THESIS APPROVAL

The abstract and thesis of Christian Leland Hansen for the Master of Science in Computer Science were presented October 30, 2001, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS: ________________________________
Karen L. Karavanic

____
Jingke Li

____
Andrew M. Fraser
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL: ________________________________
Cynthia A. Brown

ABSTRACT

An abstract of the thesis of Christian Leland Hansen for the Master of Science in Computer Science presented October 30, 2001.

Title: Towards Comparative Profiling of Parallel Applications with PPerfDB

Due to the complex nature of parallel programming, it is difficult to diagnose and solve performance related problems. Knowledge of program behavior is obtained experimentally, with repeated runs of a slightly modified version of the application or the same code in different environments. In these circumstances, comparative performance analysis can provide meaningful insights into the subtle effects of system and code changes on parallel program behavior by highlighting the difference in performance results across executions.

I have designed and implemented modules which extend the PPerfDB performance tool to allow access to existing performance data generated by several commonly used tracing tools. Access occurs from within the experiment management framework provided by PPerfDB for the identification of system parameters, the representation of multiple sets of execution data, and the formulation of data queries. Furthermore, I have designed and implemented an additional module that will generate new data using dynamic instrumentation under the control of PPerfDB. This was done to enable the creation of novel experiments for performance hypothesis testing and to ultimately automate the diagnostic and tuning process.

As data from such diverse sources has very different representations, various techniques to allow comparisons are presented. I have generalized the definition of the Performance Difference operator, which automatically detects divergence in multiple data sets, and I have defined an Overlay operation to provide uniform access to both dynamically generated and tracefile based data. The use and application of these new operations along with an indication of some of the issues involved in the creation of a fully automatic comparative profilier is presented via several case studies performed on an IBM SP2 using different versions of an MPI application.

CONTENTS

TOWARDS COMPARATIVE PROFILING OF PARALLEL APPLICATIONS WITH PPERFDB

by

CHRISTIAN LELAND HANSEN

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2001

TABLES

FIGURES

1 INTRODUCTION

Optimizing parallel code to take full advantage of a system's theoretical computational power and approach perfect parallel speedup is a notoriously difficult task. With the wide variety of platforms, system configurations, communication libraries, and computational models in use, simply increasing the processor count or decreasing the cost of a particular function call rarely results in the expected drop in execution time. Achieving optimal performance requires an understanding of the complex interdependencies between the various hardware and software elements which influence code behavior. Solutions are reached experimentally, by modifying specific parameters of the system and executing the application in a controlled environment. Ideally, what we learn under a certain set of assumptions will be applicable in future situations, such as when porting the application to a different architecture or trying out a novel communication paradigm. However, to ascertain the effects certain changes have on the overall performance requires the ability to meaningfully compare two or more executions of an application.

Currently, there are a host of performance tools in existence providing detailed trace or log based information, including AIMS [1], Jumpshot [2], Paragraph/MPICL [3,4], Vampir[5], and VT[6]. This information can be used in an iterative form of tuning where the application is run, the logs analyzed, the code modified, and the cycle repeated. This method is only sufficient when considering applications of a very small size, as it becomes very costly when the application in question needs hours or days to run and requires a large amount of dedicated, high-demand resources. Furthermore, this method generates and then neglects an often massive amount of data which has gathered over the lifetime of a particular piece of code as it is run under various conditions. Other tools [7] have made attempts to relieve the burden of large amounts of data collecting found in trace based tools by supporting dynamic instrumentation [8]. Libraries based on this technology, such as Dyninst [9] or DPCL [10], provide an interface for inserting and removing small pre-defined pieces of code into the in-memory image of a running application. The advantage to this method is that it allows the measurement timing to be more flexible, targeting only the more interesting parts of an execution and adjusting the granularity to fit the problem. Coupled with more intelligent instrumentation management, this can drastically reduce the volume of recorded data.

One drawback to both types of tools is their lack of comparative techniques. While the analyst can bring up and visually compare a pair of execution traces by running two instances of the analysis tool, the tools themselves provide no information beyond a single iteration of the turning cycle. With whatever experience in tuning a particular application they may have, the analyst is left on their own in terms of qualifying and attributing the change. To address this concern, the PPerfDB research tool provides an experimental framework necessary for identifying the free parameters in a tuning environment and for formulating and testing performance hypotheses.

With PPerfDB as a foundation, I have created the framework necessary for a comparative profiler which can be used to identify and quantify the concrete effects of code and system adjustment on application performance. I have extended the capabilities of the original tool with several modules that analyze data gathered by three commonly available tracing tools, Jumpshot, Vampir, and VT, and an additional module which can generate performance data for new experiments using dynamic instrumentation. Comparative studies are possible between existing executions in the database, between reference, database executions and new executions run under the auspices of the tool, between two running executions, and between two or more processes in a single execution. Furthermore, as run-time application control and measurements will be done through the use of the DPCL instrumentation library, comparisons can be limited to particular phases of an application, thereby reducing the amount of data store and instrumentation perturbation.

Since an application may well undergo rather radical changes during its lifetime, differences across executions are first identified with the abstract representations and associated operations defined by PPerfDB. However, to obtain a more detailed comparison encompassing a diverse set of data sources, I have created the Overlay operation, which unifies the various data formats available to the tool with a single representation, and I have generalized the Performance Difference operator, which performs an automated search for discrepancies in performance data. Further qualification and quantification of such discrepancies can then be done by side-by-side comparison of flat profiles, call graph profiles, or event traces.

In the next section, I describe the concepts behind and function of PPerfDB. This section includes a discussion of the type of data available from the tracefile formats selected for inclusion in the tool and how this data fits into the given model. This is followed by a brief explanation of the principles behind dynamic instrumentation and how it was incorporated into the model. Section 3 defines the Performance Difference and Overlay operations. Section 4 covers several case studies highlighting what information can be obtained from such comparative studies and the issues involved in doing so, and Sections 5 and 6 list related work and future plans.

2 THE FRAMEWORK FOR COMPARATIVE ANALYSIS

To meaningfully compare performance data from two or more executions, there must exist a framework capable of abstracting away the different formats in which this data is reported, of compensating for partial data or different metric types, and for classifying and managing this data. To meet these requirements, the current comparative studies and data analyses rely upon the PPerfDB tool. Specifically, the tool provides a high level representation of individual executions, methods for querying and displaying underlying performance data for multiple executions at once, and operations for comparing and contrasting executions at an abstract level. As this tool initially only handled post-mortem data obtained via the Paradyn tool, extensions to PPerfDB were devised to generalize its data extraction and manipulation facilities to include several common tracefile formats and data created from within PPerfDB using its dynamic instrumentation capabilities.

2.1 PPerfDB

The PPerfDB research tool as conceived and implemented follows the original proposal of experiment management support for parallel application tuning [11]. The proposal defines an experiment space formed on the basis of the tuning parameters of the program and system under evaluation. At one level, the experiment space allows for the formalization of performance related hypotheses and subsequent testing; at another, it provides an organizational facility for multiple executions of a single application. I call the tuning parameters attributes, and they can also be considered as the categories of descriptive data ascribed to individual executions. Attributes identify the circumstances of the execution and may include such items as the number of processes, the platform, the communication library, or any other user-defined features common across experimental scenarios. By selection of a single attribute value, the analyst can limit their consideration to a subset of the total number of executions, i.e. only those executions that share that particular value. In this regard, if we are interested in quantifying the scalability of our code and we have defined attributes for the number of processors and the platform, we might select only a single platform, thereby reducing the list of free parameters in our experiments so that we examine only those that vary in the number of processors used. The execution space has been realized as the SpaceMap, where each execution is assigned a unique identifier (EID).

At the next level of detail, individual executions are represented by a hierarchical collection of resources, or identifiable elements of the runtime system. Resource hierarchies most commonly include, but are not limited to, code modules, processes in the computation, and message tags, although the actual components and their organization is dependent on the underlying system and the tool used to generate the performance data. To represent an entire execution, the various resource hierarchies are unified into a single tree structure called an EventMap under a virtual root node. A sample of an EventMap for CLOG data from a very simple parallel bucket sort algorithm using MPI, called BSMPI, is presented in Figure 1.

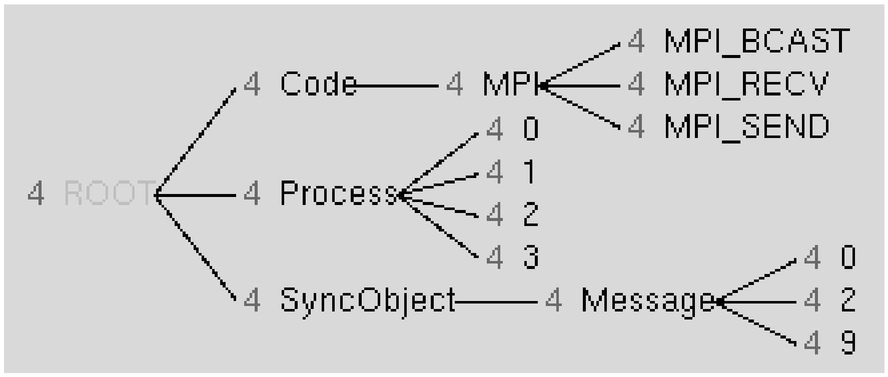


Figure 1: An EventMap for BSMPI
In this screenshot, the number 4 proceeding each node name is the EID of the execution. The ROOT node is a lighter color than the others to indicate that it is a placeholder.

Two operations have been defined on resource hierarchies, the Structural Merge and the Structural Difference. The Structural Merge is an algorithm that traverses both trees in a recursive top down fashion merging nodes with equivalent names and positions in the hierarchy, starting with the root node. Unmatched nodes are simply appended to the new tree in their original position. As EIDs are assigned to executions and their associated resources as powers of 2, the merged EventMap and its merged nodes are distinguished from the originals by their own unique EID obtained from summing the EIDs of the sources. Thus examination of the merged tree gives the analyst a glimpse into what changed and what remained the same from one execution to the next. A sample of a Structural Merge for partial EventMaps is shown in Figure 2.

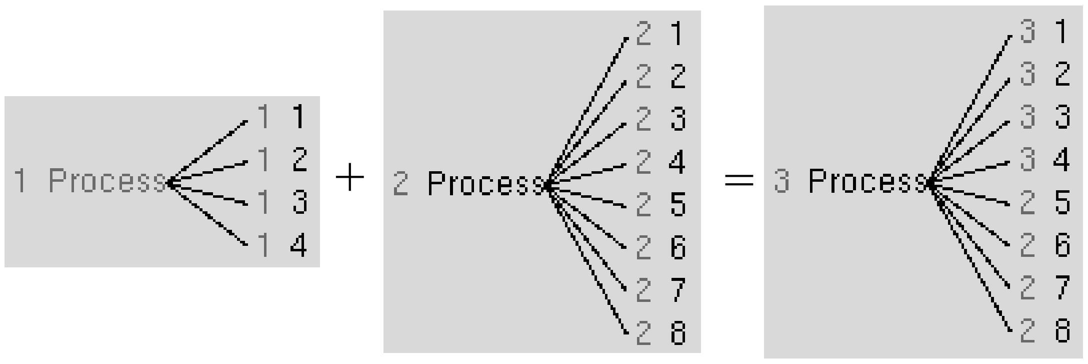


Figure 2: Structural Merge
For this series of screenshots, the SpaceMaps of two operand executions, EID 1 and 2, have nodes with equivalent names, resulting in merged nodes with EID 3.

However, the merging process is quite simplistic, and there often arise situations in which nodes representing equivalent components remain unmatched. This can occur if the spelling of a function name changes, or in cases where a program runs on different, yet identical nodes of the same machine. In such cases, it is possible to create mappings, allowing two nodes of different executions to be manually merged into a single node or to allow a parent and its only child of the same hierarchy to collapse into a single level. This is useful when the distinction between the two is unnecessary and prevents mapping between executions.

The Performance Difference operation takes a merged EventMap as its starting point and, again proceeding in a top down manner, removes all the nodes which were common to both executions except in the instance where a non merged node lies beneath it in the hierarchy and it must be retained as a placeholder. The intention of this operation is to provide immediate access into what has changed from one execution to the next by highlighting those components which were not in common. An example of this using the previous partial EventMaps is given in Figure 3.

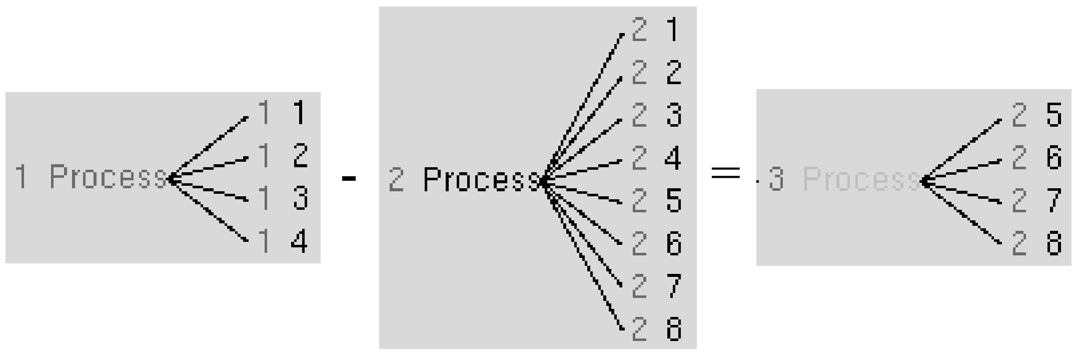


Figure 3: Structural Difference
In this series of screenshots, the operand execution EID 2 does not share several nodes found in EID 1. Since these nodes are the children of a common Process node, this node has been retained as a placeholder, indicated by its lighter color.

Another aspect to the representation of execution data is a list of metrics, the contents of which are statically determined by the tool used to generate performance measurements. For any basic profiling tool with default settings in a parallel environment, this list includes at least function call and message counts, as well as function costs and message sizes. More advanced tools or further user configuration will expand this list significantly. As a set of metrics is associated with each execution, the extension of the Structural Merge and Structural Difference Operations to encompass metric lists is simply a union of the two sets.

Apart from EventMaps and their operations giving a high level view of the run-time changes, they form the basis for queries into data files for performance information or requests for instrumentation when source files are non-existent. Selecting a single resource from each hierarchy in the representation, we create a focus, which combined with a metric and a time period, references a particular set of performance data. This operation defines what is known as the Performance Result function. In the case of a merged EventMap, the Performance Result function returns several sets of data, one for each operand execution of the merged EventMap.

A prototype of PPerfDB was developed in Tcl/Tk and currently allows the creation of an application specific SpaceMap, the definition of attributes, the loading of any number of executions, and the means to select multiple executions at once on the basis of their attribute values. Extensions to Tcl, written in C++, also provide an abstract class interface for executions, for the storage and management of execution specific EventMaps, including the Structural Merge and Difference Operations, and for the storage and handling of metric lists. EventMaps are visualized via the Tree [12] widget along with associated information presented through standard Tk objects.

2.2 Tracing Tools

In the original prototype of PPerfDB, only Paradyn data was accessible through the Performance Result function. While of demonstrated utility [13], one goal of PPerfDB is to provide a single point of access to a diverse body of performance data. Particularly, this includes historical data, which consists of the existing files generated primarily by trace or log based libraries. To this end, several common tracefile formats were examined and incorporated into the current model.

Tracing libraries linked into application code timestamp events, such as function calls or message sends, as they occur, so that on completion of the execution the behavior of the application can be recreated. In general, from the region defining beginning and ending marks provided by these event logs, it is a relatively simple matter to calculate function count and cost information, point-to-point message delivery times, and from the extra information accompanying send and receive calls, message sizes. However, I have not restricted myself to this basic set but rather define the list on a case by case basis. This is the result of an effort not to discard available data by enforcing a particular model over all event formats, as a more detailed tracefile format may allow a richer set of metrics. As such, the set of metrics available to a particular execution are determined statically and are dependent on the library used to generate the tracefile.

The use of a particular tracefile library also determines the type of resource hierarchies available, although it does not determine their actual content. While we may know that a particular tracefile provides memory allocation information, we do not know *a priori* the actual addresses and sizes of the allocated blocks. Therefore, the

EventMap associated with an execution is determined on an individual tracefile by tracefile basis. In practical terms, this means that the tracefile must be scanned to determine these values and that precisely one pass over the entirety of each tracefile is required when it is first loaded into the system.

Of the large variety of these libraries and their associated formats, the PPerfDB tool currently supports tracefiles in the CLOG [2] format as generated by the extension library that comes as part of MPICH [12] and Compaq's MPI [14]; the log format generated by Pallas' Vampirtrace [17]; and the default tracing library that comes with IBM's Parallel Operating Environment, at one time associated with the now deprecated tool VT [6].

Metric	Description
func_calls	The number of function calls
func_duration	The cost of a function
msg_bytes	The number of bytes in a message
msg_deliv_time	The point-to-point delivery time of a message

Table 1: Metrics in Common to VT, Vampir, and CLOG

Of the three formats, the first two are available on a number of systems and concern themselves primarily with the MPI communication paradigm. As a result, these two very similar formats by default report only on MPI function calls. With event data equivalent to that mentioned in the general case, they provide only the basic set of metrics previously described and summarized in Table 1. For the EventMap, both generate a code hierarchy which includes MPI calls, a process hierarchy that includes tasks in the computation, and a synchronization hierarchy that includes message tags. As it is possible in both cases to have user-inserted calls to the profiling library provide additional information on other functions and user-defined blocks, these will occasionally appear in the code hierarchy.

Metric	Description
cpu_user	Percentage of user CPU utilization
cpu_kernel	Percentage of kernel CPU utilization
cpu_wait	Percentage of wait CPU utilization
cpu_idle	Percentage of idle CPU utilization
io_read	Number of blocks read from disk
io_write	Number of writes made to disk
io_xfer	Number of transfers to and from disk
io_sent	TCP/IP packets sent
io_recv	TCP/IP packets received
proc_ctxtsw	Process context switches
proc_syscall	Process system calls
proc_pgflt	Process page faults

Table 2: VT specific metrics

The VT format, also concerned with the MPI paradigm, includes the same set of metrics and resource hierarchies found in the other two formats. However, by being tied to a particular system, both aspects of the representation have the ability to be far more comprehensive. With the tracing level set accordingly, the VT format can include periodic samples of process state information such as CPU, I/O, socket, and memory usage statistics. These additional metrics are presented in Table 2. Additionally, having knowledge of the SP2 architecture, the VT format presents a deeper process hierarchy with an intermediate level providing the actual node names to which processes were assigned. An example of this can be seen in Figure 8.

To add these components to the existing prototype, tracefile specific Perl modules were written, along with a generic interface for generating resource hierarchies, providing metric lists, and extracting performance results. Perl was chosen for its ease and agility in handling text strings, as all three formats are first translated into their ASCII equivalents for portability reasons. The interface scripts are called by Tcl code and the performance data that is returned can be visualized through standard means, such as graphs, histograms, and tables. At the moment, each new performance result query requires a single pass through the associated tracefile. Considering the extremely large size of the majority of these files, the time required to extract the relevant data for multiple queries of several executions can become unacceptable. Thus, once data has been requested, it is now cached, providing a significant reduction in the amount of time processing data. Work is underway both to further reduce the number of passes required over a tracefile and to automate the extraction process so that cached data is used even for the first request, however, that effort goes beyond the scope of this thesis.

2.3 Dynamic Instrumentation

As scalability is an important consideration of any parallel performance tool, one limitation of relying exclusively on tracefiles for analysis is that these tracefiles can become very large, and require a substantial amount of time simply for post-processing. An alternative, called dynamic instrumentation, implemented by the Dyninst [9] and DPCL [10] libraries, gives the user and tool designer the ability to insert and remove instrumentation at will during an application's execution, rather than establishing when and where to record events beforehand. By formulating the instrumentation code as an abstract syntax tree, the instrumentation library can compile this code for the target architecture and modify the running image to branch to the instrumentation code while in process [8]. It is possible to allocate variables, perform simple logical and arithmetic calculations, or to call more complex functions which are part of a dynamically loaded instrumentation library. By being able to report back the values of tool-created and application variables and to stop and start the application at will, a wide degree of flexibility is available for application tuning.

As part of the goal to support hypothesis testing and the launching of new experiments for the collection of missing or incomplete performance data, dynamic instrumentation was incorporated into the existing PPerfDB framework. Although both the libraries mentioned arise from a common code base, for the purposes of this project the DPCL library was chosen over Dyninst as it has already incorporated support for parallel applications by handling some of the more complex communication issues and data management techniques required when running a multiprocess application on remote nodes of a computational cluster. A detailed discussion of DPCL's structure and its library interface can be found in IBM's literature [10].

In terms of implementation, DPCL support was included as a separate, independent library interface component written in C++, which is started only when the need for its services is indicated. This can be done by adding a new dataless execution to the database and then requesting a view of the EventMap which can either start the target application using the DPCL library or connect to an already running application by additionally providing the library with the PIDs of the computation. Once communication with the target is established, the number of processes is reported back to the user interface and the object code is explored in order to build an EventMap for the execution. Since DPCL allows instrumentation to be inserted at the entry and exit of function calls and it allows the inspection of program variables, this translates into the presence of code and memory hierarchies in the EventMap.

The list of metrics available to a dynamic execution corresponds directly to the instrumentation probes implemented by the library interface component and a supporting probe library. Currently, PPerfDB includes function and message counts, and function costs. Each probe, when activated by the occurrence of a specified event, reports back to the user interface a timestamped data value appropriate to the metric in question. To approximate the time of an event relative to the start of the application, timestamps are generated from a sum of the user and system times reported from the `getrusage()` system call. Counts are calculated by simply marking the time of function entry, and message counts are obtained in an identical fashion, but only for message passing routines. Functional costs are calculated by timestamping the entry and exit points of a function call and reporting the difference.

Instrumentation is requested through the composition of a performance query, which is translated by the library interface module into the appropriate probe type and insertion point(s). As data is reported back to the tool interface, updates are made to the display chosen to initiate the request. Since PPerfDB allows a single visualization to display information from more than one execution ata time, this means it is possible to display dynamic data against a background of tracefile data, or to simultaneously display data from two instances of the library interface module, each monitoring a separate execution of the target application. When the application terminates or when instrumentation is removed and PPerfDB disconnects from the running program as per user action, the reported performance data and EventMaps are automatically stored for later retrieval and analysis.

3 COMPARATIVE PERFORMANCE ANALYSIS

3.1 Performance Difference Operator

While post-mortem analysis and run-time feedback and control are useful in and of themselves, additional insight can be gained by expanding analysis to include the changes witnessed over several executions. The performance difference operator was defined as a starting point for this type of analysis. It is an algorithm for locating points of interest in the resource hierarchy indicated by divergent results between two or more executions. After being characterized by the selection of a metric and a threshold value, the algorithm starts by obtaining a performance result for an application level focus, i.e. the children of the root node, then obtains a performance result for each source execution of a merged EventMap, and finally compares the distance between results to the theshold value. If this distance falls within the threshold, the process is stopped. However, if the threshold is exceeded, a process of focus expansion is undergone, in which each node of the current focus is replaced successively by its child nodes, creating a list of new foci which are used in turn to generate performance results. For each set of results, a comparison is made between their difference and the threshold value, and those which exceed the value are appended to a list of failed foci and further expanded. This process is repeated until either all the calculated distances lie within the threshold or the leaf nodes of the tree are reached. Effectively, this algorithm performs a breadth first traversal of our resource hierarchy, obtaining performance results for every possible combination of foci, and returning a list of foci which fail the threshold check. This list is useful in that it highlights the sources of divergence between execution times, greatly narrowing our further studies.

Unfortunately, this simple explanation belies some of the complex considerations underlying the operation. The actual mechanisms for calculating the difference between two different executions is not a trivial operation in the case of diverse data sources. An event based sequence of records, such as returned from the tracing tools and dynamic instrumentation as implemented in PPerfDB has no unifying time basis. This problem can be alleviated in one of two ways. The first is to define some type of summarizing operation on the data returned from the performance result operation, essentially converting it to profiled data. For some of the existing metrics, such as function counts or cost, or communication loads, this may indicate that a sum over the data is in order. For data sampled at a regular interval, such as seen in the system information provided in VT tracefiles and Paradyn histograms, a comparison of means may be more appropriate. To keep the operation as general as possible, several functions have been defined, such as the sum, the mean, the minimum, the maximum, and the standard deviation, any of which can further qualify the performance difference operation. In this regard, the distance between two performance results is defined as the algebraic difference between the results of the given summing operation applied to them. The second method is to divide the timeline up into discrete intervals and interpolate event results to match these intervals. This would allow the performance results to be treated as discrete functions and the difference defined as the distance between two functions, thereby providing time specific information into when deviations occur.

Two more difficulties with the performance difference operator arise when the time scales of two executions differ greatly or when we are faced with partial or incomplete data, such as that generated by the dynamic instrumentation of DPCL or Paradyn. The first of these situations occurs when we are comparing across platforms. Here, we may face a situation in which our application runs in a dedicated environment one system, but on another it may suffer the effects of time sharing. Or, we have less expensive, more frequent access to a slower machine, and therefore a richer set of results, which we would like to use in the context of a newer, faster, but similar system. In such instances, it is reasonable to normalize the timelines and associated timing values and support for this operation is provided. Currently, incomplete or absent data is handled by leaving the performance difference operator undefined for those segments. Thus for summary and functional comparisons the tool only reports partially defined results.

3.2 Overlays

These is another limitation to the application of the Performance Difference operator and to data queries on multiple executions in general. Once we have chosen an axis of variation among the available executions and would like to look beyond the EventMap and at the differences specifically related to individual performance results, we are presented with the problem of how to reconcile possibly very different representations of our application. Among the tracefiles themselves, there are differences in resource naming and in the depth of information available. Between tracefiles and dynamically gathered data, there is an even greater difference in representation, particularly in the arena of code resources, as DPCL provides listings organized in terms of object code modules, the functions defined therein, and then the functions called from within these functions, resulting in multiple locations for a single function.

While correlation between representations can handled in a manual, bottom-up fashion by merging, renaming, and collapsing levels of a merged EventMap until a final, common representation is obtained, it is also possible to do this automatically in a top-down manner with the use of Overlays. Overlays are essentially a set of bridging, collating, and renaming directives implemented as an artificial resource hierarchy, a list of equivalencies, and a minimal set of metrics, all of which are user configurable. Conceptually, Overlays represent what is considered "interesting" for study by the analyst and as such they can correspond to various commonly used parallel computational paradigms.

At the level of implementation, application of an Overlay to an execution consists of forming the intersection of the two sets of metrics and aliasing the resources in the Overlay's hierarchy to those in the execution (represented by a dotted line in the following figures). Aliasing occurs during a breadth first traversal of the Overlay hierarchy alongside the executions, when links are created between nodes with identical names or with those matching a pair of equivalencies found in the accompanying list. If a match is not found at the same level in both trees, nodes deeper in the executions's subtree are searched.

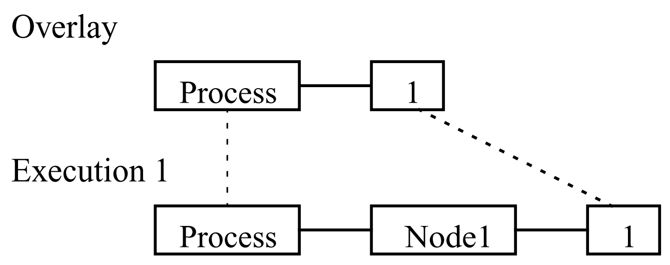


Figure 4: Bridging Effect of an Overlay
This diagram shows the links made between nodes of an Overlay and those of an executions EventMap. The links are represented by dotted lines.

If nothing is found to match a Overlay node, that node is dropped from the representation. Allowing the algorithm to descend further into the execution's tree to search for matches allows for the bridging of nodes (Figure 4) when a richer representation is given than is common across executions, and continuing the search after a single match is found, so that multiple links are established implements the idea of collating data (Figure 5). Providing a list of equivalencies enabled me to actuate the renaming of nodes to a standard scheme (Figure 6). This is useful in light of the original impetus to represent the hierarchies as close to the original data source as possible.

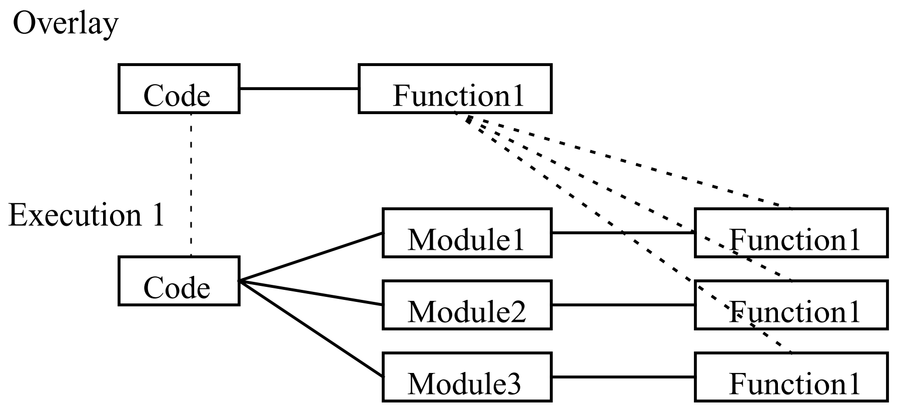


Figure 5: Collating Effect of an Overlay
This diagram shows the multiple links made between an Overlay and those of an executions EventMap. The links are represented by dotted lines.

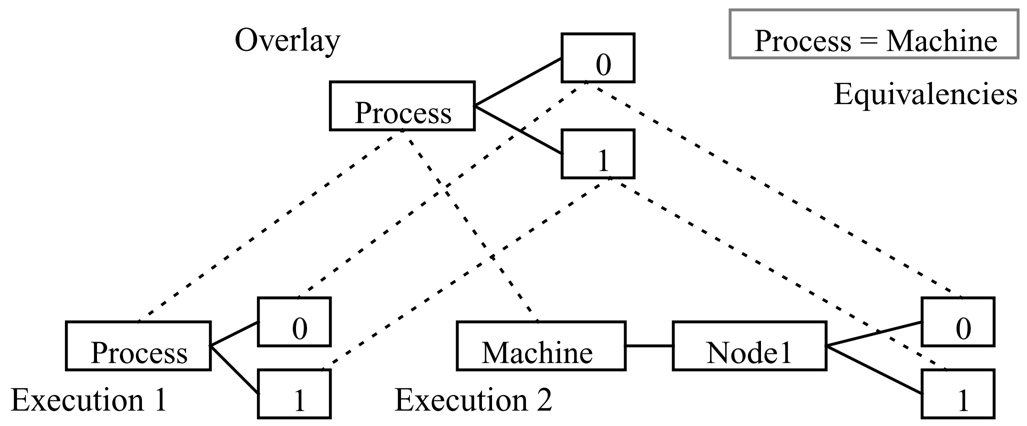


Figure 6: Renaming Effect of an Overlay
This diagram shows the links made between an Overlay and two different executions Event-Maps when applied successively. The equivalencies frame shows the provided name mappings used in creating the links.

Thus, on a performance result query, a focus is generated in the standard, pre-defined fashion, but before the actual data store is probed, this query undergoes a translation into a representation known to the original data store. Each one of the focus generates a list of aliases of which all the combinations are used to generate performance data on a per execution basis. The various results are then merged into a single result.

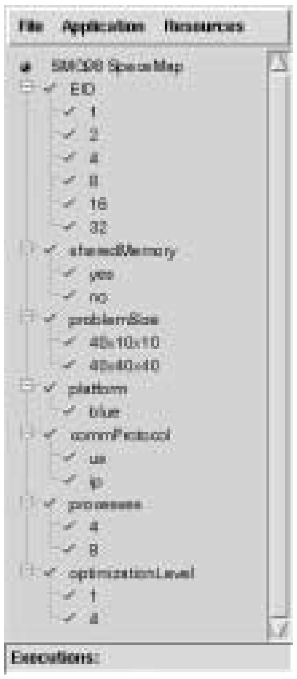
4 CASE STUDIES

In the following section, the use of the PPerfDB will be demonstrated along with some examples of how this information can be useful for performance tuning. PPerfDB runs on Solaris, AIX, and Linux, but with DPCL support currently only available for AIX, AIX was used in these studies. Throughout the examples, a single representative application was used for study, called SMG98.

4.1 SMG98

SMG98 is a semicoarsening multigrid solver developed at Lawrence Livermore National Laboratories [17]. The algorithm was designed to solve the systems of linear equations involved in finite difference, finite volume, or finite element discrete diffusion equations on distributed memory architectures. Parallelism is achieved by data decomposition according to the specified processor topology. The application's behavior is common to many parallel scientific codes in that its performance is dependant on how the data is partioned and distributed among the computational elements. The code was written in C and can be used for 2D and 3D problems, where the problems size per processor and processor topology can be specified on the command line. A thorough study of this algorithm's scalability on ASCI Blue can be found in [17].

Figure 7 shows the starting SpaceMap for the SMG98 application with six experiments already added. All of these were obtained by running the application on ASCI Blue, which is located at LLNL.



```
SMG98 SpaceMap
  ✓ EID
      ✓ 1
      ✓ 2
      ✓ 4
      ✓ 8
      ✓ 16
      ✓ 32
  ✓ sharedMemory
      ✓ yes
      ✓ no
  ✓ problemSize
      ✓ 40x10x10
      ✓ 40x40x40
  ✓ platform
      ✓ blue
  ✓ commProtocol
      ✓ us
      ✓ ip
  ✓ processes
      ✓ 4
      ✓ 8
  ✓ optimizationLevel
      ✓ 1
      ✓ 4
```

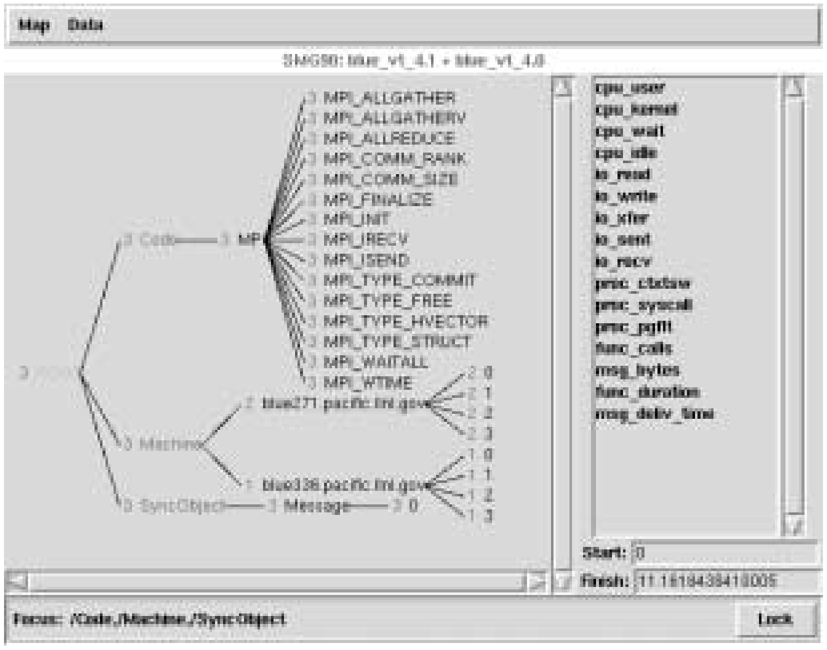
Figure 7: SMG98 SpaceMap
This screenshot shows the various attributes defined for this application along with the associated values assigned to individual executions of the application.

Several attributes have also been defined, such as whether or not shared memory was used for communication, the problem size per processor, the platform, the communication protocol, the number of processors used, and the compiler optimization level. A discussion of how the represented values apply to the executions of the study is covered in more detail in the following cases.

4.2 Compiler Optimization

For this initial example, SMG98 was run twice on ASCI Blue using a single 4-processor node. The program was compiled once with a basic level of optimization and a second time with very aggressive optimization (EIDs 1 and 2, respectively). The IBM version of MPI and their VT tracing facility was used for both cases. When run, the processor topology was set to 4x1x1 and the problem size to 40x10x10. The expectation in this test was that compiler optimization should generally provide for better CPU utilization, and with all other code and system paramemters being equal, a faster running program. However, being a parallel application, there was a possibility that this benefit may be offset by increased synchronization time.

As a starting point, the merged EventMap was created and is shown in Figure 8. A plot of executions times showed that I had indeed obtained a modest 1% speedup by using more rigorous optimization. Since an improvement in processor utilization should show a decrease in CPU wait and idle times, these two metrics were considered for the Performance Difference operator.



Start: 0
Finish: 11.1618439418005
Focus: /Code,/Machine,/SyncObject

Figure 8: Merged EventMap for Two VT Tracefiles

This screenshot shows the merged EventMap of EID 1 and 2 in the left frame and the list of available metrics in the right frame. The chosen time interval is found below the list of metrics and the selected focus can be seen at the very bottom.

Before being able to proceed, however, it was noted in the Machine hierarchy that a different host was used for each execution, spoiling comparisons between like numbered processes. As the nodes of ASCI Blue are architecturally equivalent, this distinction was unnecessary for the current study and these nodes were merged. Running the performance difference operator with a comparison between statistical means and a threshold value set to 5%, a short list of foci that exceeded the threshold was returned. The list showed that this size of a discrepancy had occurred for all the processes in the computation, but not specifically within any particular MPI function call.

Examination of the CPU idle time for a single process over the entire run of the application is provided in Figure 9.

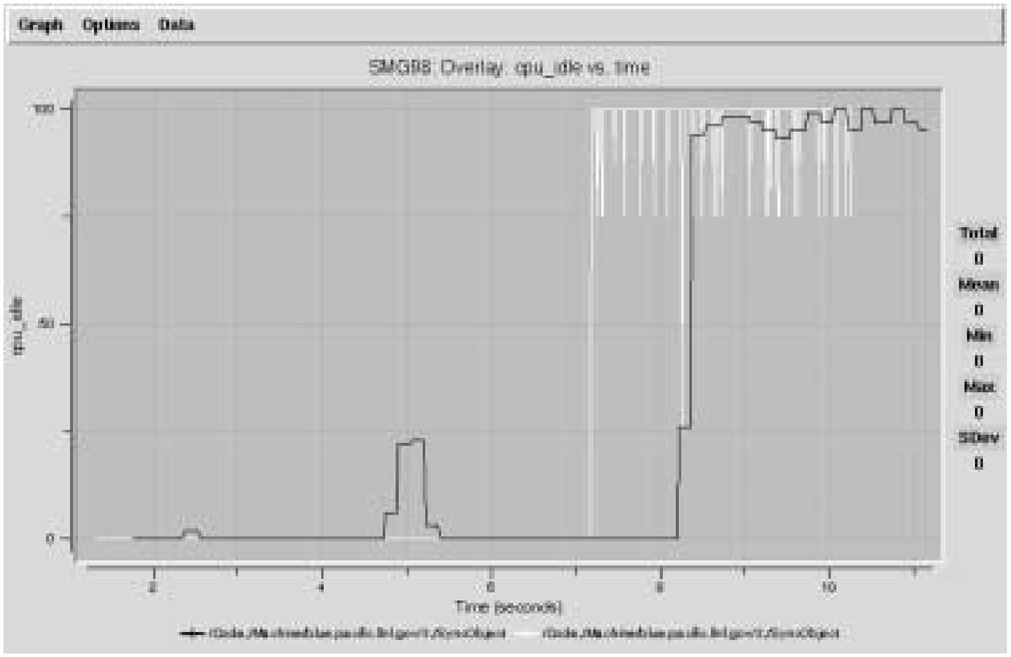


Figure 9: CPU Idle Rates for Minimal and Aggressive Compiler Optimizations
In this screenshot, the CPU_Idle rate for minimal optimization is drawn in black and aggressive optimization in white.

As the graph shows, there was decent savings in idle cycles at around the 5 second mark, where the more optimized code took full advantage of the CPU. This allowed it to finsh the computation faster, thus reaching the final data gathering phase where the processor went almost entirely idle at about the 7.16 second mark, approximately 1.2 seconds before the less optimized run. To more specifically locate the area of code that received the greatest benefit from optimization, the performance difference operator was repeated down to a percentage point in difference, but with no change in the result. Unfortunately, the optimization was not particular to any MPI call, and therefore the VT tracefile format was not able to pinpoint the particular function. To obtain this information, further studies using Paradyn or a DPCL module enhanced with CPU statistics were warranted. Unfortunately, I was unable to pursue this route due to time constraints.

4.3 Communication Protocol

In the second scenario, SMG98 was run again on ASCI Blue, this time with the intention of observing the effects of using different network protocols on communication times. Two runs of the application were done, one using Vampir to gather data and another in which DPCL was used. Both run were done with 8 processors, a topology of 2x2x2, and an problem size of 40x40x40 per processor. The IBM SP2 has two network protocols available for MPI communication, the proprietary US protocol over a dedicated switch and the more widely used IP which is shared among jobs. In this study, Vampir was used to measure US performance and DPCL to measure IP.

Since the code was unmodified for the run, Vampir trace data only reported on MPI calls, leaving a short list of possible candidates for profile comparisons (Figure 10).

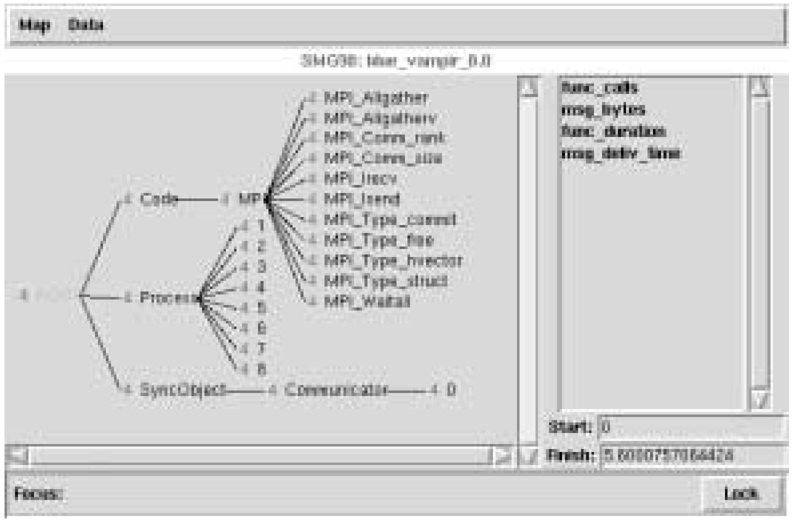
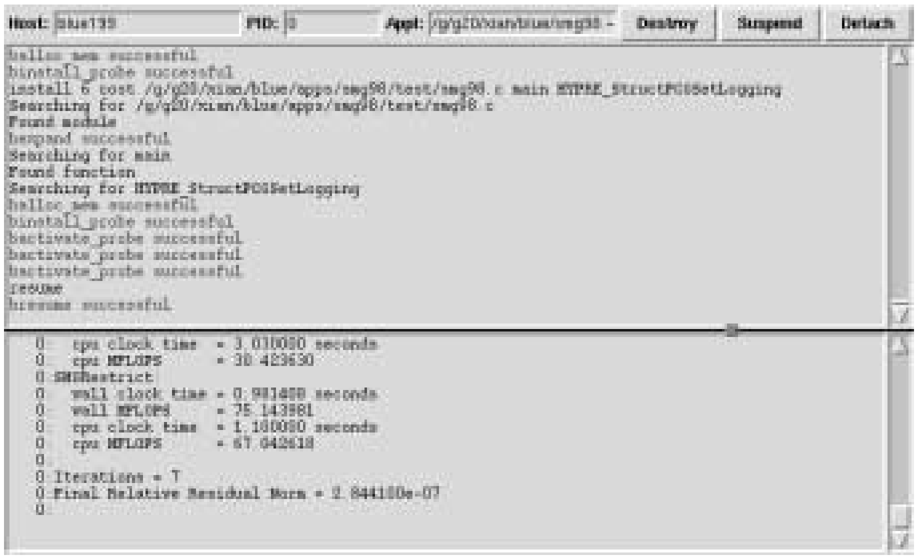


Figure 10: Vampir Tracefile
This screenshot shows the EventMap and list of metrics for a single execution, EID 4.

As non-blocking sends and receives were used for communication, MPI_Waitall was by far the most significant contributor to communication time and was selected for this initial study. Dynamic instrumentation was thus inserted to measure the duration of each MPI_Waitall call (Figure 11), and to limit overhead, this instrumentation was limited to a single process in the computation.



```
balloc_mem successful
binstall_probe successful
install 6 cost /g/g20/xian/blue/apps/smg98/test/smg98.c main HYPRE_StructPCGSetLogging
Searching for /g/g20/xian/blue/apps/smg98/test/smg98.c
Found module
bexpand successful
Searching for main
Found function
Searching for HYPRE_StructPCGSetLogging
balloc_mem successful
binstall_probe successful
bactivate_probe successful
bactivate_probe successful
bactivate_probe successful
resume
bresume successful
```

```
 0    cpu clock time  = 3.030080 seconds
 0    cpu MFLOPS      = 30.423630
 0 SMGRestrict
 0    wall clock time = 0.981400 seconds
 0    wall MFLOPS     = 75.143981
 0    cpu clock time  = 1.100080 seconds
 0    cpu MFLOPS      = 67.042618
 0
 0 Iterations = 7
 0 Final Relative Residual Norm = 2.844180e-07
 0
```

Figure 11: DPCL Control with Program Output
This screenshot shows application controls along the top, the status of the instrumentation in the middle frame, and the target application's output in the bottom frame.

As in the previous case, there were a number of discrepancies between EventMaps, but unfortunately at a much less manageable scale. Figure 12 shows just a portion of the EventMap created using DPCL. As can be seen from the image, DPCL provides access to the entirety of the code, including libraries added by default by the system, plus it allows access to program variables, represented in the EventMap by a Memory hierarchy.

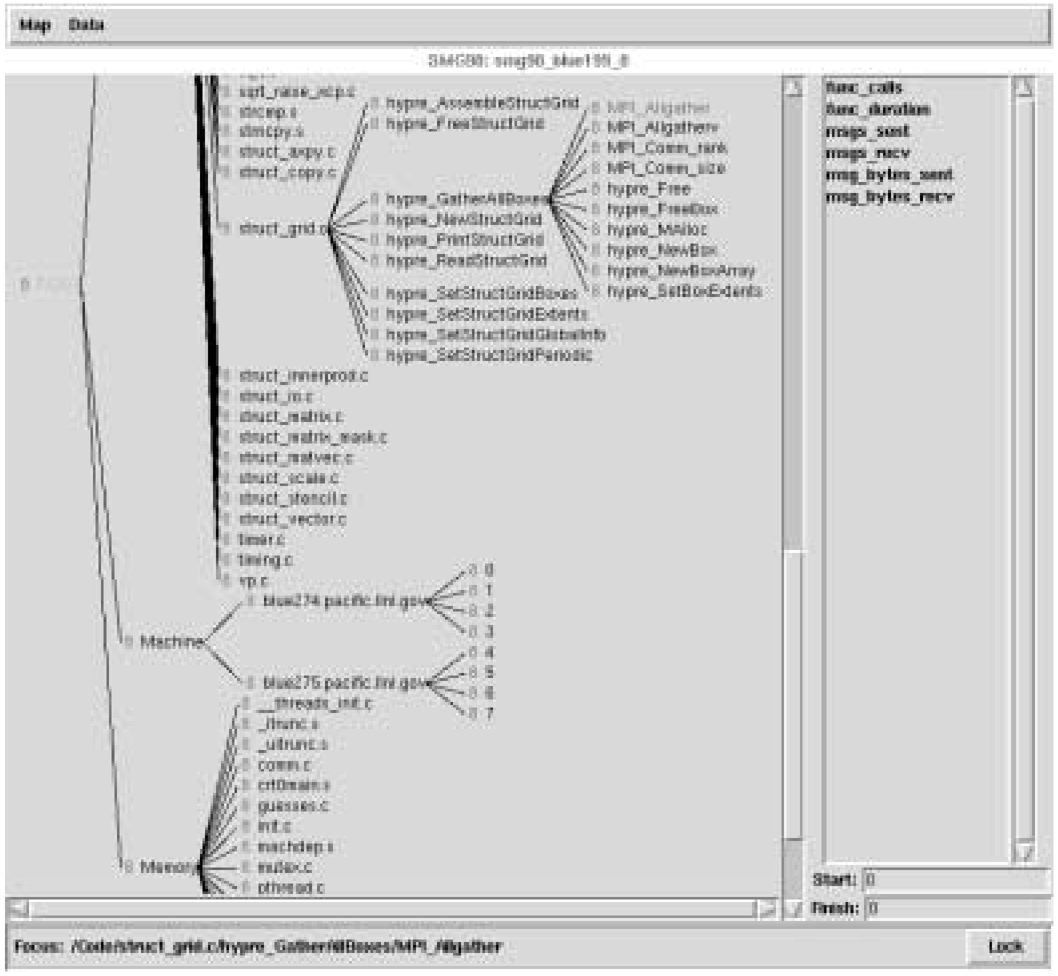
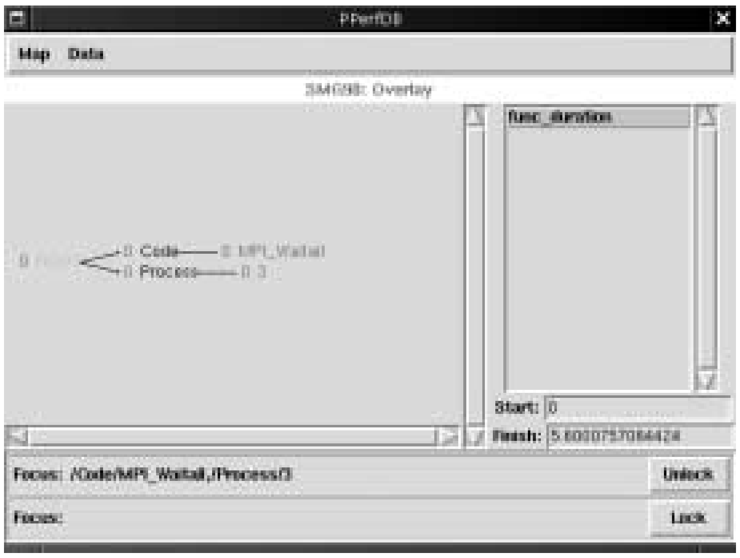


Figure 12: DPCL view of SMG98

As a Performance Merge operation would find nothing in common between these representations and manually merging and renaming nodes would be restrictively time consuming, an Overlay was applied. As the Vampir EventMap was already quite close to the minimal description of resources common between the two executions, its Code and MPI nodes were merged into a single Code node, and the whole EventMap was saved as an Overlay. An equivalency was also created between Process and Machine, so that the Machine node would be mapped to the Process node of the Overlay. Once applied (Figure 13), it was a simple matter to see what data was available in common, and to create displays based on the data.



Start: 0
Finish: 5.6000757084424
Focus: /Code/MPI_Waitall,/Process/3
Focus:

Figure 13: Data Overlay
In this screenshot, the results of applying an Overlay (always assigned EID 0) are shown.

The final difficulty that remained to a direct comparsion between performance data from the two formats was the perturbation caused by instrumentation overhead. To compensate for this, the solve time of another run using DPCL and the US protocol was recorded, and this time, 36.16s, compared against the Vampir time, 38.71s provided a scaling factor of 0.93 which was applied to all Vampir reported times. Figure

14 gives the results for a comparison between functional costs for a single process in the computation.

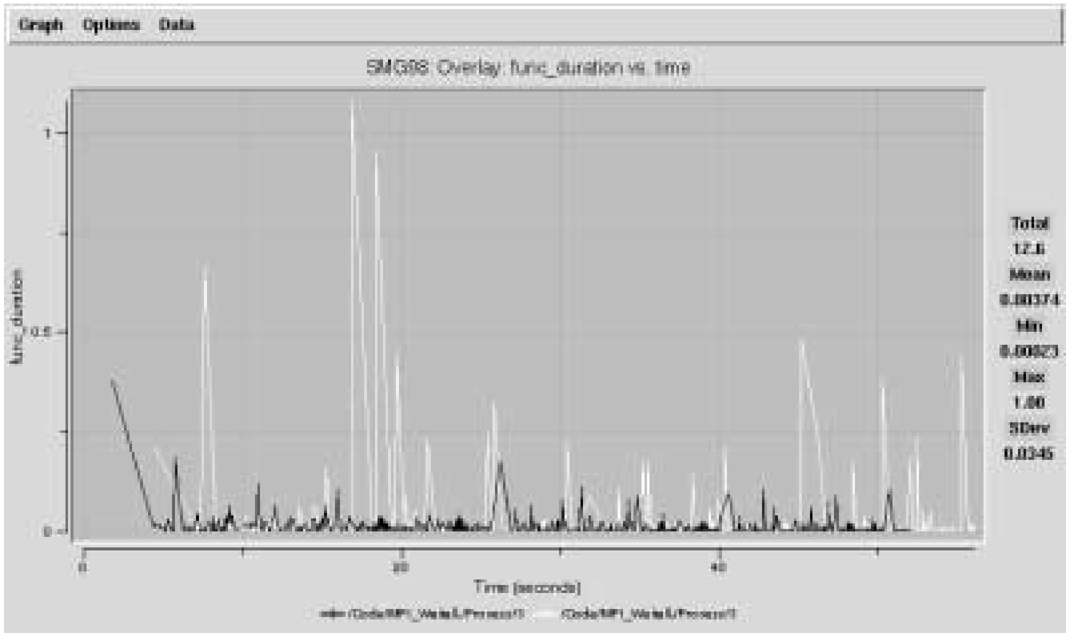
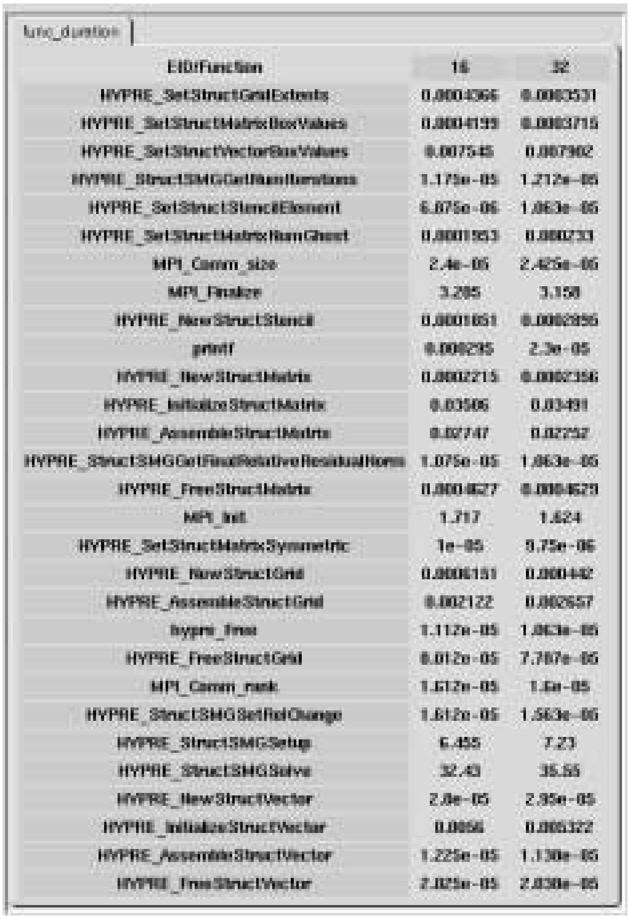


Figure 14: MPI_Waitall Times for US and IP Communication Protocols
In this screenshot, the MPI_Waitall time for the US protocol is drawn in black and for the IP protocol in white. The summary statistics for the IP protocol can be seen to the right of the graph.

As the plot shows there was an improvement in communication times, and from the summary statistics provided, there was an improvement of approximately 2.0s for this process by using the US protocol.

4.4 Shared Memory

For the previous two studies, the limitations of the tracefile data gave little profiling information outside of the MPI calls. Part of the motivation for inclusion of dynamic instrumentation in PPerfDB is to provide data unavailable through existing sources. In this particular example, I have examined the use of shared memory on communication from an application level standpoint. Again SMG98 was run twice, both runs done with 8 processors, a topology of 2x2x2, and an problem size of 40x40x40 per processor. For the first run, shared memory was used for inter-node communication, and for the second, this communication was routed over the network.



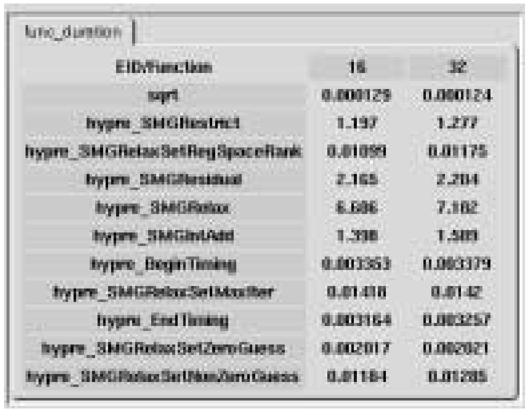
func_duration

EID/Function	16	32
HYPRE_SetStructGridExtents	0.0004366	0.0003531
HYPRE_SetStructMatrixBoxValues	0.0004199	0.0003715
HYPRE_SetStructVectorBoxValues	0.007545	0.007902
HYPRE_StructSMGGetNumIterations	1.175e-05	1.212e-05
HYPRE_SetStructStencilElement	6.875e-06	1.063e-05
HYPRE_SetStructMatrixNumGhost	0.0001953	0.000233
MPI_Comm_size	2.4e-05	2.425e-05
MPI_Finalize	3.205	3.158
HYPRE_NewStructStencil	0.0001651	0.0002895
printf	0.000295	2.3e-05
HYPRE_NewStructMatrix	0.0002215	0.0002356
HYPRE_InitializeStructMatrix	0.03506	0.03491
HYPRE_AssembleStructMatrix	0.02747	0.02252
HYPRE_StructSMGGetFinalRelativeResidualNorm	1.075e-05	1.063e-05
HYPRE_FreeStructMatrix	0.0004627	0.0004629
MPI_Init	1.717	1.624
HYPRE_SetStructMatrixSymmetric	1e-05	9.75e-06
HYPRE_NewStructGrid	0.0006151	0.000442
HYPRE_AssembleStructGrid	0.002122	0.002657
hypre_free	1.112e-05	1.063e-05
HYPRE_FreeStructGrid	8.012e-05	7.787e-05
MPI_Comm_rank	1.612e-05	1.6e-05
HYPRE_StructSMGSetRelChange	1.612e-05	1.563e-05
HYPRE_StructSMGSetup	6.455	7.23
HYPRE_StructSMGSolve	32.43	35.55
HYPRE_NewStructVector	2.8e-05	2.95e-05
HYPRE_InitializeStructVector	0.0056	0.005322
HYPRE_AssembleStructVector	1.225e-05	1.138e-05
HYPRE_FreeStructVector	2.025e-05	2.038e-05

Figure 15: Profile of `main` for Shared Memory and Network Inter-node Communication

This screenshot shows the times in seconds for each indicated function. The column headings indicated the EIDs of the execution, shared memory was used for EID 16 and not for EID 32.

Instrumentation was inserted into a single process in the computation to obtain functional cost information for all of the functions called from `main`, and the function in which most of the computation and communication was to occur during the solve phase, `hypre_SMGSolve`. The results for functions consuming more than a micro-



EID/Function	16	32
sqrt	0.000129	0.000124
hypre_SMGRestrict	1.197	1.277
hypre_SMGRelaxSetRegSpaceRank	0.01099	0.01175
hypre_SMGResidual	2.165	2.204
hypre_SMGRelax	6.686	7.182
hypre_SMGIntAdd	1.398	1.509
hypre_BeginTiming	0.003353	0.003379
hypre_SMGRelaxSetMaxIter	0.01418	0.0142
hypre_EndTiming	0.003164	0.003257
hypre_SMGRelaxSetZeroGuess	0.002017	0.002021
hypre_SMGRelaxSetNonZeroGuess	0.01184	0.01285

Figure 16: Profile of `hypre_SMGSolve` for Shared Memory and Network Inter-node Communication

second of the total execution time are tabulated in Figure 15 and Figure 16. As can be seen from the data, a modest gain of 0.77s was made in the problems setup phase (`HYPRE_StructSMGSetup`) and, more importantly, 3.12s during its solve phase (`HYPRE_StructSMGSolve`). As the `hypre_SMGSolve` is called from `HYPRE_StructSMGSolve` and comprises the main part of the solver, an examination of its results indicated that the benefit was distributed throughout the parts of the calculation.

5 RELATED WORK

Although there are relatively few performance tools that allow simultaneous comparisons between multiple data sets, there has been quite a bit of work on providing a unified representation of diverse tracefile formats.

As a self-described meta-format for tracefiles generated by various tracing libraries, SDDF [18] is the data source for the wide array of the Pablo [19] project's performance tools. This Self-Defining Data Format stores the syntactic structure of the trace events within the tracefile, thereby allowing a single C++ API for the extraction of different types of data from multiple converted formats. Visualization of SDDF data can be done through SvPablo [20], which also provides the means to graphically browse the source code, automatically or manually instrument the code, rerun the application, and then annotate the source with the count and duration information for each instrumented construct. SvPablo allows one to distinguish different runs of an application by execution environments, so that different runs in different "contexts" can be compared, but on a case by case basis.

As a language definition for tracefile analysis, EARL [21] also provides uniform access to several tracefile formats. It has four predefined event types corresponding to code region entry, region exit, message sends and message receives with associated attributes such as the time, the processing node, and the type. Implemented as an extension of the Tcl scripting language, it is possible to use the existing and extended features of this language to combine extracted events into more meaningful metrics, and then analyze the results. This is precisely what has been done with EXPERT [22], an extensible tool which searches for pre-defined behavioral patterns in parallel execution traces. EXPERT's design follows the APART [23] group's object-oriented specification of performance data, both static and dynamic, and of the well-known bottlenecks occurring in parallel code. While EARL allows programmers the ability to examine data from more than a single tracefile at a time, pursual of these types of investigations has not yet occurred.

Another tool for the visualization and analysis of tracefile data created by multiple libraries is MEDEA [25]. Modular in design, MEDEA uses a filtering module to extract tracefile information on the basis of a specified level of detail and a metric. Available metrics include the number of processors involved in execution, I/O, message transmission, message reception, and overall communication rates, and computation vs. execution times. Once filtered, basic statistical analysis of the performance data can be achieved with the clustering module. Speculation into application's behavior for instances for which tracefile data does not exists is done through a fitting module which attempts to match a curve to the available data. Finally, the visualization module graphs the results of filtering, clustering, and fitting modules. MEDEA provides the user the ability to save previous analyses from different executions under the heading of a single session. When a session contains similar metrics from two or more executions, MEDEA can further derive metrics of application speedup, efficiency, efficacy, and total execution time vs. number of processors.

The current tool differs from these approaches in terms of generality. While EARL and MEDEA provide functionality similar to components of PPerfDB, they do not integrate this into an environment which can analyze tracefile data, dynamic instrumentation data, and other types of profiling reports. This is also true of tools using dynamic instrumentation, of which there are a couple with similar functionality.

The Paradyn [7] performance tool represents a very thorough exploration of the features available from dynamic instrumentation. Paradyn implements a wide assortment of metrics providing detailed timing information and usage statistics, and a Metric Definition Language for the creation of more. Various visualization modules provide runtime feedback for on-line performance monitoring and an automatic data folding technique keeps data collection within bounds for a high degree of scalability. The Performance Consultant component of Paradyn performs automatic searches for bottlenecks by inserting instrumentation at continually more refined points in the program as hypothesis about potential bottlenecks are accepted or rejected.

Tool Gear [26], currently under development at Lawrence Livermore National Laboratory, functions as in intermediary between various performance tools, offering a database of stored performance results and the ability to view this and newly measured data via a source code browser. With underlying support for code instrumentation provided by DPCL and the PAPI library for accessing hardware performance counters, it is possible to measure function calls and cache utilization.

Finally, comparative studies between two running parallel applications have been undergone with the Guard [27] debugger. It provides two directives for correctness checking, an ‘assert’ statement which performs some operation on the basis of a conditional expression between two equivalent data structures in both programs as they are running, and ‘compare’ statement which can be used to examine data values once the application is paused. Relative debugging differs from the current study in that it is concerned with program correctness, rather than performance.

6 CONCLUSION AND FUTURE WORK

In this study, I endeavored to provide access to performance data stored in a variety of formats and to present this data in an analytical environment which would allow the identification and quantification of adjustments made to the system and code of parallel applications. Additionally, I wanted to provide the framework necessary for further exploration of application behavior under novel circumstances, the definition of which would be perhaps motivated by previous analysis. As a result, I have added to PPerfDB support for data extraction from three common types of tracefiles, the means to generate new data using dynamic instrumentation, and the ability to access this data through a uniform interface. Using this tool in the context of several performance studies, the current prototype of PPerfDB has shown that it can successfully help locate and identify performance perturbations due to code and system changes. It has also shown that it can be used for experiment definition and hypothesis testing.

However, to both broaden and deepen the scope of the kinds of studies that can be done in this framework, there remain several places that increased functionality would be helpful. These include:

- Increasing the range of the dynamic instrumentation to include system level information, such as the kernel statistics provided in the VT tracefiles and the data available from hardware performance counters.

- Finer granularity to the data available through dynamic instrumentation, so that instrumentation is not limited to function level measurements, but code blocks and lines as well.
- A broader range of automated methods for data analysis and performance diagnosis. This would include the ability to summarize results and their differences for multiple foci and multiple metrics at once, and to be able to derive application level metrics, such as application speedup, efficiency, network bandwidth, and communication vs. computation rates.
- Extended visualization techniques that would allow a greater breadth of summary type information to be presented at once, including the application specific metrics previously mentioned.

7 REFERENCES

[1] J. C. Yan. "Performance Tuning with AIMS - An Automated Instrumentation and Monitoring System for Multicomputers”. *Proceedings of the 27th Hawaii International Conference on System Sciences*, January, 1994. Vol. II. pp. 625-633.

[2] O. Zaki, E. Lusk, W. Gropp, and D. Swider. “Toward scalable performance visualization with Jumpshot”. *High Performance Computing Applications*, 13(2):277--288, Fall 1999.

[3] M.T. Heath and J.E. Finger. “Paragraph: A Performance Visualization Tool for MPI”. http://www.csar.uiuc.edu/software/paragraph.

[4] P. H. Worley. “A New PICL Trace File Format”. Technical Report ORNL/TM-12125, Oak Ridge National Laboratory, Oak Ridge, TN, October 1992.

[5] Pallas GmbH. *Vampir 2.0 User’s Manual*, Pallas GmbH, Bruhl, Germany, June 1999.

[6] IBM Corporation. *IBM Parallel Environment for AIX, Operation and Use, Volume 2*. IBM Corporation, Poughkeepsie, NY, 2000.

[7] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall, “The Paradyn Parallel Performance Measurement Tools”. *IEEE Computer 28*, 11, November 1995.

[8] J.K. Hollingsworth, B.P. Miller, and J. Cargille. “Dynamic Program Instrumentation for Scalable Performance Tools”. *1994 Scalable High-Performance Computing Conference, Knoxvill*e, TN, pp. 841-850.

[9] B. Buck and J.K. Hollingsworth. “An API for Runtime Code Patching”. *Journal of Supercomputing Applications*, 2000.

[10] IBM Corporation. *IBM Parallel Environment for AIX: Dynamic Probe Class Library Programming Guide.* IBM Corporation, Poughkeepsie, NY, 2000.

[11] K.L. Karavanic. “Experiment Management Support for Parallel Performance Tuning”. PhD Dissertation, University of Wisconsin, Department of Computer Science, Madison, WI, 1999.

[12] A. Brighton. “The Tree Widget”. http://archive.eso.org/~abrighto/tree/tree.html, September, 1998.

[13] K.L. Karavanic and B.P. Miller. “Improving Online Performance Diagnosis by the Use of Historical Performance Data”. Proceedings of the 1999 Conference on Supercomputing ‘99, Portland, OR, November 1999.

[14] W. Gropp and E. Lusk. “User’s Guide for MPE: Extensions for MPI Programs”. http://www-unix.mcs.anl.gov/mpi/mpich.

[15] Compaq Computer Corporation. *Compaq MPI: User Guide*. Compaq Computer Corporation, Houston, TX, July 1999.

[16] Pallas GmbH. *Vampirtrace 2.0 Installation and User’s Guide*. Pallas GmbH, Bruhl, Germany, November 1999.

[17] P.N. Brown, R.D. Falgout, and J.E. Jones. “Semicoarsening Multigrid on Distributed Memory Machines”. To appear in SIAM Journal on Scientific Computing.

[18] R. A. Aydt. “The Pablo Self-Defining Data Format”. University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, IL, 1997..

[19] D.A. Reed, P.C. Roth, R.A. Aydt, K.A. Shields, L.F. Tavera, R.J. Noe, and B.W. Schwartz. “Scalable Performance Analysis: The Pablo Performance Analysis Environment”. *Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society*, pp. 104-113, October 1993.

[20] L.A. DeRose and D.A. Reed. “SvPablo: A Multi-Language Architectue-Independent Performance Analysis System”. *10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools’98*, pp. 352-355, Palma de Mallorca, Spain, September 1998.

[21] F. Wolf and B. Mohr. “EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs”. Forschungszentrum Jülich Technical Report FZJ-ZAM-IB-9803, April 1998.

[22] F. Wolf and B. Mohr. “Automantic Performance Analysis of MPI Applications Based on Event Traces”. *Proceedings of the European Conference on Parallel Computing, Munchen, Germany*, September, 2000.

[23] T. Fahringer, M. Gerndt, G. Riley, J.L. Traff. “Knowledge Specification for Automatic Performance Analysis - APART Technical Report”. Technical Report FZJ-ZAM-IB-9918, November 1999.

[24] L. Massari, A. Merlo, and D. Tessera. “MEDEA - Measurements Description Evaluation and Analysis Tool - User’s Guide”. Rapporto Tecnico Dipartimento di Informatica e Sistemistica n. 103/96, 1996.

[25] M. Calzarossa, L. Massari, A. Merlo and D. Tessera. “Parallel Performance Evaluation: the MEDEA Tool.” I.H. Liddel, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *High-Performance Computing and Networking, Volume 1067 of Lecture Notes in Computer Science*, pp. 522-529, Springer-Verlag, 1996.

[26] J. Gyllenhaal and J. May. “Tool Gear”, http://www.llnl.gov/asci/pse/asde/toolgear.html, February 2001.

[27] D. Abramson and G. Watson. "Relative Debugging for Parallel Systems". *Proceedings of PCW 97*, Canberra, Australia, September 1997.