Performance Analysis Support for Object Oriented Parallel Scientific Applications [*][†]

Jeffrey Nesheiwat and Boleslaw K. Szymanski

Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY 12180-3590, USA
E-mail: {neshj, szymansk}@cs.rpi.edu

December 2, 1998

[*]This work was partially supported by NSF Grant CCR-9527151. The content does not necessarily reflect the position or policy of the U.S. Government.

[†]The supercomputers used in this study were an SGI/CrayT3E at NASA/Goddard Space Flight Center provided by the HPCC Earth and Space Sciences Project and a Beowulf system provided by High Performance Computing and Applications Group at the Jet Propulsion Laboratory / NASA.

Contents

1 Introduction and Overview

The complex and computationally demanding nature of scientific applications has fueled research in the area of parallel computing. Moving from conventional uniprocessor systems to multiprocessor systems makes designing, developing, testing, tuning, and maintaining scientific codes much more difficult. These difficulties are outweighed by the significant speedup that parallel computing can provide.

Since the primary reason for writing parallel codes is speed [13], it comes as no surprise that performance analysis is a vital part of the development process. Analysis tries to determine if a given algorithm is as fast as it can be, where the program can be further optimized, and how efficiently the underlying system is being used. Raj Jain [15] explains that analysis, for both sequential and parallel systems, can be done in one of three ways:

- **Analytic Modeling**. This involves using models of the executing program and its underlying architecture to derive performance information. While this technique can yield data quickly, the accuracy of this data is subject to the number of initial assumptions and complexity of the models used.
- **Simulation**. This affords more accuracy than analytic modeling. In this approach the target system's response to the executing program is simulated. This allows less assumptions to be made about the execution environment. The drawback is that simulation quickly becomes too time consuming and in some cases not feasible as the system becomes sufficiently complex.
- **Measurement**. This approach, the focus of the research described in this report, involves instrumenting an executing application. This is the most time consuming of the analysis techniques, but since measurements are taken on the target system it is clearly the most accurate[14].

After a brief overview of the basic structure of analysis tools, we present a high level description of the scalable instrumentation and program database technique for collecting, storing, and filtering performance data. We validate this new approach by showing preliminary results using two finite element codes, one sequential and one parallel application. We conclude with a discussion of how this approach can be extended to include parallel object oriented applications.

1.1 Motivation of Research

The goal of this research is to explore new techniques in which both sequential and parallel scientific applications can be analyzed for purposes of optimization and performance tuning. This involves exploiting technology from other disciplines, specifically databases, to collect, store, and analyze collected performance data. To be useful, it must afford the programmer the flexibility to conduct customized performance experiments and be powerful enough to provide answers to key performance related questions. In addition, such techniques must be able to support analysis of next generation object oriented parallel codes.

1.1.1 Analysis Techniques

Typically when analyzing the performance of an application, the program is run to determine if further analysis is necessary. The next step involves profiling the code at a high level, specifically to determine which modules spend the most time executing. These performance critical modules are analyzed to localize bottlenecks to specific program constructs. It is up to the programmer to determine the cause of these bottlenecks and to remove them. This top-down approach to performance analysis prevents the programmer from spending time optimizing areas that are not on the critical path [2] [13]. Analysis of this nature is carried out mainly for two reasons:

- **Optimization.**
- **Comparative Analysis.**

In addition to ensuring that a code is as fast as it can be, it is important to be able to determine how effectively the code is able to make use of the target architecture. In many cases, a code will perform significantly better on one system than another given where the strengths of that system lie and the demands on resources placed by the executing program. This is of particular importance when evaluating different architectures for specific applications [4] [7] [12].

Adaptive parallel finite element codes represent a class of applications that could benefit significantly from the analysis approach we propose. Typically these codes impose significant computational demand, hence the need for parallelism. Analysis is difficult from an experiment management perspective, these codes typically have scores of input and tuning parameters that drastically affect performance as a function of the underlying system and the problem being solved (irregularity). These issues are explicitly addressed by our *scalable instrumentation and program database* approach.

1.1.2 Structure of Analysis Tools

Performance analysis tools are comprised of three basic components, *instrumentation*, *visualization*, and *support for analysis.* [14] Instrumentation refers to steps taken to collect performance related information from executing programs. Visualization involves processing this data such that program behavior can be viewed graphically. Support for analysis is assigning causes and prescribing solutions to performance problems exposed in the instrumentation and visualization phases.

Instrumentation includes counters showing how frequently a module is invoked, timers showing how much time is spent executing, or timers measuring communication and synchronization delays. There are four metrics on which the quality of instrumentation should be judged:

- **Probe effect**
- **Granularity of data**
- **Mapping to source code**
- **Cost of invocation**

All instrumentation is subject to the *Probe Effect*[29] to varying degrees. That is, introducing instrumentation will alter the behavior of the program being analyzed. Instrumentation not only changes the execution time of the program, but asynchronous events that were governed by the program are now governed by the program and its accompanying instrumentation [19]. The probe effect goes beyond tainting timing and altering synchronization, it can drastically alter memory access patterns thus affecting cache utilization. Minimizing probe effect is a significant issue in the area of performance analysis research.

The granularity of instrumentation data range from individual statements to a holistic view of the state of the machine. This leads to the issue of *data scalability.* The amount of instrumentation data collected is proportional to the execution time of the program and to the number of processors used. In some cases, tools such as *AIMS* [14] collect and write instrumentation data to disk at a rate of megabytes per second. Overwhelming system resources in this way will taint measured performance because the program has to wait for resources, in this case the disk, when it normally would not. The same applies to other resources such as memory, or CPU. Preserving data scalability is a guiding principle in our research.

Performance data showing the state of the underlying system is not sufficient for localizing bottlenecks and analyzing an executing program. This information needs to be mapped to specific source code constructs. In this way, the program's effect on the system can be understood in terms of specific loops, communication, and synchronization events. We present a novel approach in this paper that addresses the issue of mapping instrumentation to source code features.

If the cost of adding instrumentation to an application is too high, it is likely that it will not be used. In the ideal case, instrumentation should be a parameter that is turned on or off and involves no additional effort on the user's part aside from customization. Performance tools that require instrumentation to be added manually become impractical for large applications.

Instrumentation can be introduced at various levels:

- **Hardware:** Hardware can be used to count the number of mathematical operations, monitor network utilization, time critical events, etc.
- **Kernel:** Special facilities can be provided by the kernel and operating system to monitor system call usage and probe the state of the machine using timed interrupts.
- **Binary:** Some tools, e.g., *Paradyn* [23], modify the binary image of an executing program to collect timing and frequency information for critical areas of the program. This technique involves overwriting a portion of the binary with a jump to an instrumentation module that will start/stop a timer or increment a counter.
- **Compiler:** Instrumentation can be added without modifying the original program text by having the compiler collect information about the program structure and linking in profiling libraries that extend the functionality of standard libraries.
- **Source:** When other means are not available or appropriate, adding code to the program is the easiest way to collect performance data. This can be as simple as adding timers and counters to measure specific regions of the code where bottlenecks are likely to exist.

There are benefits and trade-offs to instrumenting at each of these levels. Hardware based instrumentation provides the most accurate and fine grain data with a minimal probe effect. The drawback is that it is difficult to map this data to specific regions of the executing program. Many supercomputers provide specialized instrumentation hardware however, this hardware is specific to each system and does not have a standard interface. The advantage is that there is little or no cost for invoking hardware based instrumentation. All the other methods provide varying degrees of granularity and invasiveness. Kernel, or operating system, based instrumentation includes sampling environments where the machine's state is periodically probed. Like hardware instrumentation, this provides fine grain data with little or no mapping to the executing program and little invocation cost. Instrumenting at the binary level is dynamic, in that it can be added and removed during execution. This technique is applicable to long running programs. It is more costly to invoke in that the run-time environment must support dynamic instrumentation. Coupling data collection with the executing program makes it easier to map data to regions of the program. Compiler and source code level instrumentation offer variable levels of granularity and traditionally are most invasive. The benefit is that performance data can be easily mapped to the executing program. The cost of invoking compiler level instrumentation is minimal in that it involves linking in instrumentation modules. Instrumenting source code is more time consuming but allows the user to control the granularity and enables easy mapping of collected data to source code.

In recent years, the use of object oriented languages for parallel programming has increased, exacerbating the need for new instrumentation techniques [25]. Instrumentation centers around the program's control flow graph. Object oriented programs require a higher level of abstraction based on coupling control-flows with object connectivity. We will describe how the scalable instrumentation and program database approach we propose can be extended to provide support for object oriented parallel codes.

Visualization. A great deal of effort has gone into performance visualization. There are a large number of tools on many platforms that provide visual displays of performance data [10] [11] [9] [27]. Similar to data scalability, there is the notion of *visual scalability.* Most visualizations work well for programs that run for a short time on a small number of processors. As these quantities increase, the visual display becomes less useful and more overwhelming to the user.

Support for Analysis involves more than drawing conclusions. It involves breaking down, filtering and organizing available information, thus enabling users to draw multiple conclusions [32]. It is here that our instrumentation database approach is most useful. A framework for understanding a program's performance can be formed based on instrumentation data collected in a scalable way. This information in conjunction with static information about the architecture, program, and inputs can be used to derive an integrated view of the program's performance across multiple runs, input vectors, and architectures.

1.2 Research Issues

These tools are representative of the state of the art in performance analysis for parallel applications. They address the following issues in different ways:

- **Source code mapping.** Mapping performance data to specific program constructs is vital for optimization. Performance tools simply probe the state of the machine while the executing program affects the machine's state. In this way there is no easy way to reconcile performance bottlenecks with specific code constructs. (Figure-1a).

- **Customizable visualization front-end.** Demand on system resources varies greatly from program to program. Moreover, resource availability is tightly coupled with the underlying system architecture. This variability between program demand on resources, and system resources mandates that visualization tools support multiple views emphasizing both program structure and machine status views of the executing program.

- **Comparative Analysis.** When determining which architecture is most appropriate for a given program, it is important to collect and analyze visualization data in a platform independent way. In this way, analysts can gauge how well an application will migrate to a new system. Most tools do not support this type of analysis, and the few that do only provide support for selected architectures. The notion of comparative analysis can be extended to the more general notion of experiment management. In comparative analysis, system architecture is simply a parameter that we permute. Experiment management provides a framework for managing data when many such parameters are permuted. These parameters include input vectors and even program modules.

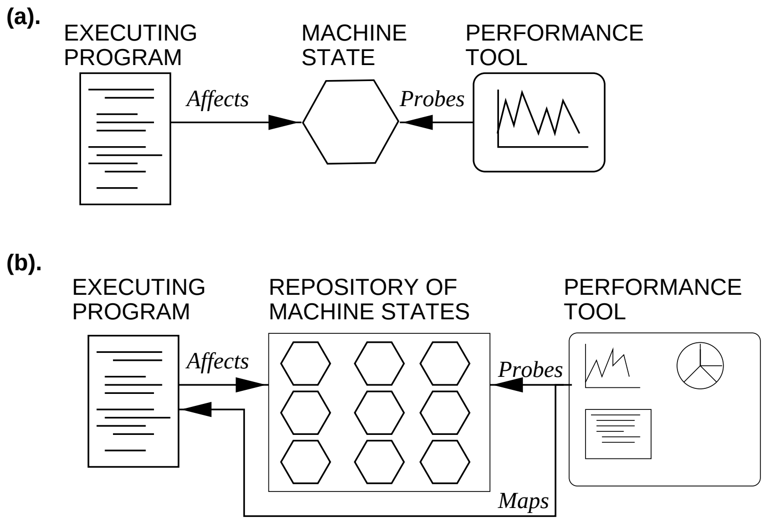


Figure 1: (a). Illustrates traditional performance tools that probe the state of the machine while executing applications affect the machine. (b). Illustrates an environment where the program's affect on machine state is archived and mapped back to specific program constructs. This information is visualized in different ways.

- **Probe Effect and Dilation.** Introducing instrumentation involves some degree of intrusion on the performance being measured. Instrumentation and accompanying intrusion compensation take time to compute and contribute significant overhead to executing programs. Ideally instrumentation should provide a minimum of invasiveness without imposing a significant dilation in "real" running time. This way programs can be evaluated on large realistic input vectors.
- **Easy to use and extensible.** In order for a tool to be useful, it must be used. It is vital that the cost of invoking the tool and adding instrumentation is minimal. It should also be possible to extend functionality to enable analysis of diverse programs and architectures.

These items represent research problems that need to be addressed. Resolving these issues will involve a multidisciplinary approach [14] [31]. Areas such as user interfaces, data visualization, compilers, artificial intelligence, automated testing, experiment management, and databases can contribute significantly.

2 Research Objectives and Approach

The focus of this work is to explore issues and research problems central to performance analysis of parallel scientific applications. Specifically support for comparative analysis through experiment management, fine grained scalable instrumentation that is mappable to source code, enabling technologies for customizable visualization / analysis, accuracy, usability and investigation of techniques for support of object oriented programs.

After considering these issues, it follows that performance data should be uncoupled from the underlying architecture and associated with the control flow graph of the executing program. The resulting

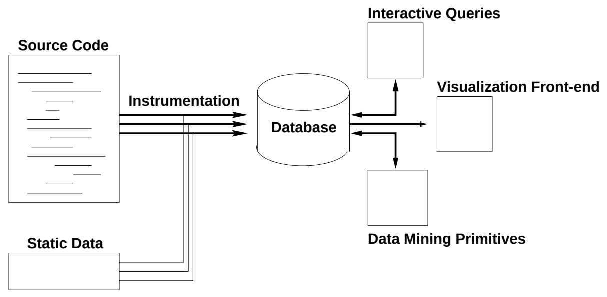


Figure 2: Instrumentation database architecture.

data structures are too complex to be captured using trace files. We propose exploiting existing database technology by mapping program structure and fixed size statistical data onto formal database schema. This novel use of an instrumentation database facilitates comparative analysis by providing a framework for experiment management and enables source code mapping since performance data is cast in terms of program structure and not underlying architecture. Scalability is maintained by aggregating statistical data during data collection. Database queries provide a powerful interface for front-end visualization and analysis tools.

This proposed instrumentation database (Figure-2) framework archives collected performance data for a given program. As the program is run repeatedly with different parameters, the database can be used to derive conclusions about overall performance (Figure-1b). Hierarchical instrumentation and static data captured by the database make it possible to map performance to source code and run-time environment.

2.1 Scalable Instrumentation

Amount of performance data is a function of the number of processors used and trace file size is a function of the running time of the application. This limitation requires that programs be analyzed using smaller input vectors or for smaller durations of time. There is a need for scalable data collection where size is a function of program structure. There is also a need to map this data back to executing source code. These issues are addressed by tightly coupling instrumentation data with the program's control flow graph (CFG). A CFG based view of the program ensures that data scalability can be achieved, assuming that data collected for each node in the CFG is of fixed size. Moreover, CFG nodes are easily mapped to specific source code constructs. This mapping is one of the main contributions of the approach we propose.

2.1.1 Control Flow Hierarchies

There are four events that impact performance significantly. These are: Procedures, Loops, Procedure Calls, and Communications [19]. Instead of considering the entire CFG, we look at the subset consisting of only these performance critical events. The resulting subgraph is the *Control Flow Hierarchy*, CFH for short. Each node in the CFH maps these performance critical events to collected performance data. A *probe* is introduced in the code at each of these nodes to collect aggregated timing and statistical data.

To ensure data scalability, individual data points are not collected. Instead running statistics are continually refined when a probe is encountered. When the statistics are updated, the data point is

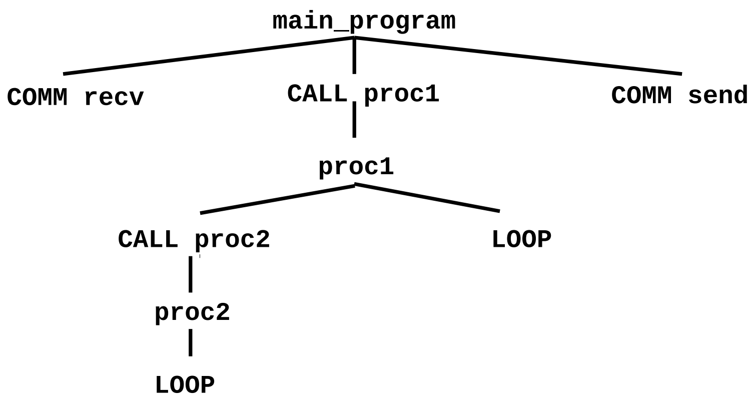


Figure 3: Sample control flow hierarchy.

OPERATION	PARAMETERS	DESCRIPTION
INIT	IDB-FILE, CFG-FILE	Initializes data structures, reads configuration file (CFG-FILE). Prepares database (IDB-FILE) for writing.
CLOSE	NONE	Flushes data structures to database and deallocates storage
START	PROBE-ID, PROBE-TYPE	Allocates data structures for probe if needed. Starts probe timers, increments counter and begins collecting noise reduction data.
STOP	PROBE-ID	Stops timer, updates noise reduction data

Table 1: Instrumentation API

discarded ensuring that each probe's information is of fixed size. The CFH's size is strictly bound by the program structure, hence data scalability is ensured. Instrumentation data is collected with calls to an instrumentation application programmer's interface (API), as shown in Table-1.

Specialized performance data can be derived, or inferred, from a minimal set of statistical data collected at run time. Table-2 shows what is collected by probes introduced at each node of the CFH.

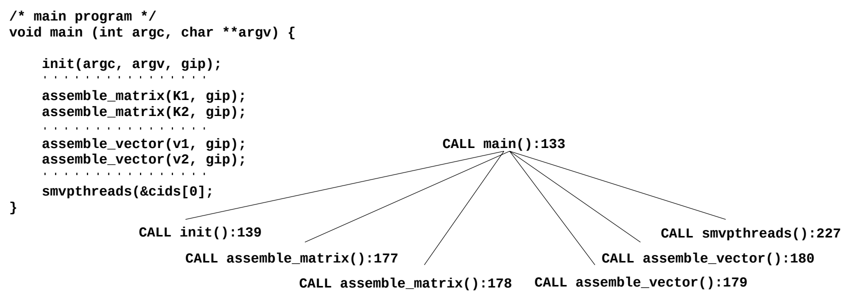
A database is used to capture these control flow hierarchies and their accompanying statistical data for each run of the program. Figures 3, 4, 5, and 6 show example control flow hierarchies.

2.1.2 Noise Reduction Techniques

Performance tools strive to collect accurate data with as little intrusion to the executing program as possible. Various noise reduction techniques are used to ensure that the collected data is accurate. Instrumented programs suffer from a *dilation effect* causing the program to take longer to execute because of instrumentation overhead. Dilation and noise are reduced by factoring out instrumentation overhead,

DATA	DESCRIPTION
AVG	Average time spent executing event
MIN	Shortest time spent executing event
MAX	Longest time spent executing event
TIME	Time spent on current execution of event
SDEV	Standard deviation of measured times
COUNT/ITER	Number of times event was executed

Table 2: Statistics collected by each probe for a given PROC, CALL, LOOP, or COMM event.



```
/* main program */
void main (int argc, char **argv) {

    init(argc, argv, gip);
    ................
    assemble_matrix(K1, gip);
    assemble_matrix(K2, gip);
    ................
    assemble_vector(v1, gip);
    assemble_vector(v2, gip);
    ................
    smvpthreads(&cids[0];
}
```

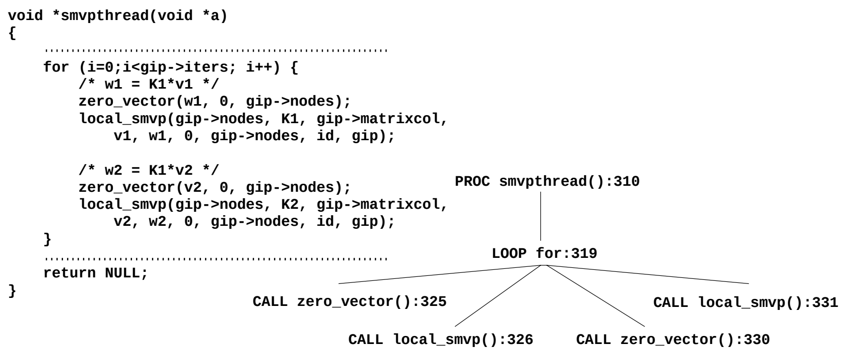
CALL main():133
CALL init():139
CALL assemble_matrix():177
CALL assemble_matrix():178
CALL assemble_vector():179
CALL assemble_vector():180
CALL smvpthreads():227

Figure 4: A fragment of the Spark98 `main` program and its accompanying control flow hierarchy.

efficiently implementing the instrumentation API, and selectively instrumenting critical portions of the executing program [22].

Each probe in the CFH has noise associated with it. This noise is a function of two parameters; the nesting level of probes from the root node in the CFH, and the number of times that these probes are activated. Each time a probe is encountered data is collected and stored in the CFH. Thus overhead is incurred on every PROC, CALL, LOOP, and COMM event. We employ two techniques to minimize and factor out this noise:

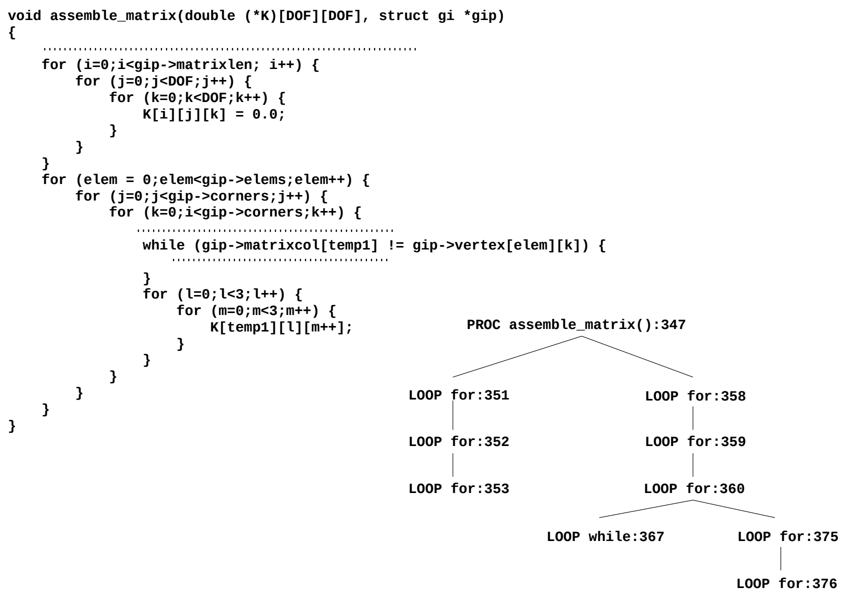
- **Factorization**. Each probe measures how long it takes to complete its own instrumentation



```
void *smvpthread(void *a)
{
    ................................................................
    for (i=0;i<gip->iters; i++) {
        /* w1 = K1*v1 */
        zero_vector(w1, 0, gip->nodes);
        local_smvp(gip->nodes, K1, gip->matrixcol,
            v1, w1, 0, gip->nodes, id, gip);

        /* w2 = K1*v2 */
        zero_vector(v2, 0, gip->nodes);
        local_smvp(gip->nodes, K2, gip->matrixcol,
            v2, w2, 0, gip->nodes, id, gip);
    }
    ................................................................
    return NULL;
}
```

Figure 5: A fragment of the `smvpthread` module and its accompanying control flow hierarchy.



```
void assemble_matrix(double (*K)[DOF][DOF], struct gi *gip)
{
    ..........................................................................
    for (i=0;i<gip->matrixlen; i++) {
        for (j=0;j<DOF;j++) {
            for (k=0;k<DOF;k++) {
                K[i][j][k] = 0.0;
            }
        }
    }
    for (elem = 0;elem<gip->elems;elem++) {
        for (j=0;j<gip->corners;j++) {
            for (k=0;i<gip->corners;k++) {
                ....................................................
                while (gip->matrixcol[temp1] != gip->vertex[elem][k]) {
                    ........................................
                }
                for (l=0;l<3;l++) {
                    for (m=0;m<3;m++) {
                        K[temp1][l][m++];
                    }
                }
            }
        }
    }
}
```

Figure 6: A fragment of the `assemble_matrix` module and its accompanying control flow hierarchy.

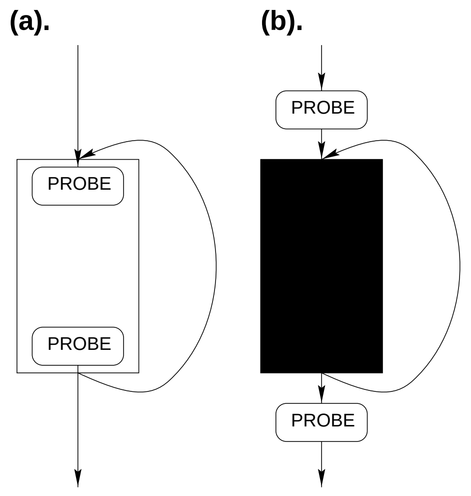


Figure 7: (a). Whitebox loop instrumentation collects instrumentation data for each iteration of the loop. (b). Blackbox loop instrumentation collects instrumentation data once, treating the loop as a single event.

activities. This information is stored locally by each probe in the CFH as its cumulative overhead contribution. Noise is factored out by summing these values for all nested probes. For example, to factor out total noise for the program, we sum these values for every node in the CFH and subtract the resulting value from the total time stored at the root node.

- **Selective instrumentation**. The best way to eliminate instrumentation overhead is to avoid instrumenting at all. There are many regions of a program that do not need to be instrumented

Function	Running Time
local_smvp()	50.37s
assemble_matrix()	23.21s
zero_vector()	00.69s

Table 3: Running times of selected Spark98 functions measured using `gprof`.

because they are provably optimal or not in the critical execution path. In other words, they do not impact performance significantly. Instrumentation can be selectively inserted in areas that are of interest. Figure-7 illustrates blackening out loops. Trivial loops are the most common source of noise. Blackening them out avoids unnecessary probing thus reducing noise and dilation.

These basic noise reduction techniques significantly improve accuracy and minimize dilation effects.

2.2 Database Approach

The term database in this context refers to a *Database Management System* (DBMS). Elmasri and Navathe define a DBMS as "a collection of programs that enables users to create and maintain a database ... that facilitates the process of defining, constructing, and manipulating databases for various applications."[5].

2.2.1 Derived Attributes and Comparative Analysis

Given the information in Table-3, simple queries can be issued to the instrumentation database to ascertain which function has the longest running time.

```
Q1 =  SELECT   procedure
      FROM     runs(Spark98, Nodes=1,
               Arch='Solaris_25', mesh='sf5.1.pack')
      WHERE    run_time = MAX(procedure.run_time)
```

This query returns the database tuples containing probe data corresponding to the control flow hierarchy rooted at the `local_smvp` node, the function with the longest running time. This is the first step in a top down analysis.

```
Q2 =  SELECT   Event
      FROM     Q1
      WHERE    run_time = MAX(event.run_time)
```

The next step is to find the bottleneck within this function. Q2 returns the node in `local_smvp`'s CFH with the largest running time.

This trivial example does not show the full analytic capabilities of our proposed approach. It does shows how a relational *view* is formed using the `run()` portion of the FROM part in Q1. It also shows how metrics that we do not explicitly collect can be derived, or inferred, from collected data. In this

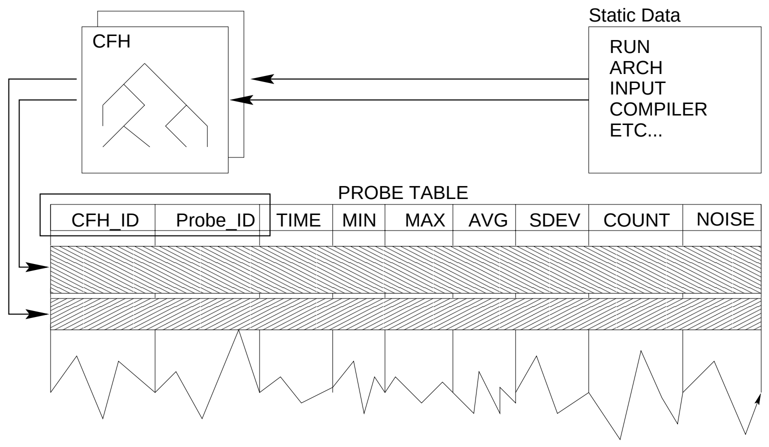


Figure 8: Static Data is associated with numerous Control Flow Hierarchies, one for each run of the program. A CFH is associated with the Probe Table. The CFH provides parent and child information for each probe. The dotted box around the first two entries in the Probe Table signify that the CFH_ID and the PROBE_ID together act as the primary key for indexing probes. This means that no two entries in the database will have the same values for these two attributes.

example, the search for the function with the longest run time is restricted to sequential runs of the Spark98 kernel [28] for a specific architecture and mesh. SQL syntax can be extended to provide easier interface to database contents. Analysis queries are dependent on the schema that defines how data is stored in the database.

2.2.2 Schema Design

The data to be stored in the instrumentation database falls into one of three categories:

1. **Static Data** associates program execution with static information such as: architecture, input vector, compiler, etc. This information provides the basis for experiment management.
2. **Probe Data** contains statistical information about each probe, namely MIN, MAX, AVG, SDEV, TIME, COUNT, NOISE, etc. Each probe's data is of fixed size.
3. **CFH Data** defines how probes for a given run are related to each other. This hierarchy resembles the control flow graph of the program. It differs in that the only nodes represented correspond to CALL, PROC, LOOP, and COMM events. The size of this data is bound by the structure of the program.

A relational database is ideal for storing tabular information [5] such as the *Static Data* or the *Probe Table* as shown in figure-8. Relational databases are not adequate for storing graph based information, such as the CFH, in such a way that it can be efficiently queried. Object-oriented databases are particularly well suited to storing and querying graph data but are not efficient for querying of statistical data. An object relation database will be used to capture the control flow hierarchy and traditional tables will be used to store statistical and static information. In this way, queries can be cast in terms of how probes are interconnected across multiple control flow hierarchies, this provides a powerful framework that enables experiment management.

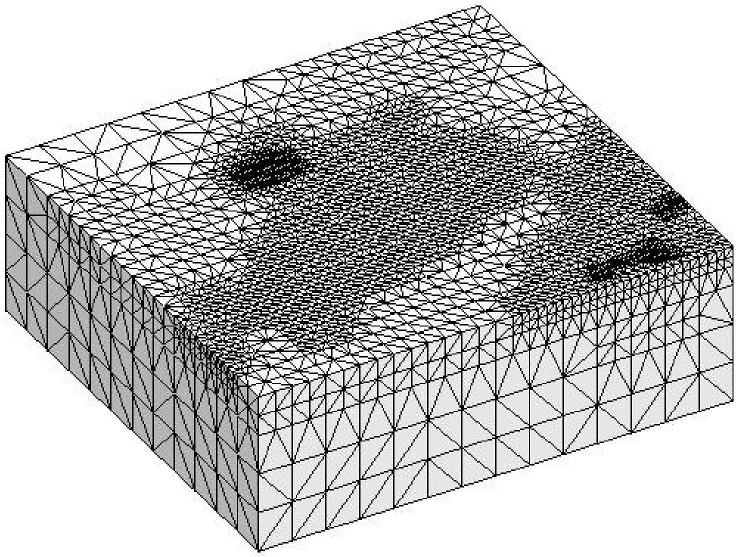


Figure 9: Spark98 input mesh modeling ground movement during an earthquake.

3 Current Work and Preliminary Results

The scalable instrumentation API has been implemented and ported to a number of platforms. This is an important first step towards validating this approach. Collected data is currently stored using flat-files. Proof of concept involves instrumenting scientific application and deriving performance information from the collected data. A database is not essential at this stage as extraction scripts simulate database queries. Two codes have been instrumented, these are *Spark98* and *Pyramid*.

3.0.3 Spark98 SMVP Kernels

Spark98 is a set of sparse matrix vector product kernels that include several shared memory and message passing parallel codes along with a sequential version. Spark98 is derived from Carnegie Mellon's Quake Project and designed to provide system designers and analysts with a small set of kernels that represent realistic SMVP applications [28]. The control flow hierarchies in this paper were derived from this code.

Fine-grained instrumentation was included prior to implementation of noise reduction algorithms. Analysis of standard POSIX compliant timing routines showed that overhead was highly platform and operating system development. For example, a call to `gettimeofday()` took 4 times longer under IRIX than on Solaris based computers. Introduction of noise reduction compensated for this overhead in probe data. Actual program execution, however, took 20% longer. This dilation was attributed to system call overhead and affects sequential codes that do not use MPI timing routines.

3.0.4 Pyramid Parallel Adaptive Mesh Refinement Library

Description. *Pyramid* is a software library for performing parallel adaptive mesh refinement on unstructured meshes [20]. Pyramid is being developed by the High Performance Computing and Applications Group at the California Institute of Technology's Jet Propulsion Laboratory. The library is designed to work on triangular and tetrahedral meshes and supports development of unstructured parallel applications such as finite element, finite volume, and visualization. The library is implemented in FORTRAN 90 and has an interface to MPI.

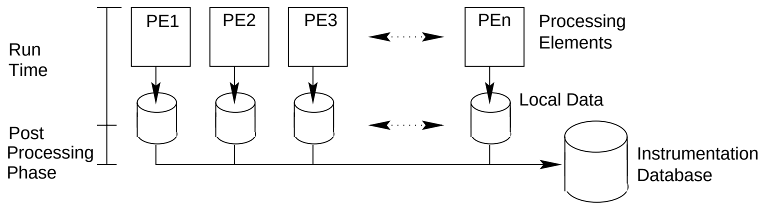


Figure 10: At run-time data is collected on each processing element and integrated during a postprocessing phase.

Introduction of scalable instrumentation probes involved a number of modifications to probe implementation and API:

- **FORTRAN - C interface.** The instrumentation API was extended such that probes can be introduced and the database can be initialized and closed from FORTRAN as well as from C.
- **MPI Integration.** Timing calls were replaced with calls to `MPI_wtime()` and `MPI_wtic()`. This ensures a standard interface to accurate timing routines regardless of platform. Probe data structures were extended to include processor identification.
- **Parallel Support.** The probe API was extended to include timing information specific to each processing element. Information collected on each node need to be stored separately and integrated into one database. Possible implementations involved having each node communicate probe data to the first node for output or having each node write data to a single database in a round-robin scheme. Both of these alternatives involved introducing synchronization delays to the executing program. Figure-10 shows the scheme that was implemented. Each node collects performance data locally at run-time. During post-processing phase these files are coalesced into one database. Postprocessing is currently done using PERL scripts that simulate database query operations and generate visualizations of collected data.

Figure-11 shows the input mesh and resulting mesh after three refinements used to test the instrumented Pyramid library. All tests were run on NASA Goddard's SGI/CrayT3E. The test program performed three refinements of the input beam-waveguide mesh. Figure-12 shows the structure of the test application.

Results. Figures 13 and 14 show execution time for a $1,978$ element mesh running with 16 and 32 processors respectively. When the mesh data is read in, it is distributed randomly to all processors. These graphs illustrate how moving from 16 to 32 nodes appears to induce a significant load imbalance in the `PhysicalAMR()` module. This imbalance is a result of the applications irregularity. Random distribution of the initial mesh is such that little or no refinement is required for mesh elements residing on processor 21.

Figure-15 shows the time spent executing probe specific operations for a large input mesh of $9,390$ elements running on 32 nodes. This data includes time spent calling MPI timing routines and manipulation of probe data structures and control flow hierarchy. This noise is negligible when compared to

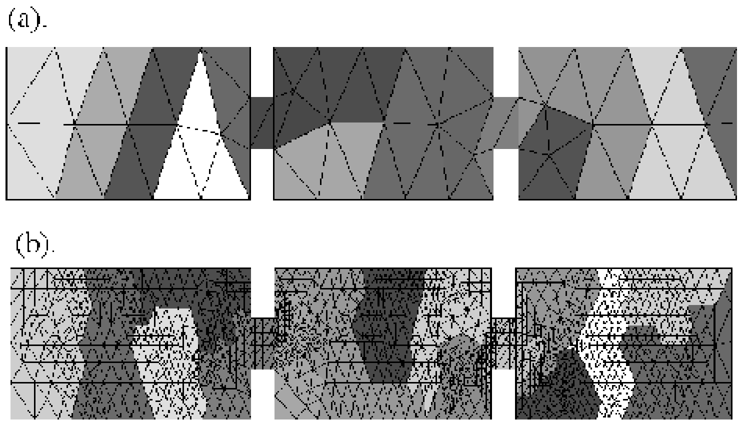


Figure 11: (a). Initial beam-waveguide mesh (b). The same mesh after three refinement phases. Shading indicate on which processors mesh elements reside.

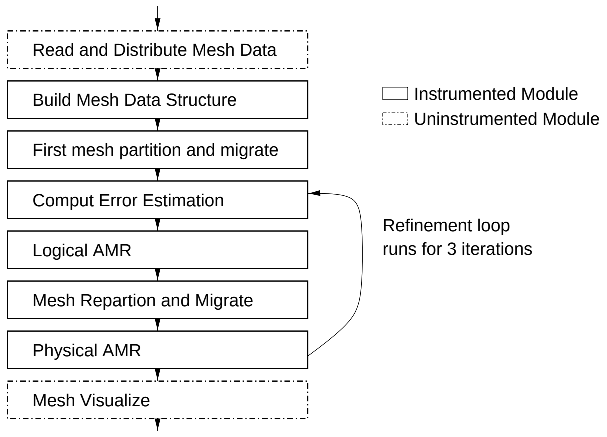


Figure 12: Flow of Pyramid test program for beam-waveguide mesh.

the times measured by probes. The graph in figure-16 is different in that the $z-axis$ shows the percent of total time spent in each module and not the actual execution time. This information is derived from timing data during the post-processing phase.

Verification. To ensure accuracy of the probe data, instrumented and pristine versions of the application were analyzed using PAT. Table-4 shows the measured execution time of the instrumented application along with the corresponding PAT instrumented version. In all cases, PAT and the instrumentation database (IDB) differ by less than one percent. The decision to use PAT was based on the fact that PAT has the least overhead and returns the most accurate timing data of any tool available on the SGI/CrayT3E.

To measure how probes dilate run time of the application, pristine and instrumented versions of the code were timed. Table-5 shows run time for pristine and instrumented versions of the code. The last column shows the run time measured by probe instrumentation. In some cases, instrumented code

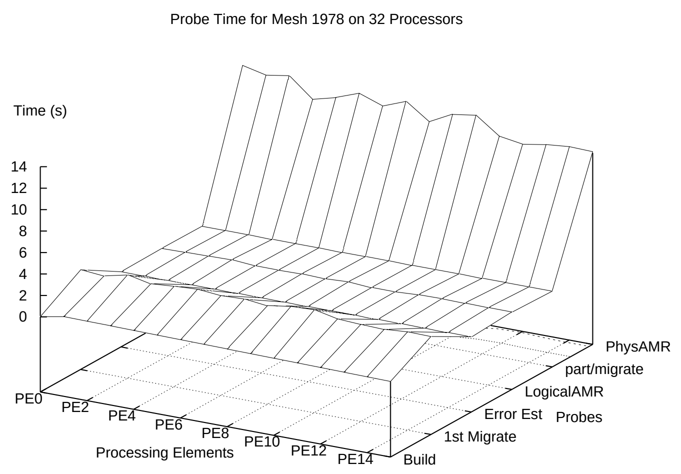


Figure 13: Probe times for 1978 mesh on 16 processors.

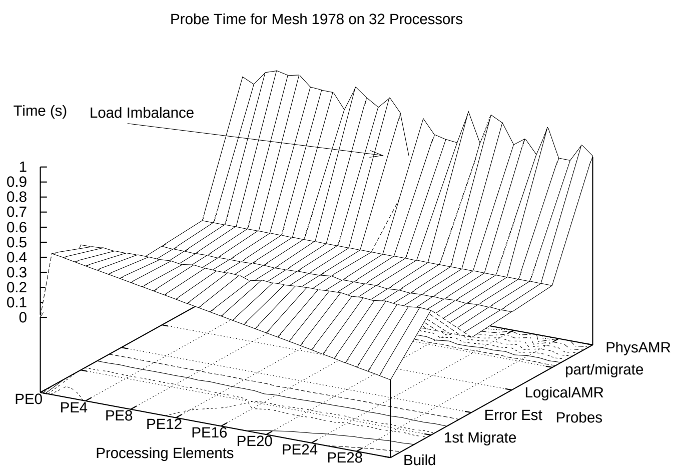


Figure 14: Probe times for 1978 mesh on 32 processors.

appeared to run faster than the pristine version. This is an effect of cache utilization which is amplified by small problem size. In all cases, the measured time is less than the wall clock time. This is because the noise reduction factors out instrumentation and system overhead whereas the `timex` command does not.

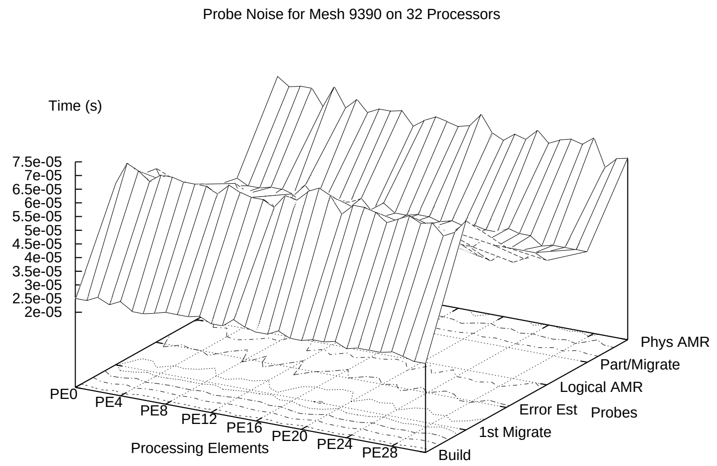


Figure 15: Probe noise for 9390 mesh on 32 processors.

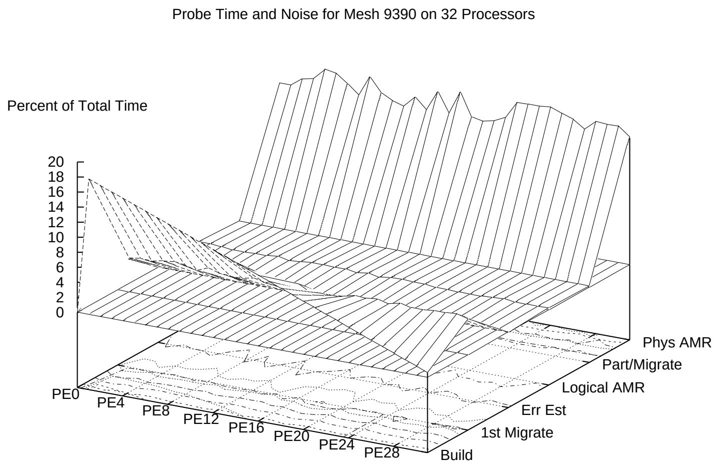


Figure 16: Percentage of time spent and noise contribution at each probe for 9390 mesh on 32 processors.

4 Support for Object Oriented Languages

Object oriented technology has significantly changed the way programs are developed. A corresponding change is needed in performance analysis of the resulting codes. [25] Traditional performance analysis focuses on the control flow graph of programs. Sequential and parallel codes differ in the number of simultaneous paths being traversed. Such control flow graph oriented views are insufficient for object oriented codes.

PROCESSORS	IDB	PAT
8	80.0 s	80.3 s
16	20.8 s	20.6 s
32	8.4 s	8.4 s

Table 4: IDB versus PAT for 2430 mesh on SGI/CrayT3E.

PROCESSORS	PRISTINE	INSTRUMENTED	IDB
8	80.2 s	80.4 s	80.0 s
16	21.8 s	21.4 s	20.8 s
32	9.1 s	9.1 s	8.4 s

Table 5: Pristine and instrumented execution times with probe times on SGI/CrayT3E for mesh 2430. Run times measured using the `timex` command.

Object oriented programs can explore inter-object or intra-object parallelism. The former is based on task parallelism in which multiple objects are executing concurrently. Intra-object parallelism explores data parallelism by processing a single instance of an object running on parallel processors (Figure- 17). Traditional control flow based techniques can be used to analyze performance of sequential and parallel member functions but they do not extend to inter-object parallelism. To make such extension, we introduce the *object space* that include all instantiated objects at a given time. Inter-object parallelism yields many control flow graphs representing simultaneous execution of member functions for objects in the object space. Some of these member functions may be running on multiple processors (data parallelism). We are investigating mechanisms that can map these disparate control flow graphs onto the object hierarchy and inter-object message passing. Such mapping enables analysis of critical path events spanning methods in multiple objects.

Instrumentation can benefit from object oriented program implementation by introducing a virtual instrumentation object from which all source code objects inherit methods and attributes needed for performance data collection. As a result, all objects have access to performance data structures and functionality for self instrumentation.

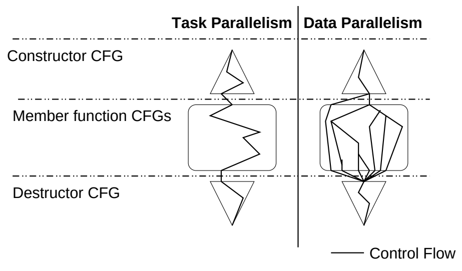


Figure 17: Data versus Task Parallelism for object oriented programs.

References

[1] Mark W. Beall and Mark S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engng.*, 40(9):1573–1596, 1997.

[2] Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *Computer*, 31(5):41–49, May 1998.

[3] Doreen Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NAS Systems Division, NASA Ames Research Center, March 1993.

[4] Mark J. Clement and Michael J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings Supercomputing '93*, pages 886–894, Los Alamitos, CA, USA, 1993. IEEE Comput. Soc. Press.

[5] Elmasri and Navathe. *Fundamentals of Database Systems Second Edition*. The Benjamin/Cummings Publishing Company, Inc., 1994.

[6] Thomas Fahringer. Estimating and optimizing performance for parallel programs. *Computer*, 28(11):47–56, November 1995.

[7] R. Fatoohi and S. Weeratunga. Performance evaluation of three distributed computing environments for scientific applications. In *Proceedings Supercomputing '94*, pages 400–409, Washington, DC, USA, November 1994.

[8] Joseph E. Flaherty, Raymond M. Loy, Can Özturan, Mark S. Shephard, Boleslaw K. Szymanski, James D. Teresco, and Louis H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1998.

[9] Michael T. Heath. *Visualization of Parallel and Distributed Systems*, chapter 31. McGraw-Hill, 1996.

[10] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[11] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, November 1995.

[12] Jukka Helin and Kimmo Kaski. Performance analysis of high-speed computers. In *Proceedings Supercomputing '89*, pages 797–808, New York, NY USA, November 1989. ACM.

[13] Jeffrey K. Hollingsworth, James E. Lumpp Jr., and Barton P. Miller. Techniques for performance measurement of parallel programs. Computer Sciences Department, University of Wisconsin and Department of Electrical Engineering, University of Kentucky.

[14] Anna Hondroudakis. Performance analysis tools for parallel programs. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, July 1995.

[15] Raj Jain. *The Art of Commuter Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley and Sons, Inc., 1991.

[16] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library version 2.0, June 1998.

[17] D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou, P. L. Springer, T. L. Sterling, and P. Wang. An assessment of a beowulf system for a wide class of analysis and design software. In *Advances in Engineering Software*, volume 26, pages 451–461, July 1998.

[18] David J. Kuck. What do users of parallel computer systems really need? *International Journal of Parallel Programming*, 22:99–127, 1994.

[19] Kei-Chun Li and Kang Zhang. Stamp: A stopwatch approach for monitoring performance of parallel programs. Department of Computing, Macquarie University, 1996.

[20] J. Lou, C. D. Norton, and T. Cwik. A robust and scalable software library for parallel adaptive refinement on unstructured meshes. *To appear in NASA Computational Aerosciences Workshop*, 1998.

[21] Victor W. Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[22] A.D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):433–50, July 1992.

[23] Barton P. Miller, Mark D. Callaghan Jeffrey K. Hollingsworth, Jonathan M. Cargille, R. Bruce Irvin, Karen L. Karavanik, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995. Special Issue on Performance Analysis Tools for Parallel and Distributed Computer Systems.

[24] Jeffrey Nesheiwat and Boleslaw K. Szymanski. Instrumentation database for performance analysis of scientific applications. In *Lecture Notes in Computer Science: Fourth International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1998.

[25] C. D. Norton. *Object Oriented Programming Paradigms in Scientific Computing.* PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, December 1996. UMI Company.

[26] Charles D. Norton and Boleslaw K. Szymanski. Monitoring scientific computations – an object oriented approach. In *Proc. 2nd Int. Conf. Parallel Processing and Applied Mathematics*, volume I, pages 104–116, September 1997.

[27] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter. Extensible parallel program performance visualization. In *Proceedings of the International Workshop on Modeling Analysis and Simulation of Computer and Telecommunication Systems 3rd*, pages pp. 205–211, Durham, NC, January 1995.

[28] David R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Sciences, Carnegie Mellon University, October 1997.

[29] Peter S. Pacheco. *Parallel Programming with MPI.* Morgan Kaufmann Publishers, Inc., 1997.

[30] Daniel A. Reed, Keith A. Shields, Will H. Scullin, Luis F. Tavera, and Christopher L. Elford. Virtual reality and parallel systems performance analysis. *Computer*, 28(11):57–67, November 1995.

[31] Diane T. Rover. Performance evaluation: Integrating techniques and tools into environments and frameworks. In *Proceedings Supercomputing '94*, pages 277–278, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.

[32] Amitabh B. Sinha and Laxmikant V. Kale. Towards automatic performance analysis. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 3, pages 53–60, Los Alamitos, CA, USA, August 1996. IEEE Comput. Soc. Press.

[33] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[34] J. D. Teresco. Pmdbtool. Personal communication with J. D. Teresco, October 1998.

[35] Abdul Waheed, Vincent F. Melfi, and Diane T. Rover. A model for instrumentation system management in concurrent computer systems. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, pages 432–441, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[36] J. C. Yan. Performance tuning with aims – an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 625–633, January 1994.