SANDIA REPORT

SAND2003-8631
Unlimited Release
Printed November 2003

Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment: A Case Study

J. Ray, N. Trebon, S. Shende, R. C. Armstrong, A. Malony

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone:	(865)576-8401
Facsimile:	(865)576-5728
E-Mail:	reports@adonis.osti.gov
Online ordering:	http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone:	(800)553-6847
Facsimile:	(703)605-6900
E-Mail:	orders@ntis.fedworld.gov
Online order:	http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-8631
Unlimited Release
Printed November 2003

Performance Measurement and Modeling of Component Applications in a High Performance Computing Environment : A Case Study

J. Ray, N. Trebon and R. C. Armstrong
HP Computing and Networking Department,
Sandia National Laboratories,
P.O. Box 969, Livermore, CA 94551-0969

S. Shende, and A. Malony
Department of Computer and Information Science,
1202 University of Oregon,
Eugene, OR 97403

Abstract

We present a case study of performance measurement and modeling of a CCA (Common Component Architecture) component-based application in a high performance computing environment. We explore issues peculiar to component-based HPC applications and propose a performance measurement infrastructure for HPC based loosely on recent work done for Grid environments. A prototypical implementation of the infrastructure is used to collect data for a three components in a scientific application and construct performance models for two of them. Both computational and message-passing performance are addressed.

Acknowledgments

This work was supported by the Department of Energy, Office of Science, via the Scientific Discovery through Advance Computing program at Sandia Nationa Loboratories, Livermore. The authors would like to thank the Department of Computer Science, University of Oregon, Eugene, for letting us use their "neuronic" cluster for our runs .

Contents

This page intentionally left blank

1 Introduction

The performance of scientific simulations in high performance computing (HPC) environments is fundamentally governed by the (effective) processing speed of the individual CPUs and the time spent in interprocessor communications. On individual CPUs, codes typically execute floating point operations on large arrays. The efficiency of such computations is primarily determined by the performance of the cache (in cache-based RISC and CISC processors) and much effort is devoted to preserving data locality. Interprocessor communications, typically effected by message passing in distributed memory machines (MPPs and SMP clusters), forms the other source of performance issues in HPC. Communication costs determine the load-balancing and scalability characteristics of codes and a multitude of software and algorithmic strategies (combining communication steps, minimizing/combining global reductions and barriers, overlapping communications with computations, etc.) are employed to reduce them.

The discipline of performance measurement has provided us with tools and techniques to gauge the interactions of scientific applications with the execution platform. Typically, these take the form of high precision timers which report the time taken to execute sections of the code and various counters which report on the behavior of various components of the hardware as the code executes. In a parallel environment these tools track and report on the size, frequency, source, destination and the time spent in passing messages between processors [1, 2, 3]. This information can then be used to synthesize a performance model of the application on the given platform - in some cases, these models have even served in a predictive capacity [4, 5].

In order to manage the growing complexity of scientific simulation codes, there has been an effort to introduce component-based software methodology in HPC environments. Popular component models like Java Beans [6] and CORBA [7] are largely unsuitable for HPC [8] and a new light-weight model, called the Common Component Architecture (CCA) [9] was proposed. The principal motivations behind the CCA are to promote code reuse and interdisciplinary collaboration in the high performance computing community. The component model consists of modularized components with standard, well-defined interfaces. Since components communicate through these interfaces, program modification is simplified to modifying a single component or switching in a similar component without affecting the rest of the application. To build a CCA application, an application developer simply composes together a set of components using a CCA-compliant framework. Details regarding the flexibility, performance and design characteristics of CCA applications can be found in [10].

While monolithic applications are hand-tooled under common assumptions and data structures to deliver maximum performance, component-based applications are composed out of standalone components, an injudicious selection of which can result in a correct but sub-optimal component assembly. It thus becomes imperative to be able to classify the performance characteristics and requirements of each implementation of a component and to have a generalized means of synthesizing a composite performance model to judge the optimality of a component assembly.

While this does not affect the fundamental performance issues in HPC, it does raise new challenges. Unlike monolithic codes, component-based software is seldom used exclusively by the authors of the components themselves and manual instrumentation of the code is impossible. Further, in a CCA environment, the final application is assembled at run time by loading shared libraries [8]; thus automatic instrumentation of an executable where a binary is rewritten or instrumented at runtime [11] has little meaning. Consequently, a non-intrusive strategy for performance monitoring is clearly indicated. Further, each component needs to be monitored to collect not only execution time and the hardware characteristics, but also the relevant inputs (like the size of arrays) that determine the data that was collected. These data then need to be synthesized into individual component performance models which then are constituted into a composite performance model for the applications using a component call-path. It is this model synthesis at the component level that holds promise for automating performance tuning in applications composed of components.

Since a containing framework creates, configures and assembles components, the framework possesses the global understanding of how the components are networked into an application. Similarly, the framework can compose a performance model of the entire application by combining the models of the participating components. This composite performance model is a dual of the application itself. This holds the promise of being the "cost function" in an optimization process by which the optimal (from the performance point of view) component application is assembled from multiple implementations of each component. The actual component encapsulates numerical and data management algorithms for use in the computation while its performance model encapsulates its predicted performance as a function of the high performance environment in which it is to be executed. This reasoning extends to the application component ensemble whose performance may be predicted by the composition model, but will be ultimately determined by the runtime conditions. The material presented in this work is far from realizing this goal, but it is essential that it be viewed as step toward realizing a completely automated system for performance prediction – and hence optimization – of high performance component based applications.

In this paper we examine some performance issues peculiar to HPC component environments. Section 2 provides a brief summary of performance measurement and modeling approaches in various component environments. Section 3 elaborates on some performance metrics specific to HPC while Section 4 describes the software infrastructure needed to measure these metrics non-intrusively. Section 5 describes a case study where we measure and model the performance of three components in a scientific simulation of a shock interacting with an interface between two gases. Our concluding remarks and future directions are given in Section 6.

2 Related Work

The three most widely-used component standards (CORBA [7], COM/DCOM [12], Java Beans [6]) are ill-suited to handle high performance scientific computing due to a lack of support for efficient parallel communication, insufficient scientific data abstractions (e.g., complex numbers), and/or limited language interoperability [9]. Thus, performance metrics developed for these environments are inadequate for HPC. In the serial environment to which these commercial component models are targeted, there is little reason for the design to account for details of hardware and memory hierarchy performance, yet this is a critical requirement in HPC. Often these distributed frameworks/component models (e.g. DCOM, CORBA CCM) use commodity networking to connect components together, entirely inadequate for HPC. In a distributed environment, metrics like round trip time and network latency are often considered useful, while quantities like bisection bandwidth, message passing latencies and synchronization cost, which form the basis of much of the research in HPC are left unaddressed. This primarily arises from the very different platforms that HPC and commercial component based applications run on - HPC is done almost exclusively on tightly-connected clusters of MPPs (massively parallel processors) or SMPs (Symmetric Multiprocessors) while commercial codes often operate on LANs (Large Area Networks) or WANs (Wide Area Networks).

However, despite the different semantics, several research efforts in these standards offer viable strategies in *measuring* performance. A performance monitoring system for the Enterprise Java Beans standard is described in [13]. For each component to be monitored, a proxy is created using the same interface as the component. The proxy intercepts all method invocations and notifies a monitor component before forwarding the invocation to the component. The monitor handles the notifications and selects the data to present, either to a user or to another component (e.g., a visualizer component). The goal of this monitoring system is to identify hot spots or components that do not scale well.

The Wabash tool [14, 15] is designed for pre-deployment testing and monitoring of distributed CORBA systems. Because of the distributed nature, Wabash groups components into regions based on the geographical location. An interceptor is created in the same address space of each server object (i.e., a component that provides services) and manages all incoming and outgoing requests to the server. A manager component is responsible for querying the interceptor for data retrieval and event management.

In the work done by the Parallel Software Group at the Imperial College of Science in London [16, 17], the research is focused on grid-based component computing. However, the performance is also measured through the use of proxies. Their performance system is designed to automatically select the optimal implementation of the application based on performance models and available resources. With n components, each having C_i implementations, there is a total of $\Pi_{i=1}^{n} C_i$ implementations to choose from. The performance characteristics and a performance model for each component is constructed by the component developer and stored in the component repository. Their approach is to use the proxies to simulate an application in order to determine the call-path. This simulation skips the implementation of the components by using the proxies. Once the call-path is determined, a recursive composite performance model is created by examining the behavior of each method call in the call-path. In order to ensure that the composite model is implementation-independent, a variable is used in the model whenever there is a reference to an implementation. To evaluate the model, a specific implementation's performance model replaces the variables and the composite model returns an estimated execution time or estimated cost (based on some hardware resources model). The implementation with the lowest execution time or lowest cost is then selected and a execution plan is created for the application.

3 Performance Measurements in HPC Component Environments

Component-based environments place very different requirements on performance measurement and modeling (PMM). Traditionally, PMM has been viewed as an analysis-and-optimization phase done by the code developers when a stable code base was ported to a new architecture. Emphasis was laid on extensive instrumentation and analysis to gauge the behavior of the application on the architecture and to optimize it. Synthesizing a performance model from such data to serve as a predictive tool was usually done from a scaling point of view [4, 5].

PMM plays a different role in component-based software. Since applications are dynamically composed at runtime, PMM can only be done in advance at a component-level. Further, since the component user is rarely the component developer, detailed instrumentation and analysis of the component by the user is not a credible option. Further, users are primarily expected to be interested in the coarse-grained performance of the component at the level of the public methods of the component. These two characteristics pose the requirements that PMM strategies (a) provide a coarse-grained performance model of the component and (2) be non-intrusive. The simplest approach, as verified in Section 2, is that of proxies, interposed between the caller and the called components, which intercept method calls and execute performance related tasks.

In this section we provide a brief summary of the CCA environment for HPC, adapt the approaches in Section 2 to HPC and address the issue of the minimal set of performance data required to construct component-level performance models.

3.1 The Common Component Architecture (CCA)

The CCA model uses the *provides-uses* design pattern. Components *provide* functionalities through interfaces that they export; they *use* other components' functionalities via interfaces. These interfaces are called *Ports*; thus a component has ProvidesPorts and UsesPorts. Components are peers and are independent. They are created and exist inside a framework; this is where they register themselves, declare their UsesPorts and ProvidesPorts and connect with other components.

CCAFFEINE [8] is the CCA framework we employ for our research. CCAFFEINE is a low latency framework for scientific computations. Components can be written in most languages within the framework; we develop most of our components in C++. All CCAFFEINE components are derived from a data-less abstract class with one deferred method called *setServices(Services *q)*. All components implement the *setServices* method which is invoked by the framework at component creation and is used by the components to register themselves and their UsesPorts and ProvidesPorts. Components also implement other data-less abstract classes, called Ports, to allow access to their standard functionalities. Every component is compiled into a shared object library that will be dynamically loaded at runtime.

A CCAFFEINE code can be assembled and run through a script or a Graphical User Interface (GUI). All components exist on the same processor and the same address space. Once components are instantiated and registered with the framework, the process of connecting ports is just the movement of (pointers to) interfaces from the *providing* to the *using* component. A method invocation on a UsesPort thus incurs a virtual function call overhead before the actual implemented method is used. CCAFFEINE uses the SCMD (Single Component Multiple Data) [8] model of parallel computation. Identical frameworks, containing the same components, are instantiated on all P processors. Parallelism is implemented by running the same component on all P processors and using MPI to communicate between them. P instances of a given component form a *cohort* [8] within which all message passing is done. The framework adheres to the MPI-1 standard ; dynamic process creation/deletion and a dynamically sized parallel virtual machine are not supported. This minimalist nature renders CCAFFEINE light, simple, fast, and very unobtrusive to the components. Performance is left to the component developer who is in the best position to determine the optimal algorithms and implementations for the problem at hand.

3.2 Performance Measurement and Modeling

A CCA application is composed of components and the composite performance of a component assembly is determined by the performance of the individual components as well as the efficiency of their interaction. Thus, the performance of a component has to be considered in a certain *context* consisting of the problem being solved (e.g., a component may have to do two functions, one which requires sequential access and the other strided access of an array), the parameters/arguments being passed to a method (e.g., length of an array) and the interaction between the caller and the callee (e.g., if a transformation of the data storage needs to be done). If multiple implementations of a component exist (i.e., implementations which provide the same functionality) then within a given context, there will be an optimal choice of implementation. This requires that performance models be available for all components and a means to generate a composite model exist.

Most scientific components intersperse compute intensive phases with message passing calls, which incur costs inversely proportional to the network speed. These calls sometimes involve global reductions and barriers, resulting in additional synchronization costs. For the purposes of this paper we will assume blocking communications where communications and computations are not overlapped. We will ignore disk I/O in this study. Thus, in order that a performance model for a component may be constructed, we require the following :

1. The total execution time spent in a method call. These methods are those in the ProvidesPorts of a component.

2. The total time spent in message passing calls, as determined by the total inclusive time spent in MPI during a method invocation.

3. The difference between the above is the time spent in computation, a quantity sensitive to the cache-hit rate. We will record this quantity for the period of the method call.

4. The input parameters that affect performance. These typically involve the size of the data being passed to the component and some measure of repetitive operations that might need to be done (e.g., the number of times a smoother may be applied in a multigrid solution).

The first three requirements are traditional and may be obtained from publicly available tools [18]. The fourth requires some knowledge of the algorithms being implemented, and is extracted by a proxy before the method invocation is forwarded to the component. We envisage that proxies will be simple and preferably, amenable to automatic generation.

4 PMM Software Infrastructure

As stated in Section 3, performance measurement will be done via proxies interposed between caller and callee components. These proxies are expected to be lightweight and serve as a means of intercepting and forwarding method calls. The actual functionality of interacting with and recording hardware characteristics will be kept in a separate component, as will the functionality of storing this data for each invocation. Our performance system consists of three distinct component types: a TAU (Tuning and Analysis Utilities) component, proxy components and a "Mastermind" component. These components work together in order to measure, compile and report the data back to the user.

4.1 TAU Component

In order to measure performance in a high performance scientific environment, a component that can interact with the system's hardware as well as time desired events is needed. For our performance measurement system, we use the TAU component[3], which utilizes the TAU measurement library[18, 19]. The TAU component is accessed via a MeasurementPort, which defines interfaces for timing, event management, timer control and measurement query. The timing interface provides a means to create, name, start, stop and group timers. It helps track performance data associated with a code region by bracketing it with start and stop calls.

The TAU implementation of this generic performance component interface supports both profiling and tracing measurement options. Profiling records aggregate inclusive and exclusive wall-clock time, process virtual time, hardware performance metrics such as data cache misses and floating point instructions executed, as well as a combination of multiple performance metrics. The event interface helps track application and runtime system level atomic events. For each event of a given name, the minimum, maximum, mean, standard deviation and number of entries are recorded. TAU relies on an external library such as PAPI [1] or PCL [2] to access low-level processor-specific hardware performance metrics and low latency timers. Timer control is achieved through the control interface, which can enable and disable timers of a given group at runtime. At runtime, a user can enable or disable all MPI timers via their group identifier. The query interface provides a means for the program to access a collection of performance metrics. In our performance system, the query interface is used to obtain the current values for the metrics being measured. The TAU library also dumps out summary profile files at program termination.

4.2 Proxies

For each component that the user wants to analyze, a proxy component is created. The proxy component shares the same interface as the actual component. When the application is composed and executed, the proxy is placed directly "in front" of the actual component. Since the proxy implements the same interface as the component, the proxy intercepts all of the method invocations for the component. In other words, the proxy uses and provides the same types of ports that the actual component provides. In this manner, the proxy is able to snoop the method invocation on the Provides Port, and then forward the method invocation to the component on the Uses Port. In addition, the proxy also uses a *Monitor* port to make measurements. If the method is one that the user wants to measure, monitoring is started before the method invocation is forwarded and stopped afterward. When the monitoring is started, parameters that influence the method's performance are sent to the Mastermind component. These parameters must be selected by someone with a knowledge of the algorithm implemented in the component. For example, for a routine that performs some simple processing on each index of an array of numbers, the performance parameter would most likely be the size of the array. Creating a proxy from a component's header file is relatively straight-forward. Currently, proxies are created manually with the help of a few scripts, but it is not difficult to envision proxy creation being fully automated.

4.3 Mastermind

The Mastermind component is responsible for gathering, storing and reporting of the measurement data. For each method that is monitored, a record object is created and stored by the Mastermind. The record object stores all the measurement data for each of the invocations of a single routine. When monitoring is started via a call to the Mastermind, the Mastermind passes the parameters to the record object and tells the record to begin timing. TAU measurements are made cumulatively, so in order to obtain the measurements for a single invocation, measurements must be made prior to the invocation and again after the invocation. To make a measurement, the TAU component is queried in order to record the current measurements for the timer, the MPI time, and any hardware metrics being measured. The MPI time is determined by the summation of the times of all the MPI routines. When monitoring is stopped, the TAU component is again queried to obtain the current time, MPI time and hardware measurements. The measurements for the single invocation are determined by the difference between the measurements obtained after the invocation and the measurements from before the invocation. The single invocation measurements, along with the parameters, are stored in the record. When a record object is destroyed, it outputs to a file all of the measurement data for each invocation that it stored.

5 Case Study

We use the infrastructure described in Section 4 to measure and model the performance of a component-based scientific simulation code. The code simulates the interaction of a shock wave with an interface between two gases. The scientific details are in [20]. The code employs Structured Adaptive Mesh Refinement [21, 22, 23] for solving a set of Partial Differential Equations (PDE) called the Euler equations. Briefly, the method consists of laying a relatively coarse Cartesian mesh over a rectangular domain. Based on some suitable metric, regions requiring further refinement are identified, the grid points flagged and collated into *rectangular* children patches on which a denser Cartesian mesh is imposed. The refinement factor between parent and child mesh is usually kept constant for a given problem. The process is done recursively, so that one ultimately obtains a hierarchy of patches with different grid densities, with the finest patches overlaying a small part of the domain. The more accurate solution from the finest meshes is periodically interpolated onto the coarser ones. Typically, patches on the coarsest level are processed first, followed recursively by their children patches. Children patches are also processed a set number of times during each recursion.

Figure 1 shows a snapshot from the simulation. The boxes outlined in purple are the coarse patches, those in red are ones which have been refined once (level 1 patches) and those in blue, refined twice (level 2 patches). The factor of refinement is 2 and the sequence of processing is $L_0, L_1, L_2, L_2, L_1, L_2, L_2$, where L_i is the set of patches on level i. Patches can be of any size or aspect ratio. This sequence is repeated multiple times.

Figure 2 shows the component version of the code. On the left is the **ShockDriver**, a component that orchestrates the simulation. On its right is **AMRMesh** that manages the patches. The **RK2** component below it orchestrates the recursive processing of patches. To its right are **States** and **EFMFlux** which are invoked on a patch-by-patch basis. The invocations to **States** and **EFMFlux** include a data array (a different one for each patch) and an output array of the same size. Both these components can function in two modes - sequential or strided array access to calculate X- or Y-derivatives respectively - with different performance consequences. Neither of these components involve message passing, most of which is done by **AMRMesh**. We will attempt to model the performance of both **States** and **EFMFlux** and analyze the message passing costs of **AMRMesh**. We will also analyze the performance of another component, **GodunovFlux**, which can be substituted for **EFMFlux**. Three proxies, one each for **States**, **GodunovFlux** and **EFMFlux** were created and interposed between **InviscidFlux** and the component in question. A proxy was also written for **AMRMesh** to capture message-passing costs. An instance each of **Mastermind** and **TAUMeasurement** component were created for performance measurement and recording.

The simulation was run on three processors of a cluster of dual 2.8 GHz Pentium Xeons with 512 kB caches. `gcc` version 3.2 was used for compiling with `-O2` optimization. Figure 3 shows where most of the time is spent in the component code. About 25% of the time is spent in `MPI_Waitsome()` which is invoked from two methods in **AMRMesh** - one that does "ghost-cell updates" on patches (gets data from abutting, but off-processor patches onto a patch) and the other that results in load-balancing and domain (re-) decomposition. The other methods, one in **States** and the other in **GodunovFlux** are modeled below.

In Figure 4 we plot the execution times for **States** for both the sequential and strided mode of operation. We see that for small, largely cache-resident arrays, both the modes take roughly the same time. As the arrays overflow the cache, the strided mode becomes more expensive and one sees a localization of timings. In Figure 5, we plot the ratio of strided and sequential access times. The ratio varies ranges from 1 for small arrays to around 4 for large ones. Further, for larger arrays, one observes large scatters. Similar phenomena are also observed for both **GodunovFlux** and **EFMFlux**.

During the execution of the application, both the X- and Y-derivatives are calculated and the two modes of operation of these components are invoked in an alternating fashion. Thus, for performance modeling purposes, we consider an average. However, we also include a standard deviation in our analysis to track the variability introduced by the cache. It is expected that both the mean and the standard deviation will be sensitive to the cache size. In Figures 6,7 and 8 we plot the execution times for the **States**, **GodunovFlux** and **EFMFlux** components. Regression analysis was used to fit simple polynomial and power laws, which are also plotted in the figures. The mean execution time scales linearly with the array size, once the cache effects have been averaged out. The standard deviations exhibit some variability, but they are significant only for **GodunovFlux**, a component that involves an internal iterative solution for every element of the data array. Note that these timings do not include the cost of the work done in the proxies, since all the extraction and recording of parameters is done outside the timers and counters that actually measure the performance of a component. Further, these instrumentation related overheads are small and will not be addressed in this paper.

If T_{States}, $T_{Godunov}$ and T_{EFM} are the execution times (in microseconds) for **States**, **GodunovFlux** and **EFMFlux** and Q the input array size, the best-fit expressions for the three components are

$$\begin{aligned} T_{States} &= \exp(1.19\log(Q) - 3.68) \\ T_{Godunov} &= -963 + 0.315Q \\ T_{EFM} &= -8.13 + 0.16Q \end{aligned} \tag{1}$$

The corresponding expressions for the standard deviations σ are

$$\begin{aligned} \sigma_{States} &= \exp(1.29\log Q) \\ \sigma_{Godunov} &= -526 + 0.152Q \\ \sigma_{EFM} &= 66.7 - 0.015Q + 9.24 \times 10^{-7}Q^2 - 1.12 \times 10^{-11}Q^3 + 3.85 \times 10^{-17}Q^4 \end{aligned} \tag{2}$$

We see that **GodunovFlux** is more expensive that **EFMFlux**, especially for large arrays. Further, the variability in timings for **GodunovFlux** increase with Q while it decreases for **EFMFlux**. While **GodunovFlux** is the preferred choice for scientists (it is more accurate), from a performance point of view, **EFMFlux** has better characteristics. This is an excellent example of a Quality of Service issue where numerical and/or algorithmic characteristics (such as accuracy, stability and robustness etc.) may need to be added to the performance model. Thus the performance of a component implementation would be viewed with respect to the size of the problem as well as the quality of the solution produced by it.

In Figure 9 we plot the communication time spent at different levels of the grid hierarchy during each communication ("ghost-cell update") step. We plot data for processor 0 first. During the course of the simulation, the application was load-balanced once, resulting in a different domain decomposition. This is seen in a clustering of message passing times at Level 0 and 2. Ideally, these clusters should have collapsed to a single point; the substantial scatter is caused by fluctuating network loads. Inset, we plot results for all the 3 processors. A similar scatter of data points is seen. Comparing with Figures 6, 7 and 8, we see that message passing times are generally comparable to the purely computational loads of **States** and **GodunovFlux**, and it is unlikely that the code, in the current configuration (the given problem and the level of accuracy desired) will scale well. This is also borne out by Figure 3 where almost a quarter of the time is shown to be spent in message passing.

6 Conclusions

We have proposed a software infrastructure for performance measurement in HPC component environments. Our prototypical implementation was used to collect performance data for a scientific simulation and construct performance models. While the data collected is no different from what is required in traditional HPC, the measurement system must be compatible with component software development methods and new strategies, such as proxies, must be adapted from other component-based environments. Proxies can be automatically generated from a component's header if the sole purpose is to time the execution of a component. However, for performance modeling, one frequently needs to record certain inputs to the component. Proxies are the logical place to extract this information before forwarding the component invocation, but this requires that this information be identifiable during proxy creation. We are currently investigating simple mark-up approaches identifying arguments/parameters which affect performance and need to be extracted and recorded.

The problem of *performance modeling* is still unsolved. The models derived here are valid only on a similar cluster. Any significant change, such as halving of the cache size, will have a large effect on the coefficients in the models (though the functional form is expected to remain unchanged). Ideally, the coefficients should be parameterized by processor speed and a cache model. We will address this in future work, where the cache information collected during these tests will be employed.

The ultimate aim of performance modeling is to be able to compose a composite performance model and optimize a component assembly. Apart from performance models, this requires multiple implementations of a functionality (so that one may have alternates to choose from) and a call trace from which the inter-component interaction may be derived. The wiring diagram (available from the framework) along with the call trace (detected and recorded by the performance infrastructure) can be used by the **Mastermind** to create a composite performance model where the variables are the individual performance models of the components themselves. Figure 10 shows a schematic of how such a system may construct an abstract dual (represented as a directed graph) of the application. Edge weights signify the number of invocations and the vertices are weighted by the compute and communication times, as predicted by the performance models of the component implementations. The caller-callee relationship is preserved to identify subgraphs that are insignificant from the performance point of view. This facilitates dynamic performance optimization which uses online performance monitoring to determine when performance expectations are not being met and new model-guided decisions of component use need to take place. This is currently underway.

References

[1] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/projects/papi/.

[2] PCL — The Performance Counter Library. http://www.fz-juelich.de/zam/PCL/.

[3] Sameer Shende, Allen D. Malony, Craig Rasmussen and Matt Sottile. A Performance Interface for Component-Based Applications. In *Proceedings of International Workshop on Performance Modeling, Evaluation and Optimization, International Parallel and Distributed Processing Symposium*, 2003.

[4] Darren J. Kerbyson, Henry J. Alme, Adolfy Hoisie, Fabrizio Petrini, Harvey J. Wasserman and Michael L. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, 2001. Distributed via CD-ROM.

[5] Darren J. Kerbyson, Harvey J Wasserman and Adolfy Hoisie. Exploring advanced architectures using performance prediction. In *International Workshop on Innovative Architectures*, pages 27–40. IEEE Computer Society Press, 2002.

[6] R. Englander and M. Loukides. *Developing Java Beans (Java Series).* O'Reilly and Associates, 1997. http://www.java.sun.com/products/javabeans.

[7] CORBA Component model webpage. http://www.omg.com. Accessed July 2002.

[8] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt and J. A. Kohl. The CCA core specifications in a distributed memory SPMD framework. *Concurrency: Practice and Experience*, 14:323–345, 2002. Also at http://www.cca-forum.org/ccafe03a/index.html.

[9] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott R. Kohn, Lois McInnes, Steve R. Parker and Brent A. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of High Performance Distributed Computing Symposium*, 1999.

[10] Sophia Lefantzi, Jaideep Ray and Habib N. Najm. Using the Common Component Architecture to Design High Performance Scientific Simulation Codes. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2003.

[11] Sameer Shende, Allen D. Malony and Robert Ansell-Bell. Instrumentation and measurement strategies for flexible and portable empirical performance evaluation. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA '2001*, pages 1150–1156. CSREA, June 2001.

[12] Jim Maloney. *Distributed COM Application Development Using Visual C++ 6.0.* Prentice Hall PTR, 1999. ISBN 0130848743.

[13] Adrian Mos and John Murphy. Performance Monitoring of Java Component-oriented Distributed Applications. In *IEEE 9th International Conference on Software, Telecommunications and Computer Networks - SoftCOM*, 2001.

[14] Baskar Sridharan, Balakrishnan Dasarathy and Aditaya Mathur. On Building Non-Intrusive Performance Instrumentation Blocks for CORBA-based Distributed Systems. In *4th IEEE International Computer Performance and Depenability Symposium*, March 2000.

[15] Baskar Sridharan, Sambhrama Mundkur and Aditaya Mathur. Non-intrusive Testing, Monitoring and Control of Distributed CORBA Objects. In *TOOLS Europe 2000*, June 2000.

[16] Nathalie Furmento, Anthony Mayer, Stepen McGough, Steven Newhouse, Tony Field and John Darlington. Optimisation of Component-based Applications within a Grid Environment. In *Proceedings of Supercomputing*, 2001. Distributed via CD-ROM.

[17] Nathalie Furmento, Anthony Mayer, Stepen McGough, Steven Newhouse, Tony Field and John Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28:1753–1772, 2002.

[18] TAU: Tuning and Analysis Utilities. http://www.cs.uoregon.edu/research/paracomp/tau/.

[19] Allen D. Malony and Sameer Shende. *Distributed and Parallel Systems: From Concepts to Applications*, chapter Performance Technology for Complex Parallel and Distributed Sys tems, pages 37–46. Kluwer, Norwell, MA, 2000.

[20] R. Samtaney and N.J. Zabusky. Circulation deposition on shock-accelerated planar and curved density stratified interfaces : Models and scaling laws. *J. Fluid Mech.*, 269:45–85, 1994.

[21] M. J. Berger and J. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *J. Comp. Phys.*, 53:484–523, 1984.

[22] M. J. Berger and P. Collela. Local adaptive mesh refinement for shock hydrodynamics. *J. Comp. Phys.*, 82:64–84, 1989.

[23] James J. Quirk. A parallel adaptive grid algorithm for shock hydrodynamics. *Applied Numerical Mathematics*, 20, 1996.

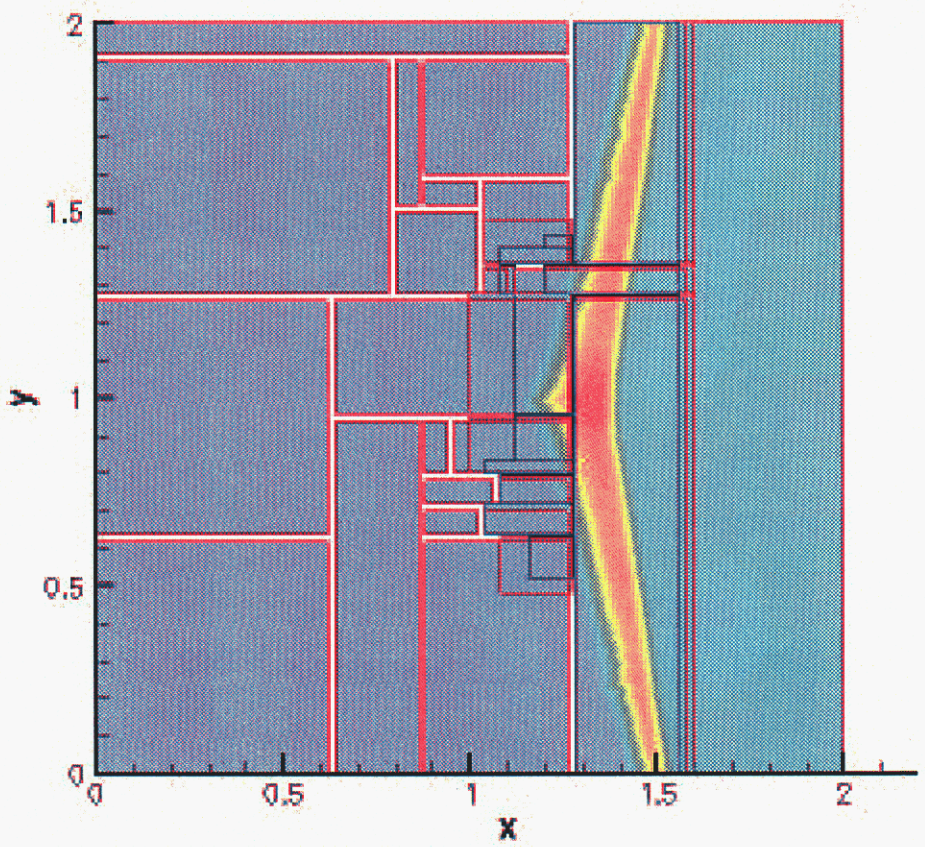

Figure 1: The density field plotted for a Mach 1.5 shock interacting with an interface between Air and Freon. The simulation was run on a 3-level grid hierarchy. Purple patches are the coarsest (Level 0), red ones are on Level 1 (refined once by a factor of 2) and blue ones are twice refined.

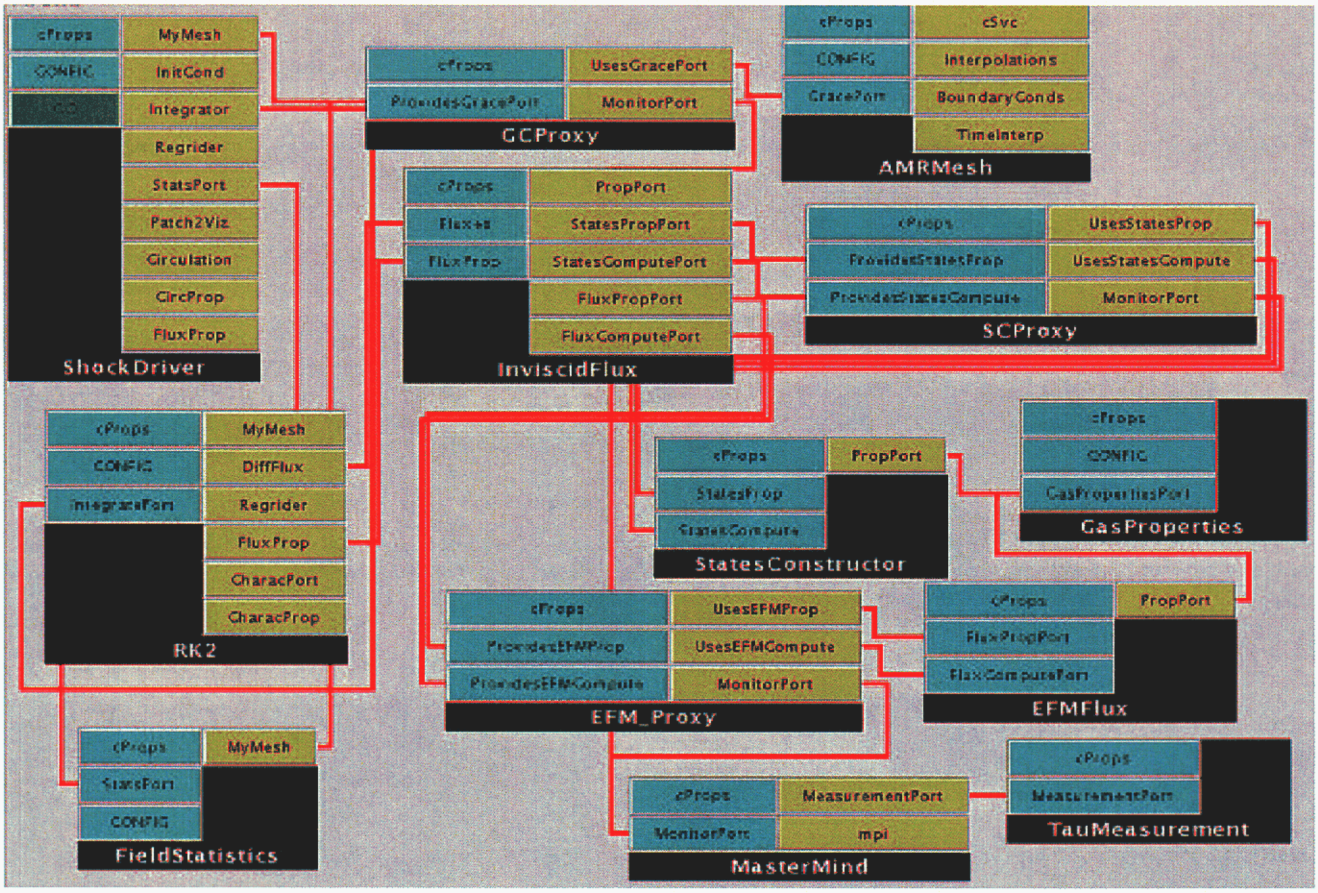

Figure 2: Snapshot of the component application, as assembled for execution. We see three proxies (for **AMRMesh, EFMFlux** and **States**), as well as the **TauMeasurement** and **Mastermind** components to measure and record performance-related data.

```
FUNCTION SUMMARY (mean):
---------------------------------------------------------------------------
%Time Exclusive  Inclusive  #Call Inclusive  Name
          msec   total msec       usec/call
---------------------------------------------------------------------------
100.0   55,244  1:52.032       1  112032939  int main(int, char **)
 24.3   27,262    27,262   12.75    2138235  MPI_Waitsome()
 12.0   13,482    13,482    1632       8261  g_proxy::compute()
 10.9   12,240    12,240    1632       7501  sc_proxy::compute()
  1.0    1,077     1,077  7029.5        153  icc_proxy::prolong()
  0.8      895       895     186       4813  icc_proxy::restrict()
  0.7      768       768   20959         37  TAU_GET_FUNCTION_VALUES()
  0.6      662       662       1     662412  MPI_Init()
  0.2      168       168    3.25      51753  MPI_Comm_dup()
  0.1      145       145    0.25     581244  MPI_Finalize()
  0.1       68        68    4.75      14358  MPI_Allreduce()
  0.0        8         8    3.25       2655  MPI_Allgather()
  0.0        3         4    6.75        594  MPI_Isend()
  0.0        2         2    3.25        913  MPI_Barrier()
  0.0     0.92     0.925       1        926  MPI_Comm_create()
  0.0    0.166     0.166   37.75          4  MPI_Irecv()
  0.0    0.158     0.158    39.5          4  MPI_Wtime()
  0.0    0.149     0.149    6.25         24  MPI_Wait()
  0.0   0.0428    0.0428       1         43  MPI_Keyval_create()
  0.0   0.0408    0.0408      16          3  MPI_Errhandler_set()
  0.0   0.0267    0.0267    6.25          4  MPI_Cancel()
```

Figure 3: Snapshot from a timing profile done with our infrastructure. We see that around 50% of the time is accounted for by `g_proxy::compute()`, `sc_proxy::compute()` and `MPI_Waitsome()`. The MPI call is invoked from **AMRMesh**. The two other methods are modeled as a part of the work reported here. Timings have been averaged over all the processors. The profile shows the inclusive time (total time spent in the methods and all subsequent method calls), exclusive time (time spent in the specific method less the time spent in subsequent *instrumented* methods), the number of times the method was invoked, and the average time per call to the method, irrespective of the data being passed into the method.

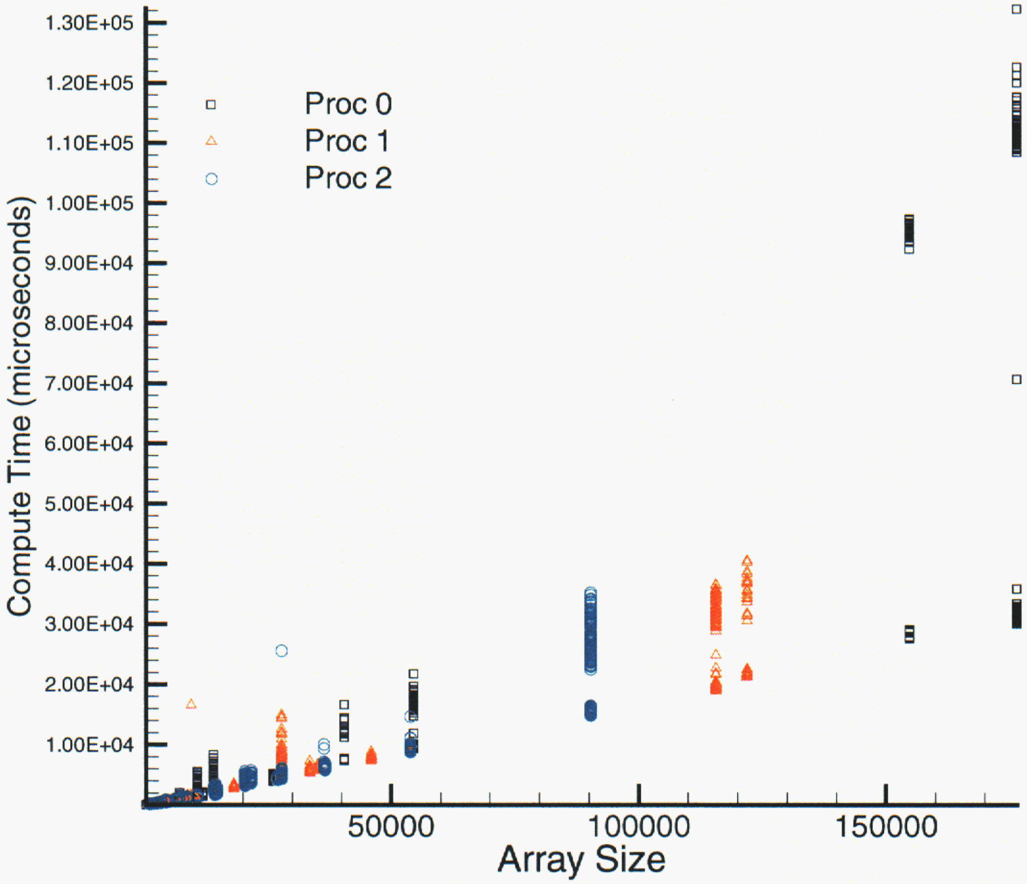

Figure 4: Execution time for the **States** component. The **States** component is invoked in two modes, one which requires sequential and the other which requires strided access of arrays to calculate X- and Y- derivatives of a field. Both the times are plotted. The Y-derivative calculation (strided access) is expected to take longer for large arrays and this is seen in the spread of timings. For small array sizes, which are largely cache-resident, the two different modes of access do not result in a large difference in execution time. Array sizes are the actual number of elements in the array. The elements are double precision numbers. The different colors represent data from different processors (Proc i in the legend) and similar trends are seen on all processors.

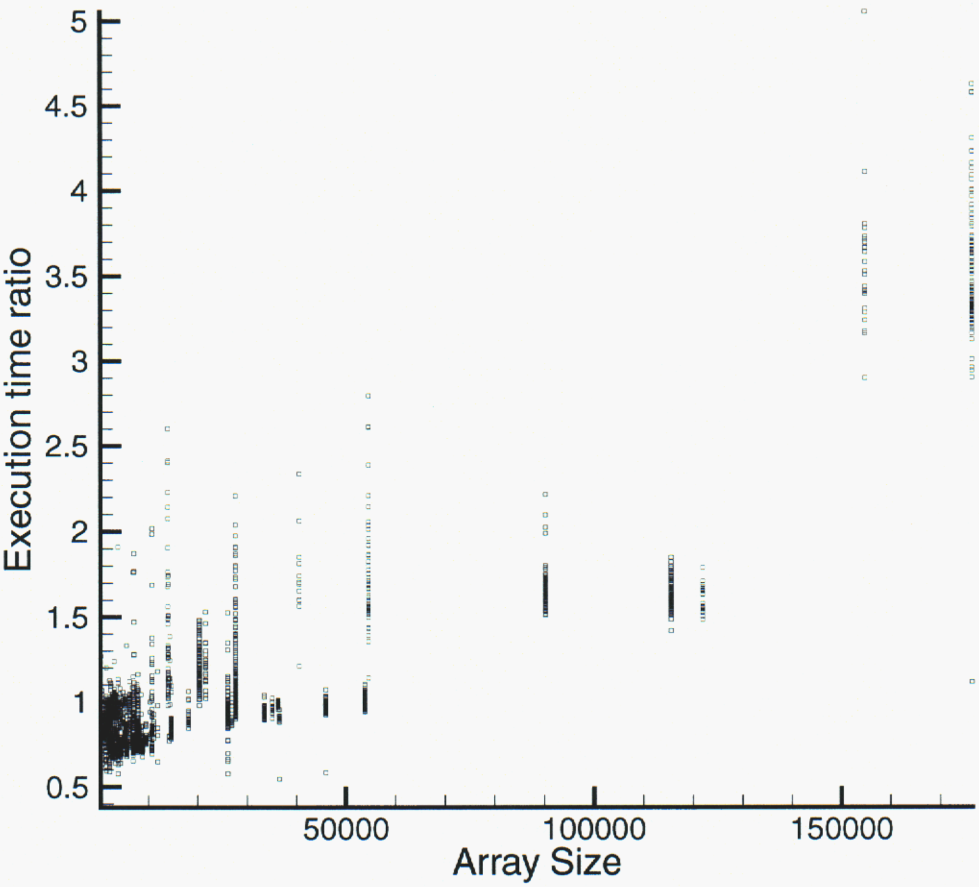

Figure 5: Ratio of strided versus sequential access (calculation of Y- and X-derivatives, respectively) timings for **States**. We see that the ratio varies from around 1 for small array sizes to around 4 for the largest arrays considered here. Array sizes are the actual number of elements in the array. The elements are double precision numbers. Further, the ratios show variability which tend to increase with array size

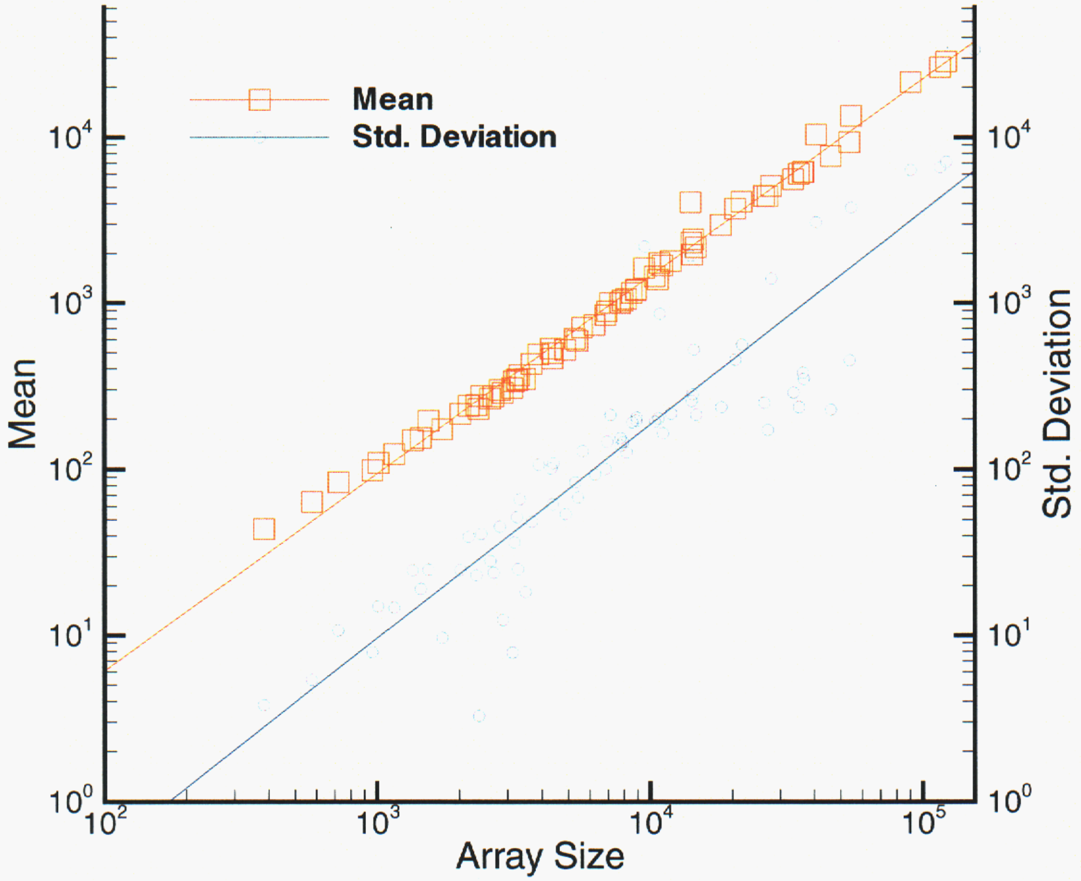

Figure 6: Average execution time for **States** as a function of the array size. Since **States** has a dual mode of operation (sequential versus strided) and the mean includes both, the standard deviation of is rather large. The performance model is given in Eq. 1. The standard deviation, in blue, is plotted against the right Y-axis. All timings are in microseconds.

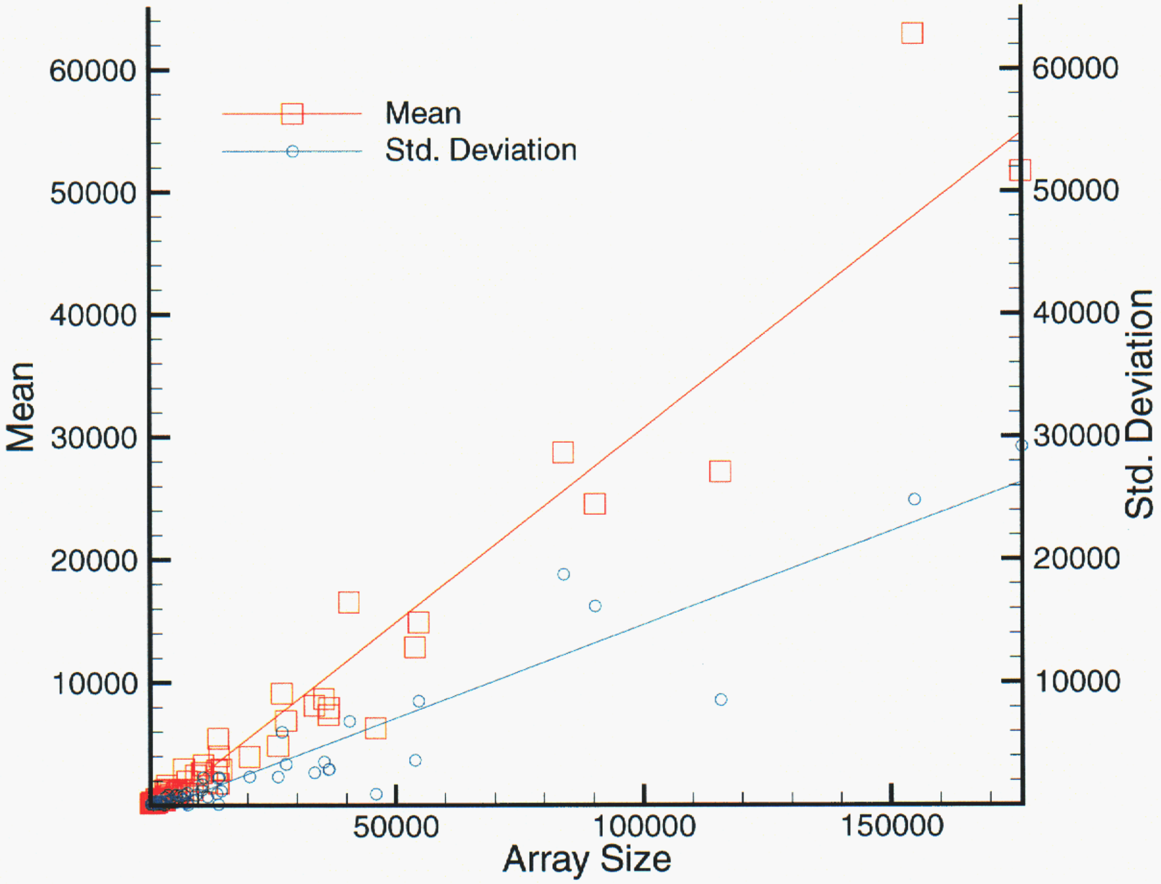

Figure 7: Average execution time for **GodunovFlux** as a function of the array size. Since **GodunovFlux** has a dual mode of operation (sequential versus strided) and the mean includes both, the standard deviation of is rather large. The performance model is given in Eq. 1. The standard deviation, in blue, is plotted against the right Y-axis. All timings are in microseconds.

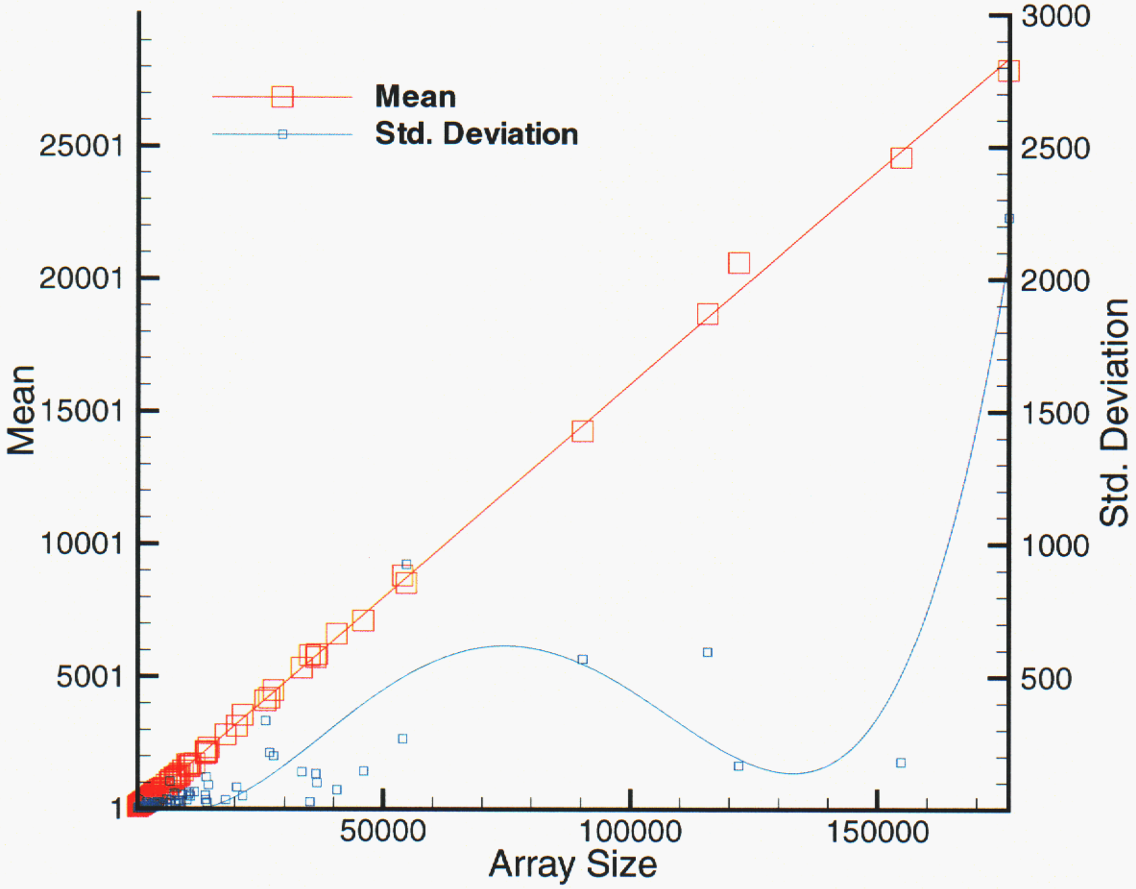

Figure 8: Average execution time for **EFMFlux** as a function of the array size. Since **EFMFlux** has a dual mode of operation (sequential versus strided) and the mean includes both, the standard deviation of is rather large. The performance model is given in Eq. 1. The standard deviation, in blue, is plotted against the right Y-axis. All timings are in microseconds.

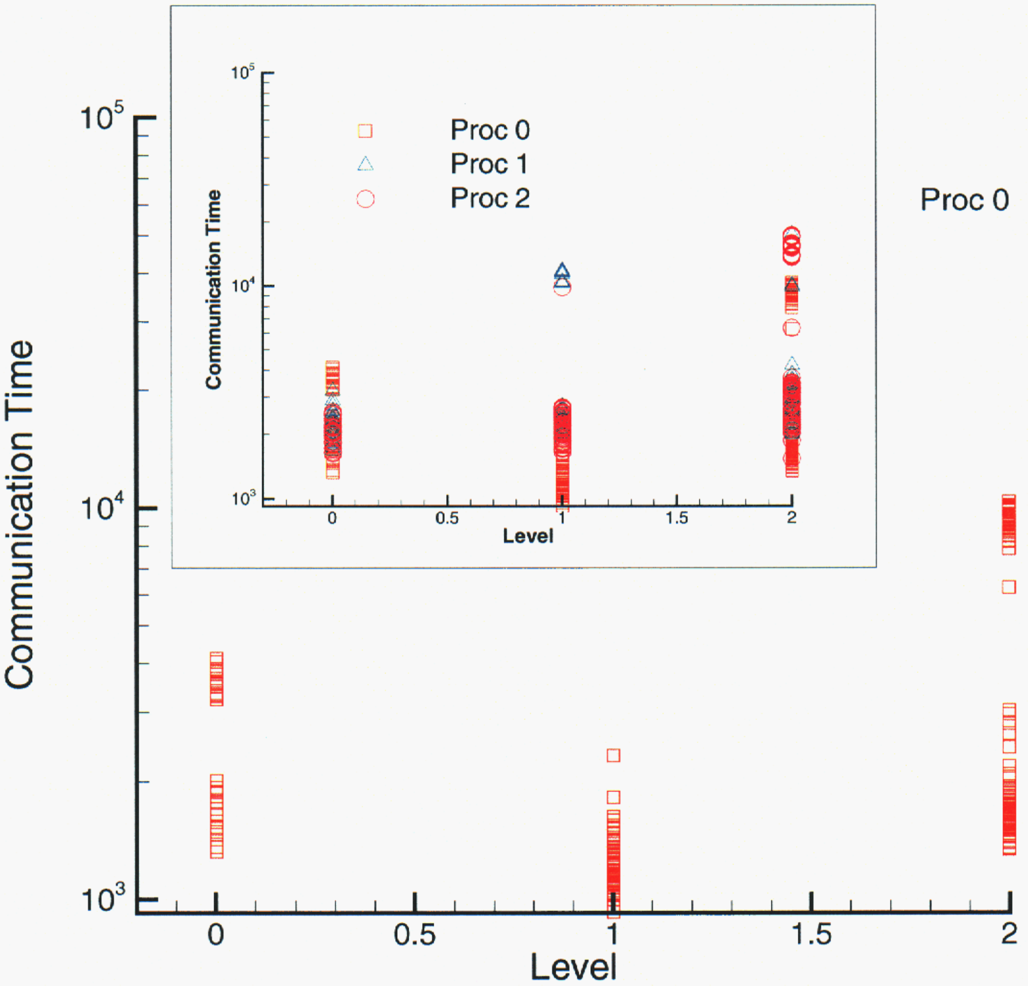

Figure 9: Message passing time for different levels of the grid hierarchy for the 3 processors. We see a clustering of message passing times, especially for Levels 0 and 2. The grid hierarchy was subjected to a re-grid step during the simulation which resulted in a different domain decomposition and consequently message passing times. Inset : We plot the timings for all processors. Similar clustering is observed. All times are in microseconds.

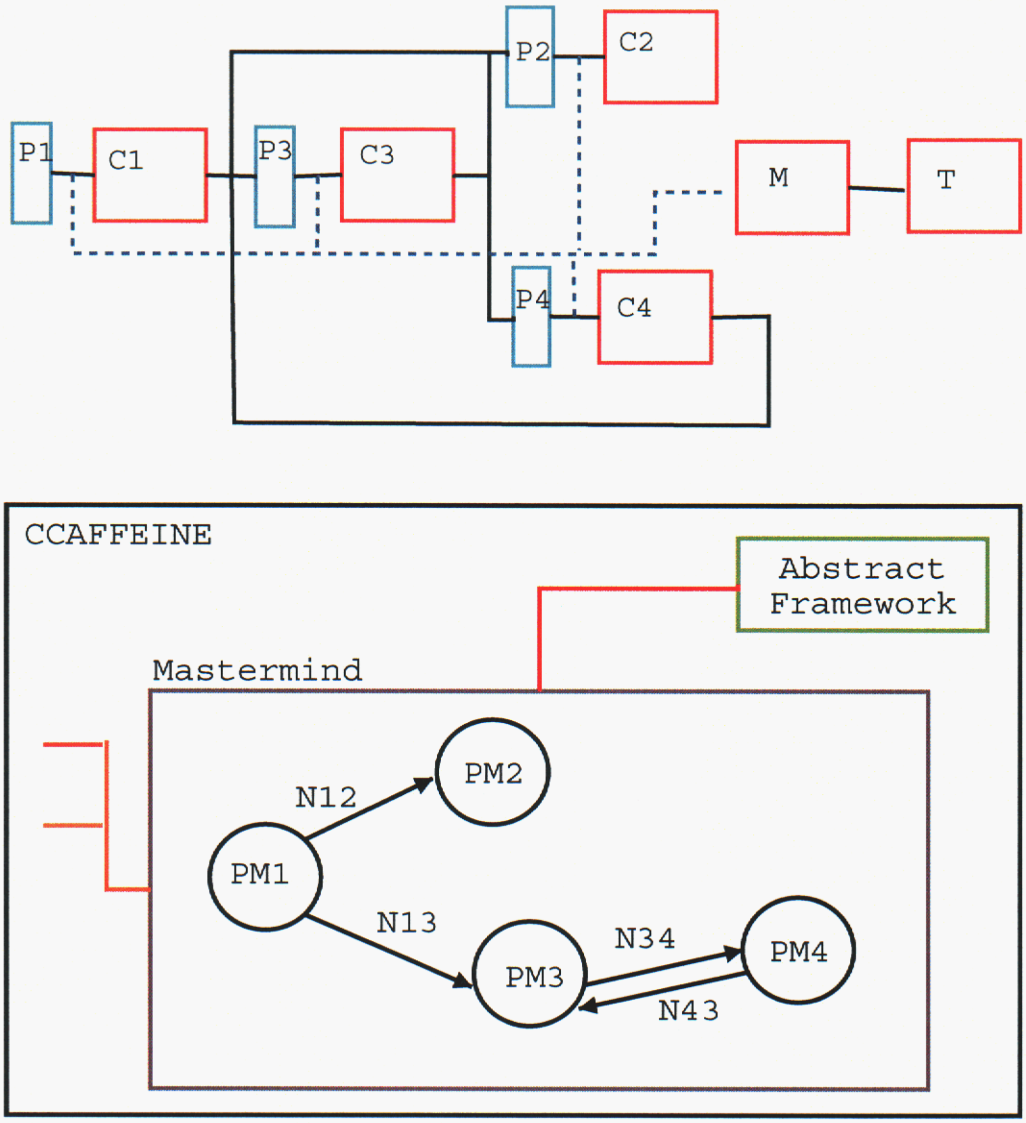

Figure 10: Above: A simple application composed of 4 components. C denotes a component, P denotes a proxy and M and T denote an instance of **Mastermind** and **TauMeasurement** components. The black lines denote port connection between components and the blue dashed lines are the proxy-to-**Mastermind** port connections which are only used for PMM. Below, it dual, constructed as a directed graph in the **Mastermind**, with edge weights corresponding to the number of invocations and the vertex weights being the compute and communication times determined from the performance models (PM_i) for component i. Only the port connections shown in black in the picture above are represented in the graph. The parent-child relationship is preserved to identify sub-graphs that do not contribute much to the execution time and thus can be neglected during component assembly optimization. The **Mastermind** is seen connected to CCAFFEINE via the **AbstractFramework** Port to enable dynamic replacement of sub-optimal components.

Distribution List

External Distribution

1 Dr. S. Shende, Department of Computer and Information Science,
1202 University of Oregon, Eugene, OR 97403.

1 Prof. A. Malony, Room 307, Department of Computer and Information Science,
1202 University of Oregon, Eugene, OR 97403

Internal Distribution

1	MS 9915	Dr. R. C. Armstrong
2	MS 9915	Mr. N. Trebon
2	MS 9051	Dr. J. Ray
3	MS 9018	Central Technical Files, 8945-1
1	MS 0899	Technical Library, 9616
1	MS 9021	Classification Office, 8511 for Technical Library, MS 0899, 9616 DOE OSTI via URL