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

J. Ray, N. Trebon, R. C. Armstrong
Sandia National Laboratories,
Livermore, CA 94551
{jairay,ndtrebo,rob}@sandia.gov

S. Shende and A. Malony
Dept. of Computer Science,
University of Oregon, Eugene, OR 97403
{sameer,malony}@cs.uoregon.edu

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. Component-based HPC applications allow the possibility of creating component-level performance models and synthesizing them into application performance models. However, they impose the restriction that performance measurement/monitoring needs to be done in a non-intrusive manner and at a fairly coarse-grained level. We 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 three components in a scientific application and construct their performance models. Both computational and message-passing performance are addressed. [1]

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. The effective processing speed 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 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. 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[2, 3]. In a parallel environment these tools track and report on the size, frequency, source, destination and the time spent in passing messages between processors [19]. 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 [11, 12].

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 [9] and CORBA [1] are largely unsuitable for HPC [5] and a new light-weight model, called the Common Component Architecture (CCA) [6], 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 [13].

CCA component-based applications are composed out of standalone components at runtime, 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 perfor-

[1] Presented at the *18th International Parallel and Distributed Processing Symposium*, Santa Fe, NM, USA, April $26^{\text{th}} - 30^{\text{th}}$, 2004.

mance model to judge the optimality of a component assembly. Further component-based software is seldom used exclusively by the authors of the components themselves and manual instrumentation of the code is impossible. Exhaustive automatic instrumentation of an executable where a binary is rewritten or instrumented at runtime [18] too has little meaning. Consequently, a non-intrusive strategy where each component is monitored to collect execution time, the hardware characteristics and relevant inputs (like the size of arrays) that affect the collected performance data is clearly indicated. These data then need to be synthesized into individual component performance models. Also, since the containing CCA framework creates, configures and assembles components, it possesses the global understanding of how the components are networked into an application. This information, coupled with the individual components' performance models, can be used to synthesize a predictive performance model of the entire application. In this work, we will attempt to create a small set of components to assemble a non-intrusive performance measurement and modeling infrastructure. This infrastructure will then be used to monitor an existing CCA-component based scientific simulation code (assembly of CCA components) and construct performance models for the components. It is assumed the none of the scientific components will be modified to assist in performance measurement nor do they have any special features (e.g. performance-related interfaces) that allow or enable the collection of performance-related data. While the material presented in this work is far from realizing the goal of synthesizing an application-level performance model and using it in a predictive capacity, 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.

2 Related Work

The three most widely-used component standards (CORBA [1], COM/DCOM [14], Java Beans [9]) 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 [6]. Thus, performance metrics developed for these environments are inadequate for HPC. 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 Multi-processors) 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 [16]. 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 [20, 21] 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 [10], 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

As indicated in Section 1, performance measurement and modeling (PMM), in a component environment, will assist

in optimizing the component assembly. It is required that PMM strategies (a) provide a coarse-grained performance model of the component and (b) be non-intrusive. The simplest approach, as reviewed 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 [5] 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) [5] 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. 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 synthesize these into a model for the application 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 [4]. 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[19], which utilizes the TAU measurement library[4, 15]. 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 [2] or PCL [3] 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 (provided by the Mastermind component; see below) 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 by the proxy 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 and effort is under way to automate it.

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. To make a measurement, the TAU component is queried at the start and the end of each method invocation to turn on/off timing and hardware counters, prior to recording the measurements. The MPI time is determined by the summation of

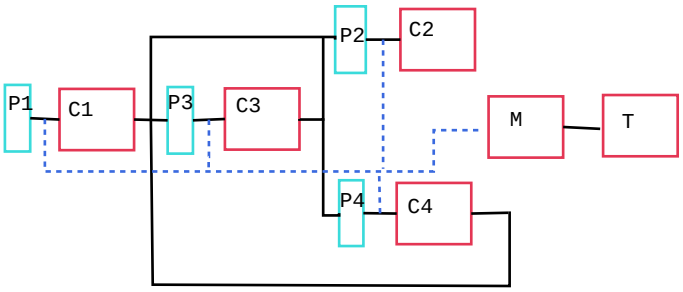

Figure 1. A simple application composed of 4 components. C denotes a component, P denotes a proxy and M and T denote an instance of Mastermind and TauMeasurement components. The black lines denote port connection between components and the dashed lines are the proxy-to-Mastermind port connections. The proxies, along with the Mastermind, TauMeasurement and the dashed port connections form the non-intrusive PMM infrastructure "around" the original component assembly.

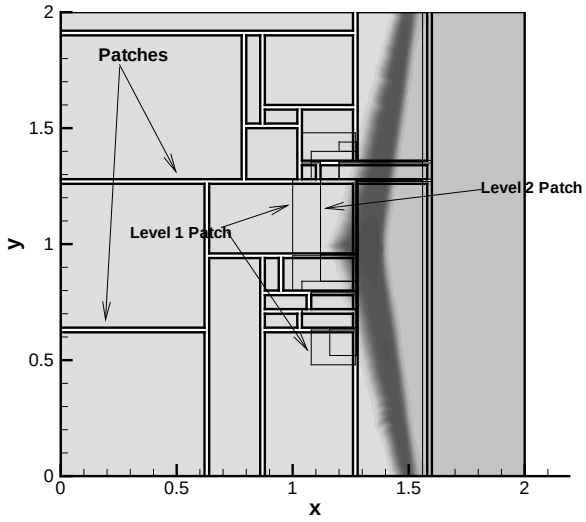

Figure 2. The density field plotted for a Mach 1.5 shock interacting with an interface between Air and Freon. The simulation was run on a 3-level grid hierarchy. Patches outlined with the thickest lines are the coarsest (Level 0), those outlined with medium lines are on Level 1 (refined once by a factor of 2) and the ones outlined with the finest lines are twice refined (Level 2).

the times of all the MPI routines. 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.

Thus we envisage a non-intrusive performance measurement and modeling component infrastructure built around an existing scientific simulation component assembly. Figure 1 shows a component assembly of four components with a PMM infrastructure of proxies, **Mastermind** and **TAU** built around it. The infrastructure is non-intrusive and modification of the scientific components was not required.

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 [17]. The code employs Structured Adaptive Mesh Refinement [8, 7] 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* (also called "boxes") 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 2 shows a snapshot from the simulation. Level 0 boxes are the coarse mesh patches, Level 1 patches are those which have been refined once and Level 2 patches have been refined twice. 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 8 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 re-

cursive processing of patches. To its right are **StatesConstructor** and **EFMFlux** which are invoked on a patch-by-patch basis. The invocations to **StatesConstructor** and **EFMFlux** include a data array (a different one for each patch) and an output array of the same size. Neither of these components involve message passing, most of which is done by **AMRMesh**. We will attempt to model the performance of both **StatesConstructor** 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 **StatesConstructor**, **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. Table 1 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 **StatesConstructor** and the other in **EFMFlux** are modeled below.

In Figure 3 we plot the execution times for each invocation of **StatesConstructor**, as collected during a simulation. The **StatesConstructor** 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 respectively. Sequential and strided access of a given array occurs in an interleaved manner, as dictated by the numerical algorithm. Both the times (per invocation) are plotted. We see that for a given array per-invocation execution time is not constant, especially for large arrays. The Y-derivative calculation (strided access) is expected to take longer for large arrays and this is seen in the spread of timings. On the other hand, 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 around two foci. The ratio of strided and sequential access times varies from ≈ 1 for small arrays to ≈ 4 for large ones. Further, for larger arrays, one observes large scatters. Similar phenomena are also observed for both **GodunovFlux** and **EFMFlux** (not shown here).

Since the **StatesConstructor** is invoked to calculate the X- and Y-derivatives an equal number of times, we will consider the average of the sequential and strided array processing times for modeling. 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 4 and 5 we plot the execution times for the **StatesConstructor** 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. Similar trends are seen the the analysis for **GodunovFlux**. (not shown here). 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

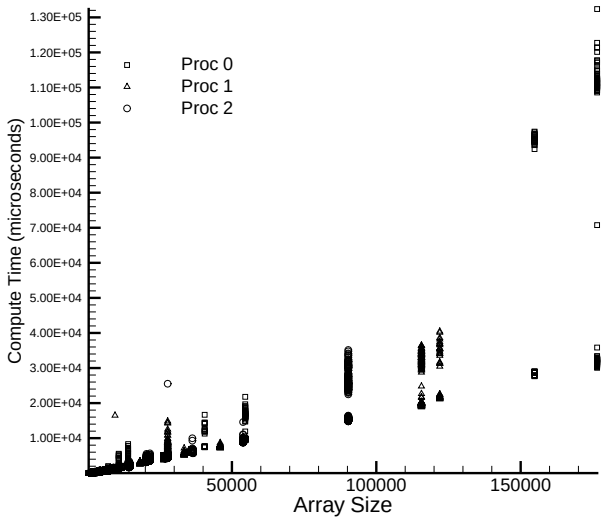

Figure 3. Raw per invocation execution time for the StatesConstructor component, collected during a simulation, plotted as a function of the data array size. For a given array size, the larger timings correspond to the calculation of Y-derivatives on the field since it involves strided access of non-cache-resident arrays. Array sizes are the actual number of double precision numbers in the array. The different symbols (□, ∘ and △) represent data from different processors (Proc i in the legend) and similar trends are seen on all processors.

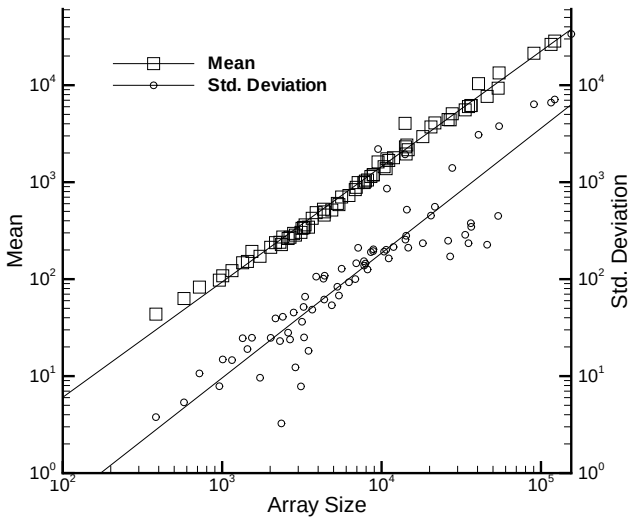

Figure 4. Average execution time (□) for StatesConstructor as a function of the array size. Since StatesConstructor has a dual mode of operation (sequential versus strided) and the mean includes both, the standard deviation of is rather large. The performance model is given in Eq. 1. The standard deviation (○) is plotted against the right Y-axis. All timings are in microseconds. Solid lines are best-fits to the data.

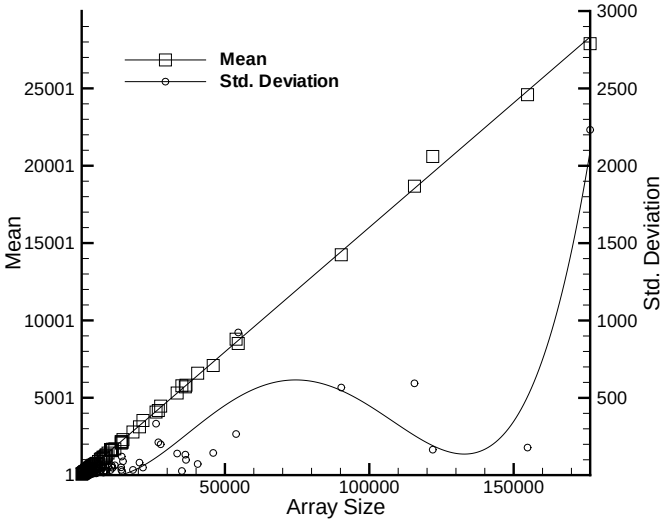

Figure 5. Average execution time (□) for EFMFlux as a function of the array size. Since EFMFlux has a dual mode of operation (sequential versus strided) and the mean includes both, the standard deviation of is rather large. The performance model is given in Eq. 1. The standard deviation (○) is plotted against the right Y-axis. All timings are in microseconds. Solid lines are best-fits to the data.

addressed in this paper.

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

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

The corresponding expressions for the standard deviations σ are

$$\begin{aligned} \sigma_{StatesConstructor} &= \exp(1.29\log Q) \\ \sigma_{Godunov} &= -526 + 0.152Q \\ \sigma_{EFM} &= 66.7 - 0.015Q + 9.24\times10^{-7}Q^2 \\ &\quad -1.12\times10^{-11}Q^3 \\ &\quad +3.85\times10^{-17}Q^4 \end{aligned} \quad (2)$$

We see that **GodunovFlux** is more expensive that **EFMFlux**, especially for large arrays. Further, the variability in timings (i.e., standard deviation) for **GodunovFlux** increase with Q while its behavior is more complex 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 6 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

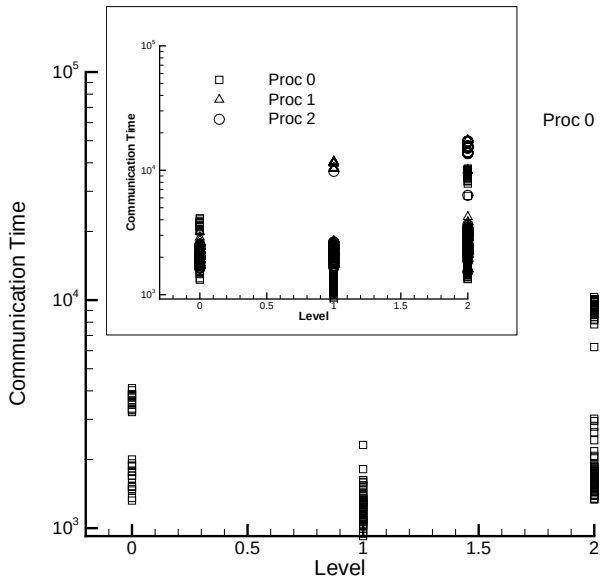

Figure 6. Per invocation message passing time for different levels of the grid hierarchy for the 3 processors. We see a clustering of message passing times, especially for Levels 0 and 2. The grid hierarchy was subjected to a re-grid step during the simulation which resulted in a different domain decomposition and consequently the clustering in message passing times. Inset : We plot the timings for all processors. Similar clustering is observed. All times are in microseconds. Symbols □, ∘ and △ indicate data from processors 0, 1 and 2.

for all the 3 processors. A similar scatter of data points is seen. Comparing with Figures 4 and 5, we see that message passing times are generally comparable to the purely computational loads of **StatesConstructor** and **EFMFlux**, 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 Table 1 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 7 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.

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,

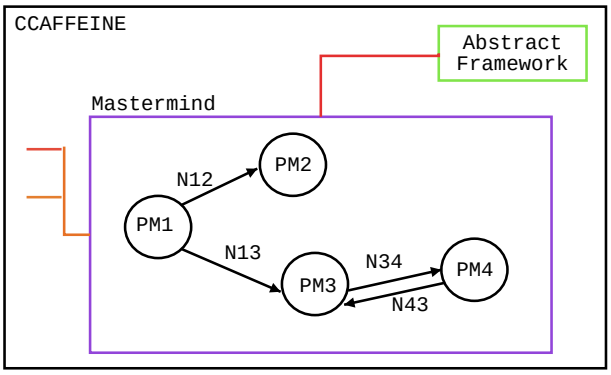

Figure 7. Dual of Figure 1, constructed as a directed graph in the Mastermind, with edge weights corresponding to the number of invocations and the vertex weights being the compute and communication times determined from the performance models (PM_i) for component i. Only the port connections shown in black in Figure 1 are represented in the graph. The Mastermind is seen connected to CCAFFEINE via the AbstractFramework Port to enable dynamic replacement of sub-optimal components.

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 .

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.

References

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

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

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

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

[5] 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.

[6] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. R. Kohn, L. McInnes, S. R. Parker, and B. A. Smolinski. Toward a Common Component Architecture for High-Performance Scientific Computing. In *Proceedings of High Performance Distributed Computing Symposium*, 1999.

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

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

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

[10] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28:1753–1772, 2002.

[11] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. L. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, 2001. Distributed via CD-ROM.

[12] D. J. Kerbyson, H. J. Wasserman, and A. Hoisie. Exploring advanced architectures using performance prediction. In *International Workshop on Innovative Architectures*, pages 27–40. IEEE Computer Society Press, 2002.

[13] S. Lefantzi, J. Ray, and H. N. Najm. Using the Common Component Architecture to Design High Performance Scientific Simulation Codes. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2003.

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

[15] A. D. Malony and S. 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.

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

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

[18] S. Shende, A. D. Malony, and R. 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.

[19] S. Shende, A. D. Malony, C. Rasmussen, and M. 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.

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

[21] B. Sridharan, S. Mundkur, and A. Mathur. Non-intrusive Testing, Monitoring and Control of Distributed CORBA Objects. In *TOOLS Europe 2000*, June 2000.

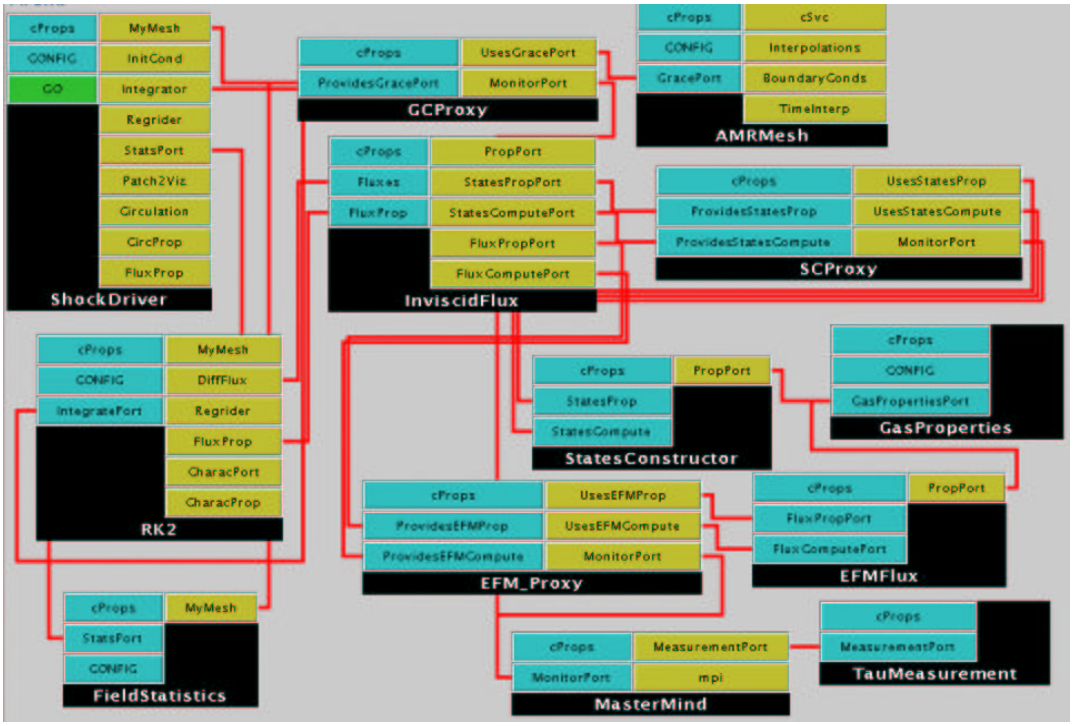

Figure 8. Snapshot of the component application, as assembled for execution. We see three proxies (for AMRMesh, EFMFlux and States), as well as the TauMeasurement and Mastermind components to measure and record performance-related data. Data arrays are allocated in AMRMesh and passed around among components via pointers.

% Time	Exclusive msec	Inclusive total msec	# Call	Inclusive usec/call	Name
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()

Table 1. Top 10 entries from a timing profile done with our infrastructure. 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. % time is calculated from a running sum of the inclusive times.