resulted in observable and expected effects in extrapolated benchmark performance behavior.

Although *ExtraP* provides an ability to do performance debugging of portable *pC++* programs, to apply extrapolation confidently in a specific execution environment, we must be able to represent the target system appropriately in the simulation and to validate extrapolated performance against measured results. We implemented a matrix multiplication code in *pC++* and ran it on the Thinking Machines CM-5 for different numbers of processors and data distributions to demonstrate how effectively *ExtraP* can approximate actual execution characteristics.

4.1 *pC++* Extrapolation Studies

The large number of simulation parameters available in the models described above makes it possible to perform a broad range of extrapolation experiments. For our experiments, we created several parameter sets, each varying a particular parameter across some range; for example, one parameter set varied the network bandwidth parameter *ByteTransferTime* from 0.2 µsec (5 Mbytes/second) to 0.005 µsec (200 Mbytes/second). For the most part, the experimental parameters reflected a distributed memory target system. We ran all benchmark codes for each parameter set for 1, 2, 4, 8, 16, and 32 processors.

Initially, to observe processor scaling effects, we selected a single parameter combination and ran *ExtraP* on all benchmarks and processor numbers. The results are shown in Figure 4. The curves clearly show the range of performance found in the *pC++* benchmark suite and the speedup values are representative of what we have observed in different real environments. Although we were not attempting to validate the extrapolation results for a particular target environment in these experiments, the parameter values selected portray a distributed memory platform with modest communication link bandwidth (20 Mbytes/second), but relatively high communication overheads and synchronization costs. *Embar* is expected to deliver linear speedup on almost all platforms, and it does so here. *Cyclic* and *Poisson* show reasonable speedup improvement, but the other codes are more severely affected by the costs of communication or synchronization barriers.

In particular, we notice that for *Grid* and *Mgrid* speedup levels off after four processors, a result that was different from what we have observed on several shared memory parallel platforms [1], where speedups greater than 14 for 32 processors are achieved. This is not too surprising, given that we used a distributed memory parameter set. However, we were curious about the cause. We selected the *Grid* benchmark for further study; the performance results are shown in Figure 5. One obvious difference between a shared memory and distributed memory system is the bandwidth for remote data access. We changed the parameters set to reflect 200 Mbytes/second communication, an approximation of the performance for remote data accesses in shared memory machines. This improved the time performance somewhat, but speedup performance was still only half of the shared memory case. From trace statistics, it was clear that barrier synchronization time was insignificant in *Grid*, so the lack of speedup must be attributed to communication start-up overhead, or so we thought. We extrapolated to an ideal execution environment where all

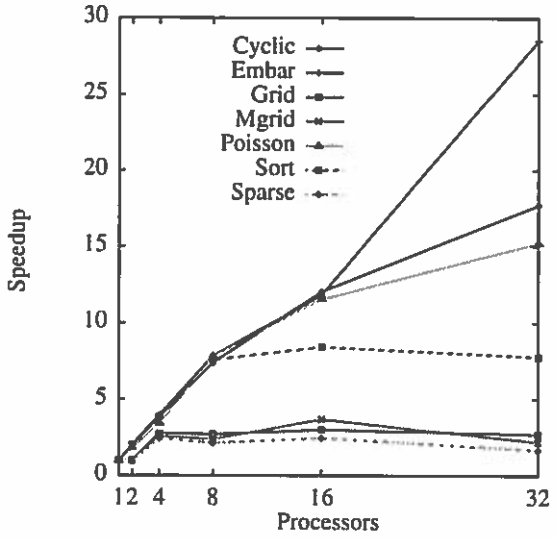
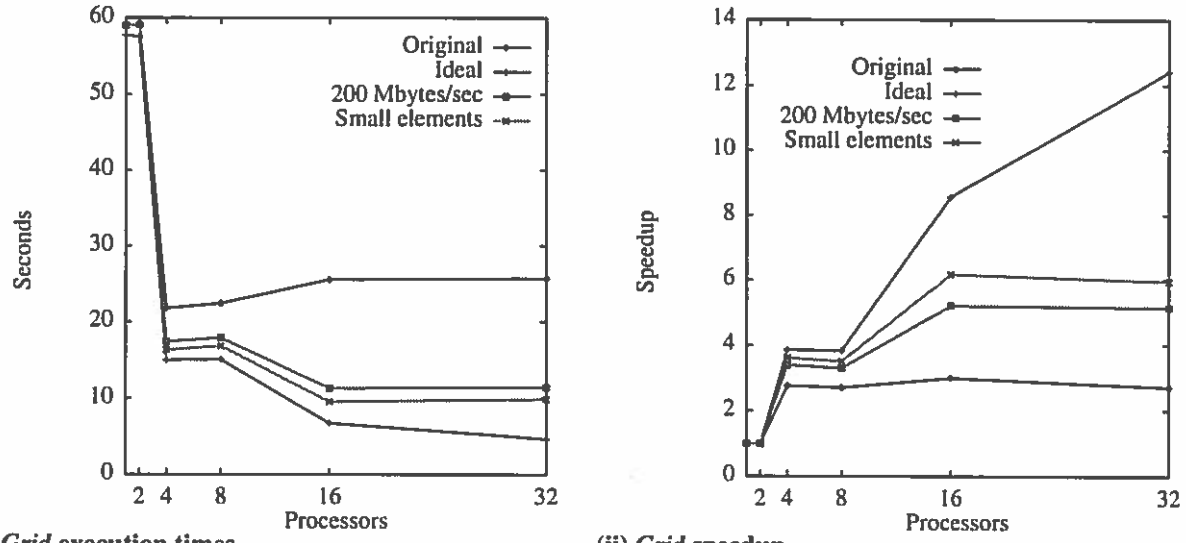


FIGURE 4. Speedup curves for all Benchmarks



(i) *Grid* execution times **(ii) *Grid* speedup**

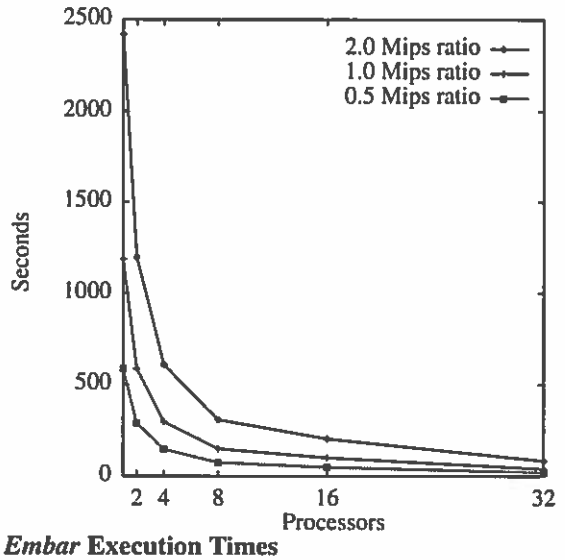
FIGURE 5. Comparison of Different Extrapolations

synchronization and communication costs were null. This produced close to the desired result, but trace statistics indicated that *Grid* does not have enough barriers (only 650) to make us concerned about synchronization overhead, and has sufficient computation that better speedup should be seen without resorting to ideal conditions, even in a distributed memory system.

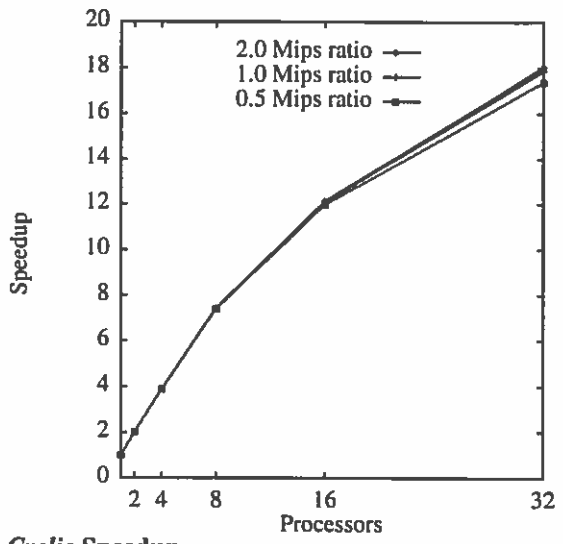
Further inspection of the trace files revealed the real problem. Our original trace measurements used high-level information from the *pC++* compiler to determine the size of remote element transfers -- a measurement abstraction that saves having to record this information in the trace. However, when generating calls to the runtime system during an optimization pass, the compiler determines if it can request part instead of the whole collection element, saving unnecessary data transfer. In *Grid*, the actual size of the remote data transferred is 2 and 128 bytes. Using high-level compiler information, our trace measurement associates the size of the grid collection element (231456 bytes) to each remote access. When we used the actual size in the simulation with the original parameter values, we achieved results comparable to the high-bandwidth test. By reducing the high communication start-up overheads, the speedup performance was further improved.

The important point of this exercise is that all of the experiments were done with the simulation system and single processor trace measurements. We were able to test easily the performance outcome of changes in the execution environment and observe the benefits of compiler optimizations. We are also able to see the effects of data distribution. For all *Grid* and *Mgrid* curves, we notice the odd behavior of no performance improvement from 4 to 8 processors. This is an artifact of the blocking distribution used for the two dimensional parallel grid data structures in *pC++* where a grid of size $P \times P$ is distributed in a (BLOCK, BLOCK) fashion to N processors such that each processor gets $\frac{P}{\sqrt{N}}$ grid elements. Notice that if N is not a perfect square, some processors will not get any elements at all. That is the reason why there is no performance improvement from 4 to 8 processors; 4 of the processors are sitting idle. Its appearance in the simulation results is rudimentary validation that extrapolation is able to capture important program execution characteristics.

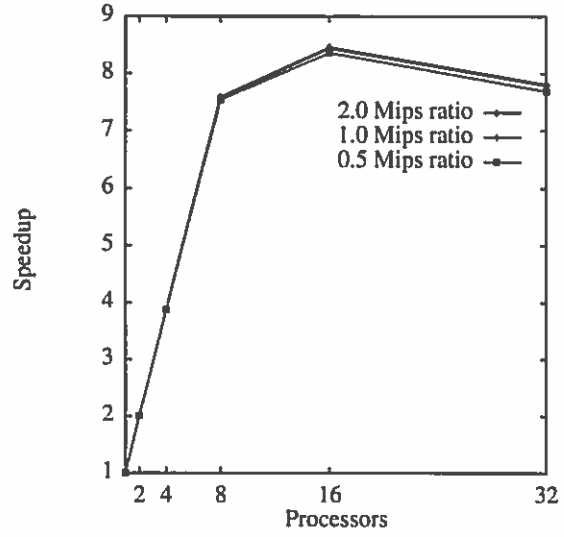
Our second experiment was designed to observe the effects of extrapolating processor speed. For all benchmarks, we



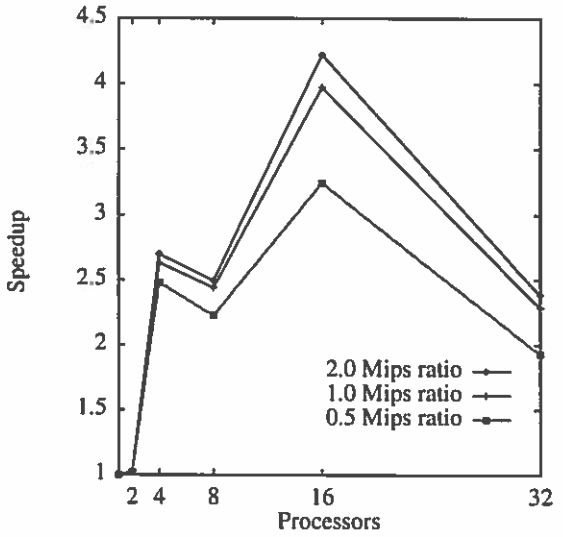
(i) *Embar* Execution Times



(ii) *Cyclic* Speedup



(iii) *Sort* Speedup



(iv) *Mgrid* Speedup

FIGURE 6. Execution Time and Speedup Results with Different *MipsRatio*

tested processor scalability with three *MipsRatio* parameter values: 2.0, 1.0, and 0.5. The value 1.0 indicates that the computation times measured in the 1-processor execution are unchanged in the simulation, 2.0 indicates a 2x

slowdown in computation, and 0.5 indicates a 2x speed increase. The expected slowdown and speedup performance effects were clearly seen in all benchmark extrapolation results, but different speedup behaviors arose depending on the effect of changes in the ratio of computation to communication. The *Embar* curves in graph (i) of Figure 6 highlight the increase in execution times for $MipsRatio = 2.0$ and the decrease for $MipsRatio = 0.5$. *Cyclic* and *Sort* speedup curves shown in graphs (ii) and (iii) of Figure 6, show little effect of varying *MipsRatio*. One would not expect this to be true, however, for codes with a major communication constituent. The influence of changing computation/communication ratio can clearly be seen in the *Mgrid* speedup curves, graph (iv) of Figure 6, whereas the performance effect of a growing communication bottleneck in *Poisson* is not significant until 32 processors.

To further explore the *Mgrid* results, we looked at the effect of *MipsRatio* on performance when communication parameter values are changed. Figure 7 plots the execution times for $MipsRatio = \{1.0, 0.25\}$ and

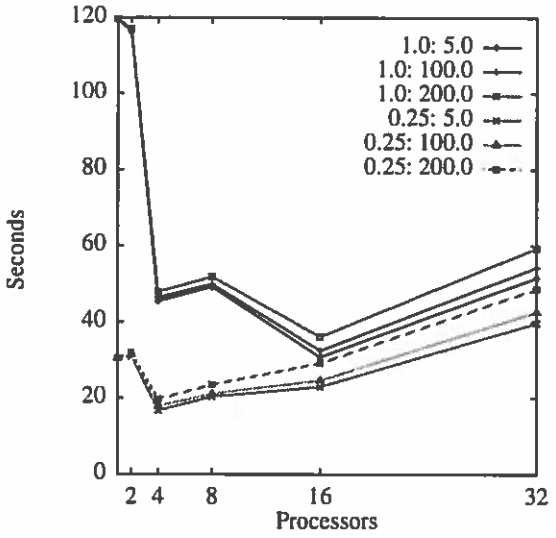
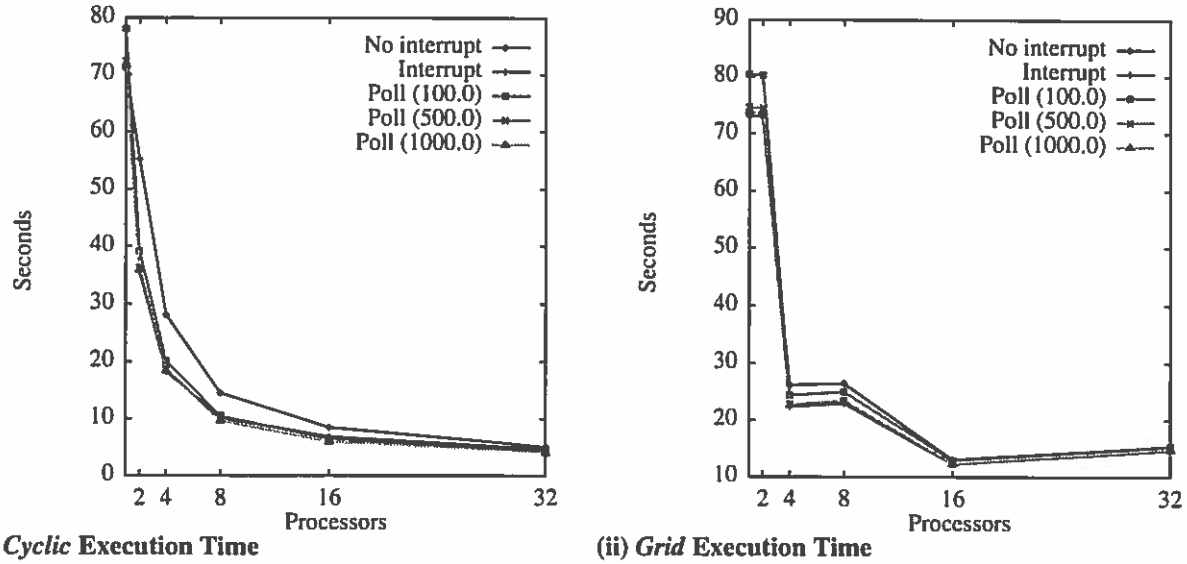


FIGURE 7. Effect of *MipsRatio* and *CommStartupTime* on *Mgrid*

$CommStartupTime = \{5.0, 100.0, 200.0\}$ μsec. The interesting performance behavior to note is the change in the number of processors delivering minimum execution time: 16 processors for $MipsRatio = 1.0$ and 4 processors for $MipsRatio = 0.25$. The obvious reason is the earlier influence of communication overhead when $MipsRatio = 0.25$. The ability to extrapolate target system parameters such as processor speed and communication start-up and to see such performance effects is important when "what if" questions are posed, especially about systems that do not physically exist.

The last experiment with the benchmarks that we report here (see [20] for more results) demonstrate *ExtraP*'s ability to simulate different runtime system policies for servicing remote data accesses. Two policies are supported in the actual *pC++* system: polling and interrupt. The choice of which policy to implement is usually based on whether the target machine environment supports message interrupts (e.g., active messages on the CM-5). However, we might be interested in the performance benefits of an interrupt policy versus a polling policy. For that matter, we might want to know how much of a performance improvement interrupt and polling policies give over just servicing remote requests while waiting for remote replies, at barriers, or at the end of compute phases. Also, if a polling policy must be used, a port of *pC++* requires the choice of polling interval. An optimal choice of the polling interval is certainly system and likely problem specific. All of these questions can be explored with extrapolation.

Figure 8 shows the execution time curves for the *Cyclic* and the *Grid* benchmarks when the remote data access policies are changed. (It should be kept in mind that the complete set of parameter values affect simulation results. Important parameters to keep in mind here are and $CommStartupTime = 100.0$) In both graphs, the "No interrupt/poll" curve performs the worst, as expected, but only by a maximum of 10% for all processors, in the case of *Grid*; in *Cyclic* the performance is significantly worse, but improves with larger numbers of processors. Hence, we can see from the extrapolation that program execution characteristics *do* affect the



(i) *Cyclic* Execution Time (ii) *Grid* Execution Time

FIGURE 8. Effects of Remote Data Request Service Policy

performance benefit of implementing an interrupt or a polling policy. What about the choice of interrupt versus polling, or the choice of polling interval? We might think that an interrupt policy would always perform best because remote data accesses would not be delayed by computation on the remote processor. In fact, we observe this result in the *Grid* graph. However, for *Cyclic*, it is interesting to see that a polling policy wins out for larger numbers of processors, even when the polling interval is as small as 100 µsec. Larger polling times perform better for *Cyclic* which might indicate that frequent interactions between the threads are enough to service remote element requests without the overhead of more frequent polling. The extrapolation process allows us to again observe the potential performance effects of changes in the target execution environment, this time at the runtime system level, and to possibly use that information to make application-specific runtime system optimizations.

4.2 Validation

For performance extrapolation to be an effective technique for performance debugging, it must be able to produce results that closely match the performance behavior found in actual target systems. In the case of *pC++*, there are many target systems, as the language is intended to be portable. In extrapolating to any particular one, or even to a hypothetical system, the key is to capture as best as possible the characteristics of the execution environment in the parameters used for extrapolation. The validity of the extrapolation results can then be evaluated in how accurately the predicted performance phenomena is representative of the real environment and can be applied to bottleneck analysis, algorithm design decisions, and program tuning. To validate *ExtraP*, we took a simple matrix multiplication program written in *pC++* and performed processor scaling experiments for different matrix distribution choices, extrapolating the performance to a CM-5 execution environment. The results are described below.

Matmul is a simple *pC++* program which multiplies two $N \times N$ matrixes A and B; B is given in transposed form. Both A and B^T are distributed in the same way. The first row of B^T is broadcast to all the rows of a temporary matrix T. A pointwise multiplication of A and T is then performed and the result is placed in another temporary matrix S. A right to left global summation (reduction) in each row of S produces the first column of the result matrix $A \bullet B$. This process is repeated for all the rows of B^T. Though *Matmul* is a naive matrix multiplication program, it serves to illustrate the usefulness of the extrapolation technique.

The program was run with nine different combinations of two-dimensional data distributions for the matrices, as determined by the per dimension distribution attributes available under the *pC++* compiler: **Block**, **Cyclic**, and **Whole**. The trace files were generated on a Sun 4 machine and then extrapolated using simulation parameters to match the CM-5. The predicted execution times from *ExtraP* and the actual results from CM-5 are shown in Figure 9. Some of the important parameters used for extrapolation are shown in Table 3. The processor speed ratio was

obtained by measuring the scalar MFLOPS of the CM-5 (2.7645) and the Sun 4 (1.1360) machine. The other parameters were obtained from [13,17].

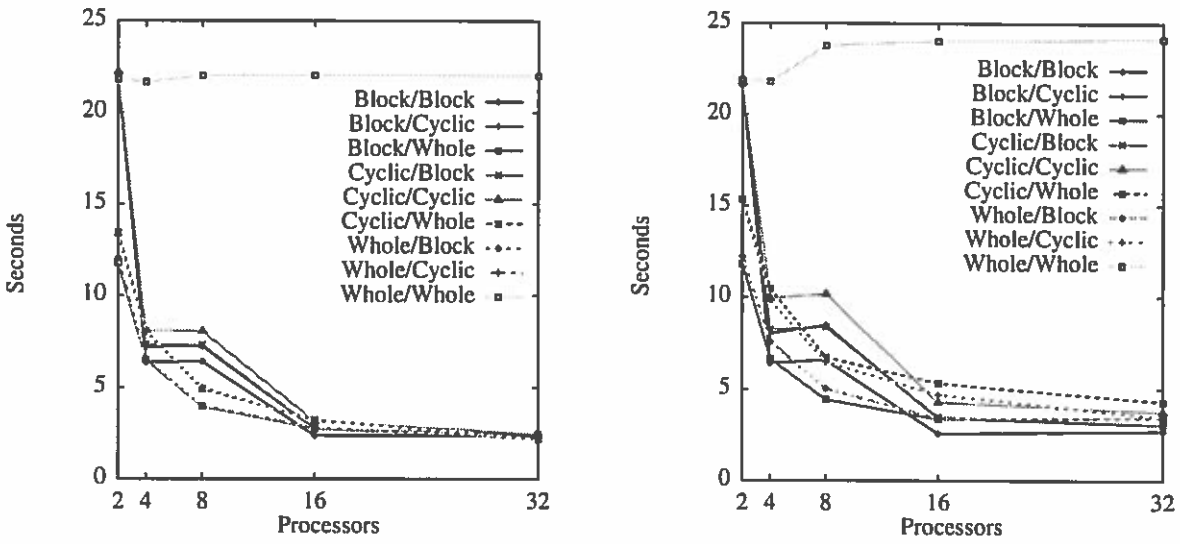


FIGURE 9. Results from *Matmul* program

Parameter	Value
BarrierModelTime	5.0 μsec
CommStartupTime	10.0 μsec
ByteTransferTime	0.118 μsec (8.5 Mbytes/second)
MipsRatio	0.41

TABLE 3. Parameters used for Matching CM-5 Characteristics

The extrapolation clearly brings out the effect of data distribution on the execution time of *MatMul*. In addition to matching the general shape of the actual curves, the predicted curves also reasonably match the relative ranking of the different distributions. The extrapolation picks out the same best choice as the measurement for all number of processors except 32, in which case the execution time of the predicted best choice in the actual machine is within 3% of the optimum. This demonstrates that extrapolation can capture the relative performance ordering of algorithm design choices and, thus, can be used to make optimization decisions during the performance tuning process.

Concerning actual execution times, the predicted values differ somewhat from the measured values. Although they are not excessive, certain errors are expected, considering the fact that a high-level simulation has been performed to achieve these results. Our opinion is that the shape and relative positioning of the curves is more important. The trade-off in accuracy, of course, can be found in the utility and speed of extrapolation. The ability of extrapolation to predict the results very quickly without compromising the relative ordering of various design choices makes it very attractive in a rapid prototyping environment.

5 Applicability to other Platforms

The extrapolation technique described here assumes that certain conditions hold during simulation. In particular, the order of a thread's measured events (i.e., remote element request, barrier synchronization) are assumed to be

unaffected by the remote data actions of other threads; hence, the reuse of thread traces. This assumption can be warranted by the actual execution environment or enforced in the simulation. For instance, *pC++* execution semantics were initially based on an *owners compute* data-parallel model, wherein only the thread that "owns" an element is able to modify its value. In addition to local computation, a thread then needs only to service read-only requests for remote elements from other threads. As a result, a thread's execution path (and, thus, the sequence of thread events) cannot be affected by the order or the time of inter-thread interactions (via remote element requests or barrier synchronizations), albeit its timing certainly can. In other language systems where such a "deterministic" execution model applies, the extrapolation techniques can be equally well applied.

The more intriguing issue is the application of extrapolation techniques to execution environments where the thread execution *can* be affected by inter-thread interactions. There are two general cases to consider. First is the case where the possibility exists in the target system that a thread's execution can be altered due to the effects and timing of interactions with other threads, but that this can be guaranteed to never occur in a program's execution in any environment. As an example, the current *pC++* language system does allow remote element write operations, but programs can easily be written that do not use remote writes (as is the case in our benchmarks) or that do not lead to timing-dependent execution behavior as a result of it. As a matter of fact, the modification to *ExtraP* to support remote element writes in this case is trivial. The second case is where thread event reordering can occur due to changes in the target execution environment. The principal question here concerns whether enough information is known about program execution behavior and enough measured performance data is available for the extrapolation to accurately determine when and which execution reorderings occur. The counter question is whether the extrapolated execution, under the assumption of no reordering of events, can ever occur in the target system. In certain cases, it may be possible to achieve a middle ground whereby the extrapolation is considered a "controlled execution" in the target environment based on the data ordering and synchronization constraints of an actual n-processor measured execution.

6 Future Work and Conclusions

Performance extrapolation uses performance information from one execution environment to predict the performance in another target environment. We have described a particular performance extrapolation technique that uses the execution time and events from a n-thread, 1-processor run to predict the execution time of a n-thread, n-processor run of the same program. Our experience suggests that performance extrapolation is a viable technique to be use in a performance debugging system, particularly for language environments, like that of *pC++*, where simple, restricted execution semantics make programs more amenable to performance prediction. Two important results of our research are that access to actual target platforms are not always required for accurate performance evaluation of parallel programs, and that low-level system simulations are not the only alternative required for detailed performance prediction. Performance extrapolation based on high-level events, like implemented in *ExtraP*, can support both diagnosis and tuning in a performance debugging system.

This work can be extended in a several ways. The simulation can be extended to handle multithreaded processors (i.e., the scheduling of more than one thread on a processor). This will extrapolate the performance from a n-thread, 1-processor run to a n-thread, m-processor run, where $m \leq n$. We are currently modifying *ExtraP* to support multithreading. Another direction is to apply this work to other language systems, like HPF. The focus of our present work is to integrate the *ExtraP* tool into the *pC++* program analysis environment.

7 References

[1] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, B. Mohr, Implementing a Parallel C++ Runtime System for Scalable Parallel Systems, Proc. Supercomputing 93, IEEE Computer Society and ACM SIGARCH, pages 588-597, November 1993.

[2] Z. Bozkus et al., Compiling Distribution Directives in a Fortran 90D Compiler, Technical Report SCCS-388, Northeast Parallel Architectures Center, July 1992.

[3] E. A. Brewer, C. N. Dellarocas, A. Colbrook, W. E. Weihl, Proteus: A High Performance Parallel Architecture Simulator, Technical Report MIT/LCS/TR-516, MIT Laboratory of Computer Science, September 1991.

[4] E. A. Brewer, W. E. Weihl, Developing Parallel Applications Using High-Performance Simulation, Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, pages 158-168, May 1993.

[5] M. E. Crovella and T. J. LeBlanc, Parallel Performance Prediction Using Lost Cycles Analysis, Proc. Supercomputing 94, IEEE Computer Society and ACM, pages 600-609, November 1994.

[6] P. Dickens, P. Heidelberger and D. Nicol, Parallelized Direct Execution Simulation of Message Passing Parallel Programs, Technical Report ICASE/94/94-50, Institute for Computer Applications in Science and Engineering, NASA Langley, 1994.

[7] T. Fahringer, Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers, Ph.D. Thesis, University of Vienna, September 1993.

[8] D. Gannon, F. Bodin, S. Srinivas, N. Sundaresan and S. Narayana, Sage++, An Object Oriented Toolkit for Program Transformations, Technical Report, Department of Computer Science, Indiana University, 1993.

[9] D. Gannon and J. K. Lee, Object Oriented Parallelism: *pC++* Ideas and Experiments, Proc. Japan Society of Parallel Processing, pages 13-23, 1991.

[10] D. C. Grunwald, A Users Guide to AWESIME: An Object Oriented Parallel Programming and Simulation System, Technical Report 552-91, Department of Computer Science, University of Colorado at Boulder, November 1991.

[11] R. Helm, A. D. Malony and S. F. Fickas, Capturing and Automating Performance Diagnosis: The Poirot Approach, Proc. International Parallel Processing Symposium

[12] High Performance Fortran Forum, High Performance Fortran Language Specification version 1.0, TR92225, Center for Research on Parallel Computation, Rice University, January 1993.

[13] T. T. Kwan, B. K. Totty and D. A. Reed, Communication and Computation Performance of the CM-5, Proc. Supercomputing 93, IEEE Computer Society and ACM SIGARCH, pages 192-201, November 1993.

[14] A. D. Malony, Event-Based Performance Perturbation: A Case Study, Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1991.

[15] A. D. Malony, V. Mertsiotakis and A. Quick, Automatic Scalability Analysis of Parallel Programs Based on Modeling Techniques, Computer Performance Evaluation - Modelling Techniques and Tools, Lecture Notes in Computer Science 794, Springer-Verlag, pages 139-158, 1994.

[16] A. D. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, Performance Analysis of *pC++*: A Portable Data-Parallel Programming System for Scalable Parallel Computers, Proc. IPPS International Parallel Processing Symposium, pages 75-85, April 1994.

[17] On-line information on CM-5, CM-5 Technical Summary, http://www.think.com/Prod-Serv/Products/cmmd.html, November 1992.

[18] D. K. Poulsen and P. C. Yew, Execution-Driven Tools for Parallel Simulation of Parallel Architectures and Applications, Proc. Supercomputing 93, IEEE Computer Society and ACM SIGARCH, pages 860-869, November 1993.

[19] 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, Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 48-60, May 1993.

[20] K. Shanmugam, Performance Extrapolation of Parallel Programs, Master's Thesis, Department of Computer and Information Science, University of Oregon, June 1994.

[21] H. Wabnig and G. Haring, PAPS - The Parallel Program Performance Prediction Toolset, Computer Performance Evaluation - Modelling Techniques and Tools, Lecture Notes in Computer Science 794, Springer-Verlag, pages 284-304, 1994.