Speedy: An Integrated Performance Extrapolation Tool for pC++ Programs†

Bernd W. Mohr, Allen D. Malony

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

Kesavan Shanmugam

Convex Computer Corp.
Richardson, TX 75083, USA
kesavans@convex.com

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 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. In pC++, the programmer can distribute a *collection* of objects to various processors and can have methods invoked on those objects execute in parallel. Using performance extrapolation in the development of pC++ applications allows tuning decisions to be made in advance of detailed execution measurements. The pC++ language system includes τ, an integrated environment for analyzing and tuning the performance of pC++ programs. This paper presents *speedy*, a new addition to τ, that predicts the performance of pC++ programs on parallel machines using extrapolation techniques. *Speedy* applies the existing instrumentation support of τ to capture high-level event traces 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 a target *n*-processor machine. We describe how *speedy* works and how it is integrated into τ. We also show how *speedy* can be used to evaluate a pC++ program for a given target environment.

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. For this purpose, the process of *performance debugging* (the iterative application of performance *diagnosis* [8] and *tuning*) is applied as an integral part of a parallel program development methodology. Application of performance debugging in practice has invariably required the development of performance tools based on the measurement and analysis of actual parallel program execution. Parallel performance environments [16] support performance debugging through program instrumentation, performance data analysis, and results presentation tools, but have often lacked in their integration with parallel programming systems. However, recent efforts on developing portable high-level parallel language systems has motivated work in integrated program analysis environments where performance debugging concerns are more closely coupled with the language's use for program development [10,14]. Of particular interest is the incorporation of performance prediction support in the programming environment for giving feedback to the user on algorithm implementation or to the compiler on optimization strategies [5,11]. In most instances, however, there is a dependence on actual machine access for performance debugging, restricting the parallel programmer

† This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135 and Fort Huachuca contract ARMY DABT63-94-C-0029.

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, whether by empirical evaluation or prediction based on measurement, is usually not possible.

Ideally, an integrated program analysis environment that supports performance debugging would include a means to predict performance where only limited access (if any) to the target system is given. The environment would measure only those performance data which is necessary, 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]. The user would demand a level of detail from predicted performance analysis comparable to that provided by measurements; however, static prediction tools often cannot provide this. Similarly, the user will be frustrated if the time taken to generate predicted results is significantly greater than the time taken by measurement-based experiments, a problem often faced by simulation systems that analyze program execution at too low a level. For example, the Proteus system [3] and the Wisconsin Wind Tunnel [17] have considerably advanced the efficiency and effectiveness of dynamic prediction techniques for architectural studies, but the overhead is still too high to allow 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 environment. In [20], we demonstrated 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, 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, yet at the same time retaining the flexibility and robustness of model-based prediction techniques. However, there remains the problem of how performance extrapolation can be seamlessly integrated in a parallel language system, where it both leverages and complements the capabilities of the program analysis framework.

We have integrated our performance extrapolation techniques into the τ program analysis environment for pC++, a data-parallel C++ language system. In Section 2, we describe the pC++ language and the features of τ to show how the environment can easily be extended to support performance prediction of pC++ programs. The performance extrapolation approach to pC++ prediction is discussed in Section 3. In Section 4, we show how the performance extrapolation techniques have been integrated into τ, in the form of the *speedy* tool. We performed several experiments which we used to validate Speedy's results (Section 5) and to evaluate its use for program tuning (Section 6). The paper concludes with a discussion on future work.

2 pC++ and TAU

In this section, we give a brief overview of τ, (TAU, for **T**uning and **A**nalysis **U**tilities), an integrated, portable program and performance analysis environment for pC++. pC++ is a language extension to C++ designed to allow programmers to compose distributed data structures with parallel execution semantics. The basic concept behind pC++ is the notion of a *distributed collection*, which is a structured set of objects which are distributed across the processing elements of the computer. To accomplish this, pC++ provides a simple mechanism to build *collections of objects* from a base *element* class. Member functions from this element class can be applied to the entire collection (or a subset) in parallel. This mechanism provides the user with a clean interface to *data-parallel* style operations by simply calling member functions of the base class. To help the programmer build collections, the pC++ language includes a library of standard collection classes that may be used directly or subclassed. This includes classes such as *DistributedArray*, *DistributedMatrix*, and *DistributedVector*.

pC++ and its runtime system have been ported to several shared and distributed memory parallel systems, validating the system's goal of portability. The ports include the KSR-1, Intel Paragon, TMC CM-5, IBM SP-1 / SP-2, Sequent Symmetry, SGI Challenge, Onyx, and PowerChallenge, Cray T3D, Meiko CS-2, Convex SPP and homogeneous clusters of UNIX workstations using PVM and MPI. pC++ also has multi-threading support for running applications in a quasi-parallel mode on UNIX workstations; supported thread systems are Awesime [6], Pthreads, LWP, and the AT&T task library. This enables the testing and pre-evaluation of parallel pC++ applications in a familiar desktop environment. More details about the pC++ language and runtime system can be found in [1,12].

τ provides a collection of tools with user-friendly graphical interfaces to help a programmer analyze the performance of pC++ programs. Elements of the τ graphical interface represent objects of the pC++ programming model: collections, classes, methods, and functions. These language-level objects appear in all τ tools. By plan, τ was designed and developed in concert with the pC++ language system. It leverages off pC++ language technology, especially in its use of the Sage++ toolkit [2] as an interface to the pC++ compiler for instrumentation and for accessing properties of program objects. τ is also integrated with the pC++ runtime system for profiling and tracing support. Because pC++ is intended to be portable, the tools are built to be portable as well. C++ and C are used to ensure portable and efficient implementation, and similar reasons led us to choose Tcl/Tk [15] for the graphical interface.

The τ tools are implemented as graphical *hypertools*. While the tools are distinct, providing unique capabilities, they can act in combination to provide enhanced functionality. If one tool needs a feature of another one, it sends a message to the other tool requesting it (e.g., display the source code for a specific function). With this design approach, the toolset can be easily extended. τ meanwhile has also been retargeted to other programming environments, including HPF.

One important goal in τ's development was to make the toolset as user-friendly as possible. For this purpose, many elements of the graphical user interface are analogous to *links* in *hypertext* systems: clicking on them brings up windows which describe the element in more detail. This allows the user to explore properties of the application by simply interacting with elements of most interest. The τ tools also support the concept of *global features*. If a global feature is invoked in any of the tools, it is automatically executed in all currently running tools. Examples of global features include locating information about a particular function or a class across all the tools.

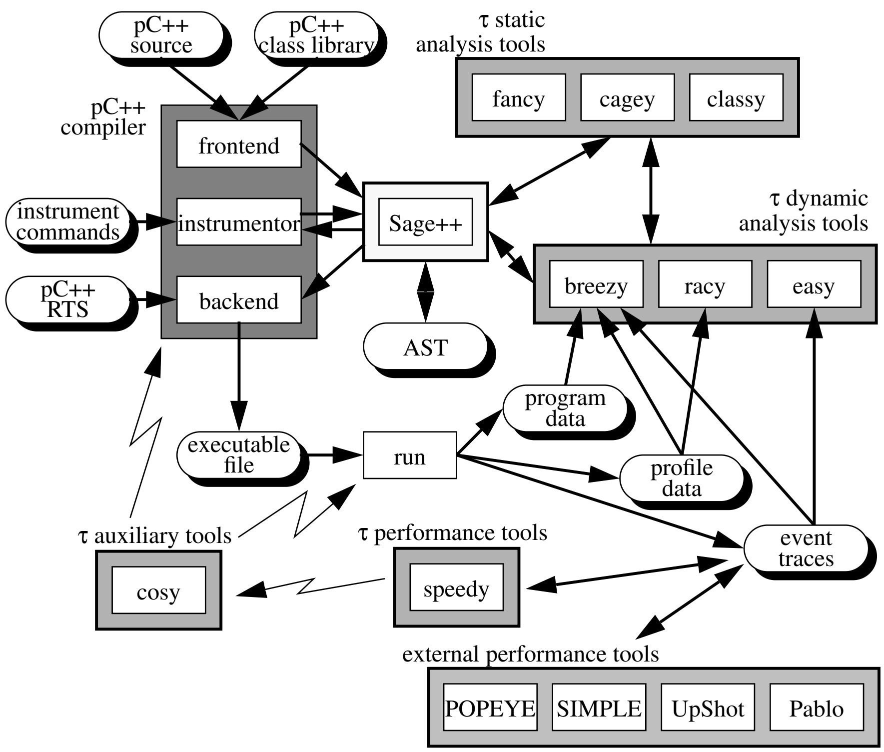


Fig. 1. pC++ Programming Environment and τ Tools Architecture

Figure 1 shows the pC++ programming environment and the associated τ tools architecture. The pC++ compiler frontend takes a user program and pC++ class library definitions (which provide the predefined collection types) and parses them into an *abstract syntax tree* (AST). All access to the AST is done via the Sage++ library. Through command line switches, the user can choose to compile a program for profiling, tracing, and breakpoint debugging. In these cases, the instrumentor is invoked to do the necessary instrumentation in the AST. The pC++ backend transforms the AST into plain C++ with calls to the pC++ runtime system. This C++ source code is then compiled and linked by the C++ compiler on the target system. The compilation and execution of pC++ programs can be controlled by *cosy* (**CO**mpile manager **S**tatus displa**Y**); see Figure 6, bottom.This tool provides a graphical interface for setting compilation and execution parameters. The program and performance analysis environment is shown on the right side of Figure 1. They include the integrated τ tools, profiling and tracing support, and interfaces to stand-alone performance analysis tools developed partly by other groups [7,9,13,16]. The τ toolset provides support both for accessing static program information and for analyzing dynamic data obtained from program execution.

2.1 Static Analysis Tools

One of the basic motivations behind using C++ as the base for a new parallel language is its proven support for developing and maintaining complex and large applications. However, to apply the C++ language capabilities effectively, users require support tools to manage and access source code at the level of programming abstractions.

Currently, τ provides three tools to enable the user to quickly get an overview of a large pC++ program and to navigate through it: the global function and method browser *fancy* (**F**ile **AN**d Class displa**Y**), the static callgraph display *cagey* (**CA**ll **G**raph **E**xtended displa**Y**), and the class hierarchy display *classy* (**CLASS** hierarch**Y** browser). The tools are integrated with the dynamic analysis tools through the global features of τ, allowing the user to easily find execution information about language objects. For instance, to locate the corresponding dynamic results (after a measurement has been made), the user only has to click on the object of interest (e.g., a function name in the callgraph display).

2.2 Dynamic Analysis Tools

Dynamic program analysis tools allow the user to explore and analyze program execution behavior. This can be done in three general ways. *Profiling* computes statistical information to summarize program behavior, allowing the user to find and focus quickly on the main bottlenecks of the parallel application. *Tracing* portrays the execution as a sequence of abstract *events* that can be used to determine various properties of time-based behavior. *Breakpoint debugging* allows a user to stop the program at selected points and query the contents of program state. For all analysis modes, the most critical factor for the user is how the high-level program semantics are related to the measurement results. τ helps in presenting the results in terms of pC++ language objects and in supporting global features that allow the user to locate the corresponding routine in the callgraph or source text by simply clicking on the related measurement result or state objects.

τ's dynamic tools currently include an execution profile data browser called *racy* (**R**outine and data **AC**cess profile displa**Y**), an event trace browser called *easy* (**E**vent **A**nd **S**tate displa**Y**), and a breakpoint debugger called *breezy* (**BR**eakpoint **E**xecutive **E**nvironment for visuali**Z**ation and data displa**Y**). A more detailed discussion of the τ tools can be found in [4,12,14].

3 ExtraP - A Performance Extrapolation Tool for pC++

ExtraP is a performance extrapolation system for pC++ that has been integrated into τ in the guise of *speedy*. This section explains the modeling approach of extrapolation and the techniques used by *ExtraP*.

3.1 Performance Extrapolation

Performance extrapolation (Figure 2) 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. As used above, an execution environment embodies the

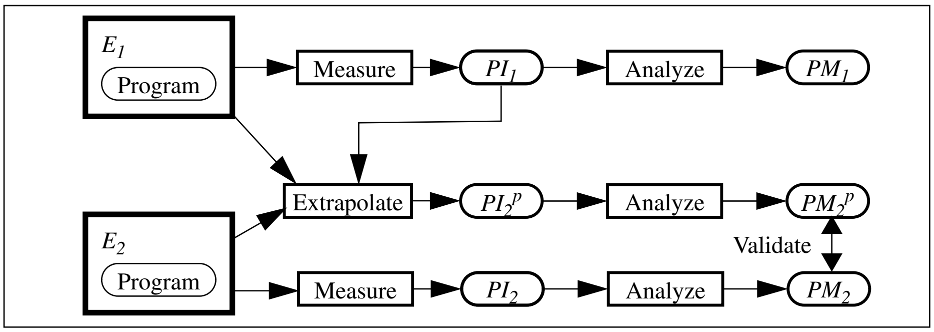


Fig. 2. Performance Extrapolation

collection of compiler, runtime system, and architectural features that interact to influence the performance of a parallel program. If in addition the execution environment E_2 is physically available, it can be used to validate the predicted results PM_2 by comparing them against the measured results PM_2.

3.2 A Performance Extrapolation Technique for pC++

We have developed a technique that extrapolates the performance of a *n*-thread pC++ program from a 1-processor execution to a *n*-processor execution. In this technique, a *n*-thread *pC++* program is executed on a single processor using a non-preemptive threads package. Important high-level events including remote accesses and barriers are recorded along with timestamps during the program run in a trace file. The instrumented runtime system is configured such that these remote accesses are treated as taking place instantaneously and the threads are seen to be released from a barrier as soon as the last thread enters it. Such a trace file captures the order of events in a pC++ program along with the computation time between the events, but leaves the actual timing of the events for later extrapolation analysis.

The events are then sorted on a per thread basis, adjusting their timestamps to reflect concurrent execution. This is possible because the non-preemptive threads package switches the threads only at synchronization points, and because global barriers are the only synchronization used by pC++ programs. This imposes a regular structure to the trace file where each thread records events between the exit from a barrier and an entry into another without being affected by another thread. The sorted 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. For example, the timings for remote accesses and barriers are absent in these trace files. A trace-driven simulation using these trace files attempts to model such 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 is depicted in Figure 3. For more details refer to [18,19,20]. The next section explains the various models used for trace-driven simulation.

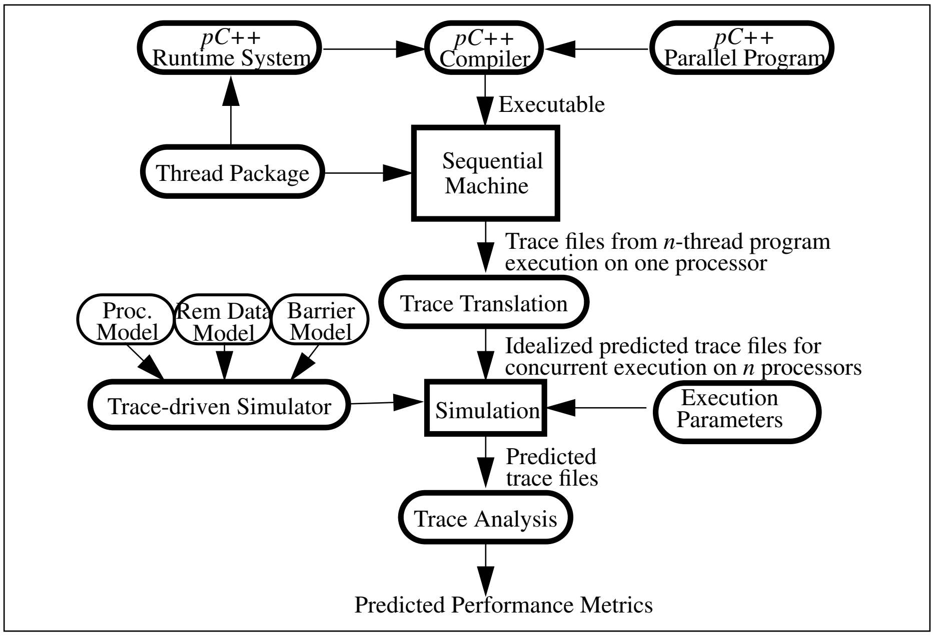


Fig. 3. A Performance Extrapolation Technique for *pC++*

3.3 Simulation Architecture and Models

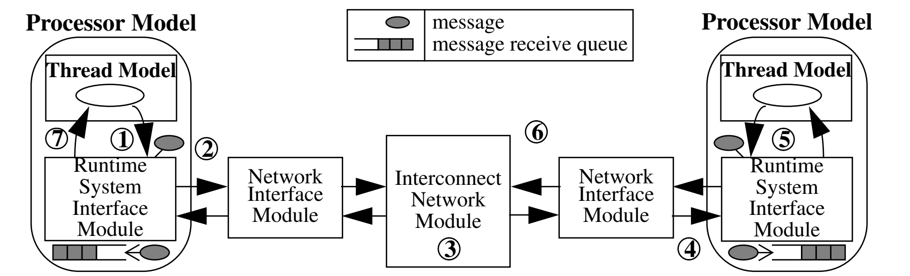
The trace-driven simulation is the heart of the pC++ performance extrapolation. The simulation system consists of three main components: the *processor*, the *remote data access*, and the *barrier* model. The *processor* model uses a simple ratio of processor speeds to scale the computation time between events from the measurement environment to the target environment. It is also responsible for choosing a policy for servicing remote data references; when a request for data is submitted by a remote thread, a processor can service it in three different ways:

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 *remote data access* model determines how a remote data access is translated into messages and how it is handled by the various components in the system. During the simulation each remote access in the program is modelled as a remote request for data from one thread to the thread that "owns" the data. 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 4



1. remote reference made by thread; "get" message created in RSIM which models runtime overhead
2. "get" message passed to NIM which models transfer to network (setup overhead and latency)
3. "get" message passed to INM which models network delay; message delivered to receiving NIM
4. NIM models transfer of "get" message to runtime system message receive queue
5. RSIM models polling or interrupt driven message handling; "element" message created in RSIM
6. transfer of "element" message to RSIM of requesting thread modeled by NIM and INM
7. RSIM models message handling while waiting for a reply; "element" data passed to thread

Fig. 4. Remote Data Access Model

graphically depicts the how the remote data accesses are processed in the simulation using messages. The *remote data access* model itself is composed of three subcomponents: the *runtime system*, the *network interface*, and the *interconnect network* model. Each of these models have various parameters that represent the characteristics of the remote data access model. For example, the *interconnect network* model includes the latency, bandwidth of the target platform as its parameters.

ExtraP uses a linear, master-slave barrier model to handle the barrier events. 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 parameters in the barrier model can be controlled so that hardware barriers or barriers implemented through shared memory can be 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.

All models described above have a variety of parameters that can be tuned to match a specific target environment. For example, Table 1 lists the parameters used in the *barrier* model and their sample values. For a complete list of parameters, refer to [18,20]. The next section explains how these parameters can be set and the extrapolation experiment carried out using τ. A new addition to τ called *speedy*

Parameter	Description	Example
EntryTime	Time for each thread to enter a barrier.	5.0 µsec
ExitTime	Time for each thread to exit the barrier.	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 it.	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 barriers.	1
BarrierMsgSize	Size of a message used for barrier synchronization.	16

Tab. 1. Parameters for the Barrier Model

interacts with *ExtraP* to perform the necessary extrapolation experiments. This integration of *ExtraP* with τ is import because *ExtraP* is intended to be used as part of a program analysis environment to provide a performance debugging methodology.

4 Integrating TAU and ExtraP

ExtraP is integrated with in two ways: First, for generating the traces needed for the simulation, *ExtraP* uses the extensive built-in event tracing system of pC++. The pC++ compiler has an integrated instrumentor module for automatic instrumentation of the application program. If necessary, the user can also use an instrumentation command file to selectively insert additional trace recording instructions. In addition, there are instrumented versions of the predefined pC++ class/collection library and the pC++ runtime system. The inserted event markers in the different modules of the system are assigned to *event classes* (e.g., user, runtime, collections) which can be activated or deactivated separately at runtime. Event tracing is fully operational on all parallel computer systems supported by pC++. This has several advantages for an *ExtraP* user. The traces used for simulation can be analyzed with all the event trace browsers supported by τ (currently *easy*, *Pablo* [16], *SIMPLE* [13], and *upshot* [9]). As the *ExtraP* model is based on the operational characteristics of pC++ event classes, the user can also generate semantically equivalent traces on real parallel computer systems for comparing or validating the extrapolation results. Finally, the users can use τ's integrated performance analysis tool, *racy*, to analyze their program execution and compare these to the simulated results. For example, in Figure 5, *racy* displays performance results for the pC++ Poisson benchmark (used in Section 6) executed on a 8 processor SGI PowerChallenge. *Racy* measures and displays function execution time profiles (shown on the left) and local/remote data access ratios (on the right).

Second, the actual extrapolation experiments can be controlled through a new τ tool *speedy* (**S**peedup and **P**arallel **E**xecution **E**xtrapolation **D**ispla**Y**). Pressing the *speedy* button in the τ main control window (see Figure 6, top), brings up its main control panel (see Figure 7). Here, the user can control the compilation of the specified pC++ object program, specify the parameters for the extrapolation model and the

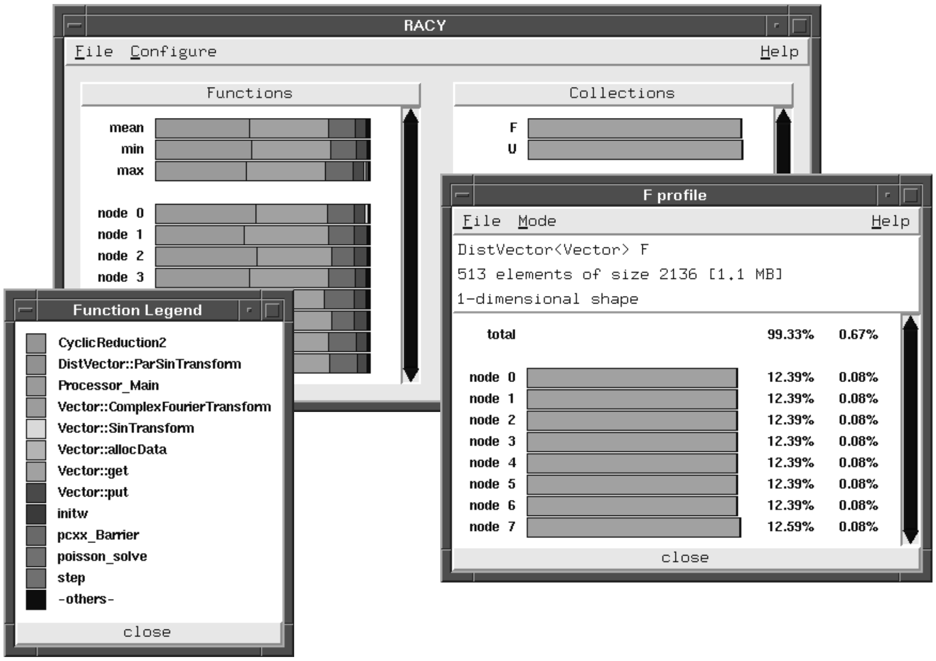


Fig. 5. RACY Performance Analysis Display for Poisson Benchmark

experiment, execute the experiment, and finally view the experiment results. *Speedy* uses *cosy* (see Figure 6, bottom) for automatically performing the necessary compilation, execution, trace processing, and extrapolation commands. *Speedy* also automatically keeps track of all parameters by storing them in *experiment description files* and managing all necessary trace and experiment control files. By loading a former experiment description file into *speedy*, the user can re-execute the experiment or just reuse some of the parameter specifications.

In Figure 7, the user specified an complex experiment where the value of the parameter "*Number of Processors*" is stepping through powers of two starting from one to thirty-two. In addition, the parameter "*Latency*" should vary between 10 and 100 in steps of 10. After each iteration of the extrapolation, the execution time as well as the speedup graph is updated. The user can also perform smaller experiments by specifying a special value `-NONE-` as the second or for both varying parameters.

The experiment and extrapolation model parameters can be entered and viewed through the *ExtraP* parameter file viewer (see Figure 8). Numerical parameters can either be entered directly into the input entry or manipulated through a slider bar and increment / decrement buttons. Parameters with discrete values can be specified through a pull-down menu (like *ProcessMsgType* in the picture). In Figure 8, the viewer displays the parameters associated with the modeling of the processor of the target machine. Other modeling parameter groups can be displayed by pressing one of the buttons in the top of the viewer window. Besides the five parameter groups described in Section 3.3, the group "General" allows the setting of parameters controlling the generation and post-processing of the execution traces.

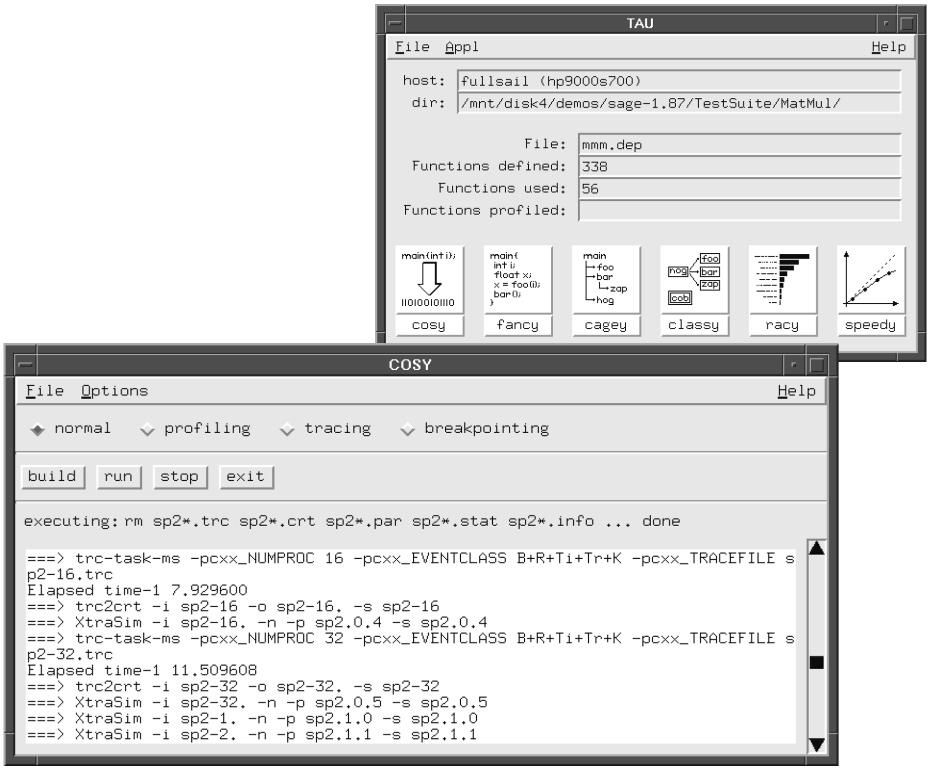


Fig. 6. TAU Main Control Window and COSY

5 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 target system, 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. To validate *ExtraP*, we took a simple matrix multiplication program written in pC++ (*MatMul*) and performed processor scaling experiments for different matrix distribution choices, extrapolating the performance to a CM-5 execution environment. 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.

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

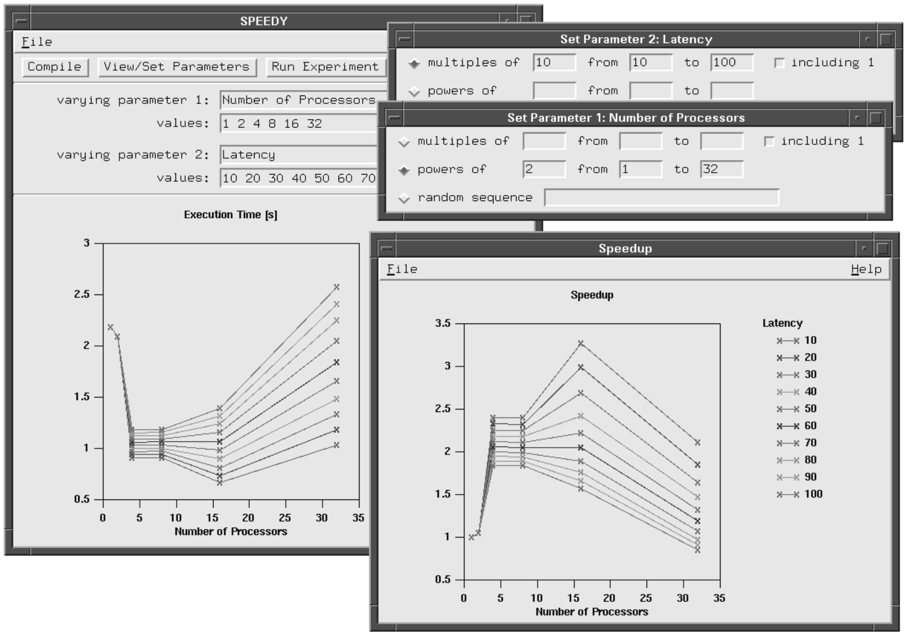


Fig. 7. SPEEDY Main Control Panel

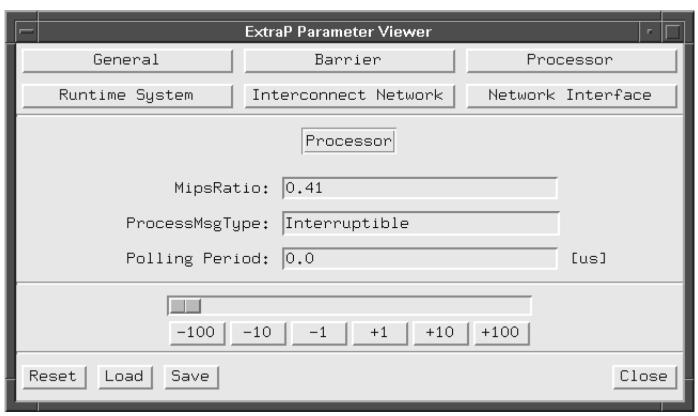
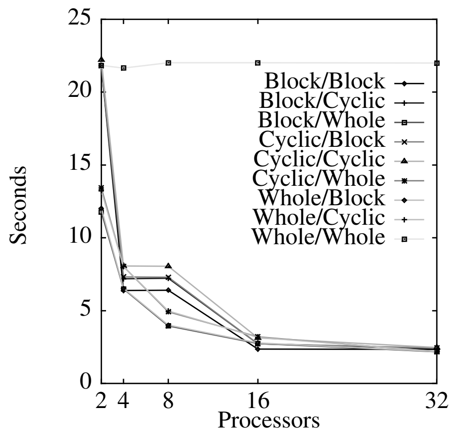


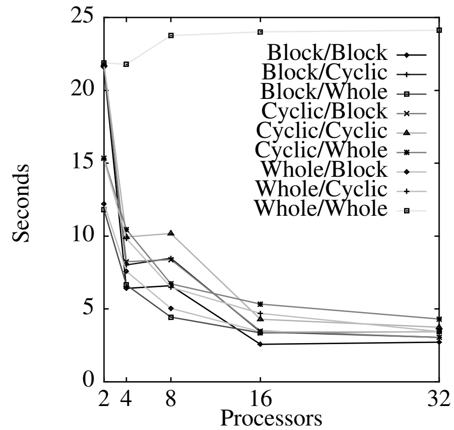
Fig. 8. ExtraP Parameter File Viewer

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



(i) Measured Times



(ii) Predicted Times

Fig. 9. Results from *MatMul* Program

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.

6 Experiments

In this section, we will see how *speedy* can be used during the development cycle of pC++ programs. *Speedy* allows the programmer to explore various design choices during the development process itself. *Speedy* also provides a framework in which the user can study different parts of the program and their contributions to the performance of the program. In this sense, *speedy* can be used as a profile prediction tool for parallel programs. *Speedy* allows all of this to be done from a workstation environment without ever having to run the programs on target machines.

Our first experiment is designed to show how various design choices can be made using *speedy*. This can be easily seen in the *MatMul* matrix multiplication program we used for validating the *speedy* tool (see Figure 9, right). Using a (WHOLE/WHOLE) distribution for the data is obviously a bad choice. In general, the predicted results suggest to use a (BLOCK, WHOLE) distribution. The results also show that (BLOCK, BLOCK) wins if the number of processors is a perfect square which even beats (BLOCK, WHOLE) on 16 processors. Such crossover point information is very useful for the programmer during the development process.

The goal of our next experiment is to show how *speedy* can be used to selectively study various portions of the program. Such profile information is useful when the programmer wants to tune parts of the program for a particular machine. We used the pC++ version of the NAS benchmark *Poisson* as the test case. It is a fast poisson solver which uses FFT based sine transforms together with a cyclic reduction algorithm to solve PDEs. We used τ to selectively instrument the code for the transforms and cyclic reduction. After extrapolating the performance to a CM-5 architecture, *speedy* predicted the results shown in Figure 10. While the code for sine

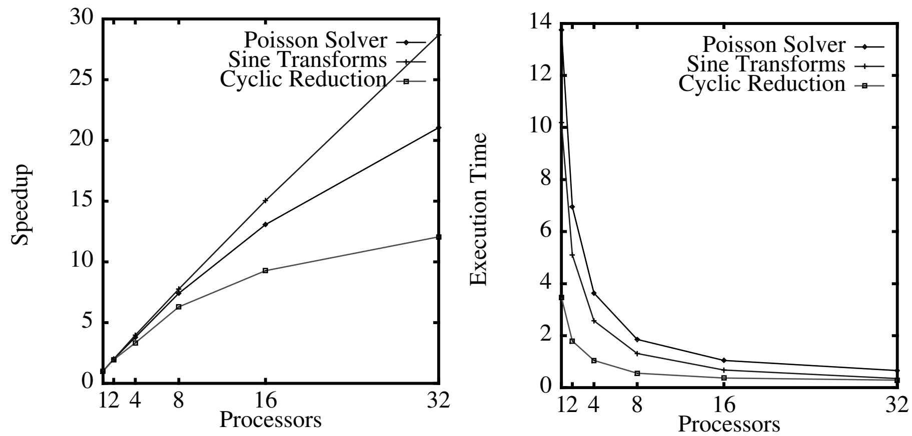


Fig. 10. Predicted Results for Poisson

transforms scales up very well with a speedup of 28.69 for 32 processors, the speedup curve for cyclic reduction starts to flatten after 16 processors. A further study of the trace files revealed that there are no remote accesses in the sine transform part of the poisson solver which accounts for its near-linear speedup. In contrast, the number of remote accesses in cyclic reduction increases as the number of processors thus affecting the performance badly. The overall speedup for *Poisson* is predicted to be in between that of sine transforms and cyclic reduction. This experiment tells us that to improve the performance of Poisson, we must tune Cyclic reduction first because it is the bottleneck. *Speedy* can be used in this way to locate bottlenecks in a program. The performance behavior observed using *speedy* is consistent with actual results [12].

7 Conclusion

The *speedy* and *ExtraP* tools are representative of the level of parallel performance evaluation support that is expected to be available for high-level parallel languages and to be integrated in program analysis environments where a performance engineered code development process is desired. The requirement for performance prediction as part of this process, is driven by the need to evaluate parallel codes that are intended to be ported to different execution platforms. Furthermore, the integration aspects (e.g., of merging *ExtraP* into τ) are of key importance as the extrapolation techniques must utilize the compiler, runtime system, and tool infrastructure to make the application of performance prediction in parallel code development feasible.

Our future work will concentrate on making the *ExtraP* technology more robust with additional models so that the different target system environments can be better represented. We also intend to extend the capabilities of the *speedy* tool to provide more support for automated performance experimentation and to better link the τ analysis and visualization tools to the performance data that *ExtraP* produces.

Documentation, technical papers, and source code for pC++, Sage++, and τ are available via FTP from `ftp://ftp.extreme.indiana.edu/pub/sage` or via WWW at the URLs `http://www.extreme.indiana.edu/sage` and `http://www.cs.uoregon.edu/paracomp/tau`.

8 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, pp. 588-597, November 1993.

[2] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, B. Winnicka, Sage++: An Object Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools, Proc. Oonski '94, Oregon, 1994.

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

[4] D. Brown, S. Hackstadt, A. Malony, B. Mohr, Program Analysis Environments for Parallel Language Systems: The TAU Environment, Proc. of the Workshop on Environments and Tools For Parallel Scientific Computing, Townsend, Tennessee, pp. 162-171, May 1994.

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

[6] 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.

[7] S. Hackstadt, A. Malony, Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development, Proc. Parallel Architectures and Languages Europe, (PARLE), Athens, Greece, 1994.

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

[9] V. Herrarte, E. Lusk, Studying Parallel Program Behavior with Upshot, Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne Natl. Lab., 1991.

[10] S. Hiranandani, K. Kennedy, C.-W. Tseng, S. Warren, The D Editor: A New Interactive Parallel Programming Tool, Proc. Supercomputing'94, IEEE Computer Society Press, pp. 733-742, November 1994.

[11] J. Kohn and W. Williams, ATExpert, Journal of Parallel and Distributed Computing, Vol. 18, 1993, pp. 205-222.

[12] A. 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. 8th Int. Parallel Processing Symb. (IPPS), Mexico, IEEE, pp. 75-85, April 1994.

[13] B. Mohr, Standardization of Event Traces Considered Harmful or Is an Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible?, Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, Elsevier, Advances in Parallel Computing, Vol. 6, pp. 103-124, 1993.

[14] B. Mohr, D. Brown, A. Malony, TAU: A Portable Parallel Program Analysis Environment for pC++, Proc. of CONPAR 94 - VAPP VI, Linz, Austria, Springer Verlag, LNCS 854, pp. 29-40, September 1994.

[15] J. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, 1994.

[16] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Birkett, D. W. Jensen, B. A.A. Nazief, B. K. Totty, Scalable Performance Environments for Parallel Systems. Proc. 6th Distributed Memory Computing Conf., IEEE Computer Society Press, pp. 562-569, 1991.

[17] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis and D. A. Wood, The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, Proc. ACM SIGMETRICS Conf. on Measurement and Modeling of Comp. Systems, pp. 48-60, 1993.

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

[19] K. Shanmugam, A. Malony, Performance Extrapolation of Parallel Programs, Proc. ICPP'95.

[20] K. Shanmugam, A. Malony, B. Mohr, Performance Extrapolation of Parallel Programs, Technical Report CIS-TR-95-14, University of Oregon, Department of Computer and Information Science, May 1995.

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