2005

Dynamic multi-resource monitoring for predictive job scheduling.

Lun Liu
University of Windsor

Dynamic Multi-Resource Monitoring for Predictive Job Scheduling

by

Lun Liu

A Thesis
Submitted to the Faculty of Graduate Studied and Research
Through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2005

Your file Votre référence
ISBN: 0-494-09803-1
Our file Notre référence
ISBN: 0-494-09803-1

Abstract

Standard job schedulers rely on either the user's estimation, or a few approaches that use performance databases to keep information about job runtimes to predict future runs. Co-scheduling for improved resource utilization, however, requires more detailed information as regards behavior on multiple resources to make predictions about slowdowns. Thus, information about communication, I/O, and computation at application level is needed but hard to estimate by the user. Furthermore, dynamic adaptive resource allocation requires information about the different processes on different machine nodes.

We present an intelligent monitoring tool, ScoPro, which provides such information. To make monitoring more feasible, ScoPro harnesses the dynamic instrument techniques, which postpone insertion of instrumentation code until the application is executing. To keep intrusion low, we limit monitoring to short test phases.

Tests demonstrated that ScoPro can monitor certain function groups (such as I/O and communications) from multiple parallel applications simultaneously, and collect metrics such as computation time per loop, application-level communication time and communication/calculation ratio, communication volumes, and applications' progresses during a short test phase with minor hinting from user. The comparing test shows application-level metrics acquired within a few iteration steps is acceptable close to the results through the whole application. Our test also shows that relating the progress data of co-scheduled job can lead to a more accurate running time prediction.

Acknowledgement

I feel happy to take this opportunity to thank those people who had helped me significantly during the whole process of my graduate study, and towards the completion of my thesis.

First, I would like to give my deep appreciation to my supervisor, Dr. Sodan. The work could not be accomplished without her extensive guidance and persistence.

Secondly, I would like to thank all my friends and colleagues, who shared with me their precious time to discuss academic issues, provide useful suggestions, and worthwhile experiences during the research.

Thirdly, I would like to give my special thanks to brothers and their families, for all kinds of help they have given to me.

I am also very grateful to the committee members, for their valuable advices and revision.

Thanks, my mom and dad, for the persistent and deep love to me. This paper is dedicated to them.

Table of Contents

1. Introduction

Metacomputing is a high performance computational platform, formed by combining various computing resources together through a network. In searching ways to provide instant and accurate application monitoring data for the scheduler of cluster (the homogeneous structured meta-computing system) while keeping monitoring overhead low and under control, we present a tool, ScoPro, which harnesses dynamic instrument technique for parallel application monitoring, which can dynamically insert and remove instrument code while the parallel application is running. When application is running in un-instrumented mode, there is almost no overhead asserted.

Upon the instrument components, we build a mechanism which effectively collects and relates datum from multiple parallel applications.

Providing effective data for the scheduler is another focus of us, we use more flexible ways to dynamically instrument data of parallel applications. The data provided by ScoPro is able to reveal the following:

- Application level Communication / calculation ratio of the parallel application.
- Dynamically insert/remove the instrumentation code.
- Heterogeneous nodes characteristics where the parallel applications are running.
- Intrinsic behaviors of parallel applications including the communication volumes.
- The progresses of whole applications in certain environment.
- The related information of multi-applications for more accurate running time prediction of co-scheduling jobs.

2. Background review

2.1 The monitoring of parallel computation system

High efficient meta-computing systems rely on accurate and instant information of both parallel applications and their running environment gathered through resource monitoring systems and application monitoring systems. The resource monitoring systems can provide fluctuating status of various resources of the system including CPU, memory, IO and network links etc. The application monitoring systems instrument the status of the running parallel applications, acquiring the quantity and efficiency of using these resources.

In general, the main purposes of resource and application monitoring include the following:

1). To provide information for dynamic resource/task match (scheduling/check pointing)

Any meta-computing system must include a scheduler, either human or automatic, the goal of which is to select the most-appropriate resources, such as hosts, network links, disk storage, etc. that are going to be used by an application. The scheduler must choose dynamically the best resources according to resource characteristics at the moment. Because resource monitoring can dynamically provide information about the variation of the performance of grid resources, it became essential for the schedulers.

As stated in [12], "Dynamically Forecasting Network Performance Using the initial scheduling results using the NWS are promising"

2). Performance alarm and fault tolerance

Some resource monitoring systems such as NWSALARM [11] have a pre-set performance threshold. When hardware fails or large performance degradation happens in a heterogeneous grid computing environments, the resource monitoring system can inform other grid components or the user to take corrective actions.

In some other more advanced implementations such as those studied in [13], they use fuzzy logic to analyze a set of the datum acquired by contract monitors and determine if the contract of performance is violated.

3). Adapt application behavior to improve performance

In a cluster and grid environment, not only are the characteristic resources' demands of applications very variable, but also the resource performance fluctuates significantly. Some monitoring systems, by monitoring these two aspects simultaneously, can dynamically choose the access pattern to the resources through an actuator, improving the performance of parallel applications. Such an example is Pablo [9], in which Dr. Reed and Vetter first introduced the concept of "resource policy actuator"

4). Online and offline performance analysis and visualization

The monitored data, acquired by the sensor or probe, can be collected by an agent and sent to the client side, so that the user can conduct an analysis of the performance. For example, he can find the bottleneck of grid environment in running the applications. The data collected then can be visualized in real time or can be visualized in a post-mortem way.

5). Improve accuracy of prediction of parallel applications

The parallel applications are sensitive to the performance of the computation and communication resources to a greater or lesser extent, as studied in papers [1] and [16]. The execution time of an application can be more accurately predicted, by dynamically monitoring the performance of resources accessed by it.

2.2 The parallel application monitoring

While a resource monitoring system is critical for large heterogeneous meta-computing system to provide real-time resource information, parallel application monitoring is also very important, mainly for the following two reasons:

Knowing the characteristics of the application helps the scheduler to find the more efficient resource for this application.

Parallel application is not a passive object in a meta-computing system; it also actively affects the status of resources and other parallel applications.

A common way to implement application performance monitoring is by inserting a piece of instrumentation code into specified places of the source code of the applications either manually or automatically before compilation (e.g. [8]).

There are four kinds of performance instrumentation techniques of parallel application: timing, sampling, counting, and event tracing, which will be briefly described below.

Timing means the measurement of aggregate execution time. Timing can

reveal the approximate performance bottleneck, but cannot tell the exact time of it and the component responsible for it. To implement a timing facility, one needs only low latency access to a clock whose resolution is high compared to the elapsed time of events being measured.

Counting records the number of times an event occurs but not when and where. Having the total time and count, one can accurately calculate the average execution times. Counting is efficient, low intrusive, and produces very limited amount of data.

Sampling is accomplished by periodically observing the system state and incrementing the counter corresponding to the observed state. An example of sampling [2] is using the timer interrupt service routine (ISR) that logs the instruction pointer of the interrupted instruction. The distribution of the instruction pointers indicates where the program spends most of its time.

The event tracing is the most intrusive method because it generates a detailed record of each event occurred. The information acquired by event tracing can include the following:

1. What action occurred.
2. The time when the event occurred.
3. The location of where the event occurred.
4. Any additional data that defines the event circumstances.

2.3 The dynamic instrumentation of application

The normal cycle of developing a program is to edit source code, compile it, and then execute the generated binary. Dynamic instrumentation can modify the generated executable and redirect the execution from certain points to

code snippet generated by third party. Thus, there is no requirement for users to submit source code to accomplish it.

In dynamic instrumentation, the program that finds the inserting point in the application's image and modifies it is called mutator; the application to be instrumented, of which the executable image is to be modified, is called mutatee.

The two primary abstractions are points and snippets. A point is a location in a program where instrumentation can be inserted. A snippet is a representation of a bit of executable code to be inserted into a program at a point. Snippet usually includes simple operations that change the value of a counter or a timer. Because this feature of the dynamic instrumentation, it is language independent but could be platform dependent.

A typical procedure of the dynamic instrumentation is listed in the following:

- Load the image of executable into the buffer and stop the application.
- Find the instrument points.
- Generate the instrument code and insert the instrument code.
- Run the application.

2.4 The Dyninst_API

The dyninst_api [14] is a set of Application Program Interfaces (API) developed by Dr. Bryan Buck of University of Maryland for implementing dynamic instrumentation under Linux, Solaris and WinNT environments.

The unique feature of this interface is that it makes it possible to insert and

change instrumentation in a running program. This differs from other post-linker instrumentation tools that permit code to be inserted into a binary before it starts to execute.

Using Dyninst_API, a mutator must create a single instance of the class BPatch. This object is used to access functions and information that are global to the library.

The first thing a mutator needs to do is to identify the application process to be modified by specifying the executable file name and process id. If the application has not yet started, it must provide executable file name and the arguments of the applications.

Once the application thread has been created, the mutator defines the snippet to be inserted and the points where they should be inserted.

Bpatch_image class stands for the image of the program, which could be acquired from the instance of Bpatch.

After the acquiring the image handle of the program, the next step will be to find the point in the image where the snippet could be inserted.

The points in dyninst_API could be entry points, exit points, call-site entry points and call-site exit point of functions, basic running block and even outer loops in the mutatee. However, finding points of functions of is the easiest way because there is a function name associated with certain points. So the list matching the search results will be largely narrowed down. In any case, it will return a list of matching points.

After acquiring the matching points, the next step is to generate the snippet

using dyninst_API. Although statements in snippet can be generated one by one using Dyninst_API, it is highly complicated and of low efficiency in this way. A more acceptable way is to generate a piece of code in a function and compile it into a shared library. At the run time, a mutator can dynamically find the function contained inside the shared library prepared previously, and insert a function-call statement to this function as a snippet into the mutatee.

After inserting the snippet, the mutator can start running the mutatee. Dyninst_API also support some other functions, including the stop/restart of the running process, listening of the termination of mutatee etc.

2.5 Other tools using dyninst_API

2.5.1 Introduction to Paradyn

Paradyn [4] is a performance measurement tool for parallel and distributed programs. Paradyn uses several novel technologies so that it scales to long-running programs (hours or days) and large (thousand nodes) systems, and automates much of the search for performance bottlenecks. It can provide precise performance data down to the procedure and statement level.

In addition, Paradyn provides a tool for the automatic isolation of performance bottlenecks and an open visualization interface, which is implemented with a W3 search model trying to answer three separate questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur.

In addition, several performance visualizations are provided.

In Paradyn, monitoring data can be constantly and periodically transferred

to a visualizer in real time. Periodic sampling of these structures provides accurate information about the time varying performance of an application without requiring the large amount of data needed by full tracing.

2.5.2 Introduction of Dynaprof

The most well known application monitoring tools using dyninst_API is paradyn. However, our work is based on another tool, Dynaprof [5, 6]. It is developed by Dr. Mucci of University of Tennessee, regarded as "A portable tool to dynamically instrument serial and parallel programs for the purpose of performance analysis."

Dynaprof provides a simple and intuitive command line interface like GDB. It also provides visualizers using java/Swing GUI. Instrumentation of Dynaprof is done through the run-time insertion of function calls to specially developed performance probes.

Dynaprof provides 3 kinds of sensors, including the CPU counter sensor, the wallclock sensor, and the specified sensor for coupling the probes and the visualizers. The wallclock sensor records the total execution time of a specified function and count the number of times a measured function is called.

However, the instrumentation data Dynaprof is saved to a local file only after the parallel application (mutatee) finishes. So strictly it is a post-mortem analysis tool.

3. The motivation of our approach

Current dynamic instrumentation tools mainly focus on performance trouble-shooting of single parallel application. Other cluster/meta-computing

application monitoring tools [9] [29] have been introduced in other papers; however, they did not harness dynamic instrumentation method.

Moreover, there are several obvious advantages of dynamic instrumentation:

- No source code modification; dynamic instrumentation makes this more realizable and furthermore, the user can keep the privacy of their source code.
- Can dynamically instrument and un-instrument the monitoring code; there is no overhead asserted to the application when it is running in the un-instrument mode, which also enables us to shortly measure several loops and predict the remainder.

Because of the advantages of dynamic instrumentation, we believe it is feasible to apply the dynamic instrumentation method for parallel applications monitoring. Our work focuses on verifying this feasibility and on doing some initial studies on what kind of useful information could be acquired and provided to the scheduler for the purpose of better resources' utilization, which we will give a detailed description about this in the later chapters.

4. Our goals

Firstly ScoPro should be able to simultaneously monitor multiple parallel applications using dynamic instrumentation and acquire the resource related characteristics of parallel applications through the dynamic instrumentation, and provide the acquired information to other modules for the purpose of resource usage optimization.

Secondly the data acquired by ScoPro should be able to demonstrate the following:

- Application-level Communication/calculation ratio of parallel application.
- The characteristics of heterogeneous nodes where the parallel applications are running.
- Intrinsic behaviors of parallel applications including the communication volumes.
- The progresses of the whole applications in certain environments.
- The related information of multi-applications for more accurate running time prediction of co-scheduling jobs
- Intrinsic behaviors of parallel applications including the communication volumes.

The monitoring data should be acquired by shortly inserted and triggered measurement using dynamic instrumentation, and we should verify the effectiveness of the data in better resource-task allocation and better prediction of resource usage (how long and the intensity) in our work.

5. The functionality and extension of ScoPro

While Paradyn and Dynaprof are mainly performance bottleneck shooting tools for parallel applications, we hope to harness dynamic instrumentation for monitoring the resource access behavior of all the parallel applications in meta-computing system.

Because of this different orientation with Dynaprof and Paradyn, ScoPro provides a mechanism to instrument multiple applications at the same time, more methods to reduce or control overhead, more flexible ways to instrument in acquiring resource access behavior of parallel application.

In monitoring the resource accessing behavior, we are more concerned

with the capability of ScoPro in comparing the difference of different processes of the same application, in comparing different applications running in the same environments, and in comparing the performance data of a parallel application with its historic data running in a different context.

5.1 The main extension of ScoPro to Dynaprof

The main functionality extension of ScoPro to the Dynaprof includes the following:

1) Can dynamically instrument and un-instrument the parallel application; when in the un-instrument mode, there is no overhead asserted.

2) The measurement can be triggered by certain external events (e.g. the arrival of a certain new job which may affect remarkably the running environments).

3) Can monitor multiple parallel applications at the same time. Data from different parallel application are collected and combined by the controller, being enabled to monitor and analyze data from different nodes and different applications.

4) Can implement some complex logic, i.e. the measurement could be triggered to start or stop when one function is called a certain number of times.

5) Can acquire absolute timestamp value of function calls.

6) Can acquire the parameter values of the monitored function calls.

7) Can support mpich [18] applications running on ch_gm devices, which have higher performance for communication compared with ch_p4.

8) Overhead can be controlled by several ways: monitoring can be stopped and started in a more flexible and controlled way. We can set a time-limits and number-of-calls limit option as a condition for start and stop measurements.

9) The monitor application is running in event blocking wait status, instead of using a poll (provided by dyninstAPI) to respond the end of mutatee or other events.

10) Use shared memory to buffer monitoring date. And the data is transferred after parallel application switch from instrumented mode to un-instrumented mode.

11) The complex instrumentation condition enables us to instrument data right within the loop, i.e. start at the beginning of the loops and end at the beginning the loops also, which enable us to take the measurements of one or several whole loops without approximation.

However, Dynaprof currently supports instrumentation of hardware counter that provides very useful CPU related metrics from the monitored applications, which we have not yet integrated into our work.

5.2 Main difference from Paradyn

1) Paradyn is a performance analysis tool, targeting the monitoring of one application and finding the performance bottleneck of a specified parallel application while ScoPro is a performance-monitoring tool, providing the

resource related performance information of multiple parallel applications.

2) Monitored data of ScoPro is packaged and transferred when the monitored application switches to run in un-instrumented mode. (Experiment in [27] shows that monitoring-and-forwarding is much more expensive than the batching-and-forwarding in which the data is first buffered at the local site and data transfer happens less frequently).

3) ScoPro has more ways to reduce or control overhead, more flexible ways to instrument in acquiring resource access behavior of parallel applications as stated in 4.1.

6. The environment and implementation of ScoPro

6.1 The Overall Tool Environment for ScoPro

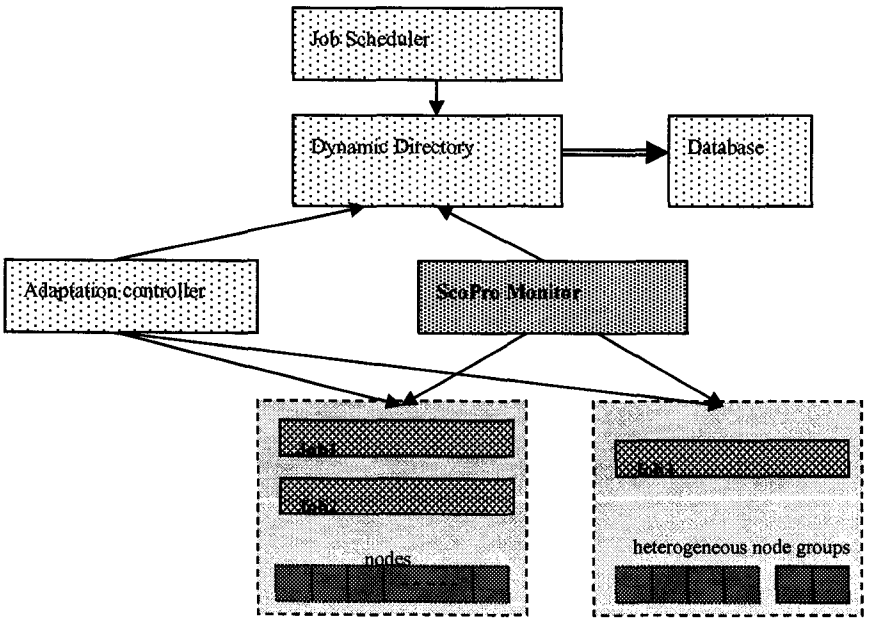


Figure 1. The Context of our monitoring environment with job scheduler, dynamic directory, and adaptation control. Copied from [30].

As we explained in chapter 4, we envision employment of ScoPro to obtain detailed application characteristics for the purpose of optimization of resource usage. The architecture of overall environment for ScoPro is shown in Fig1, The job scheduler will perform adaptive space allocation [21] and/or coscheduling with approaches like LOMARC [22]. The dynamic directory [23] will maintain the data extracted by monitoring as long as the job is in the system. Long-term information about program runs will be stored in the database, permitting historical evaluation.

6.2 The implementation of ScoPro

As shown in Fig2, ScoPro, a centric structured dynamic monitoring system, can simultaneously monitor multiple parallel applications, each of which can have multiple processes running on different sites (nodes).

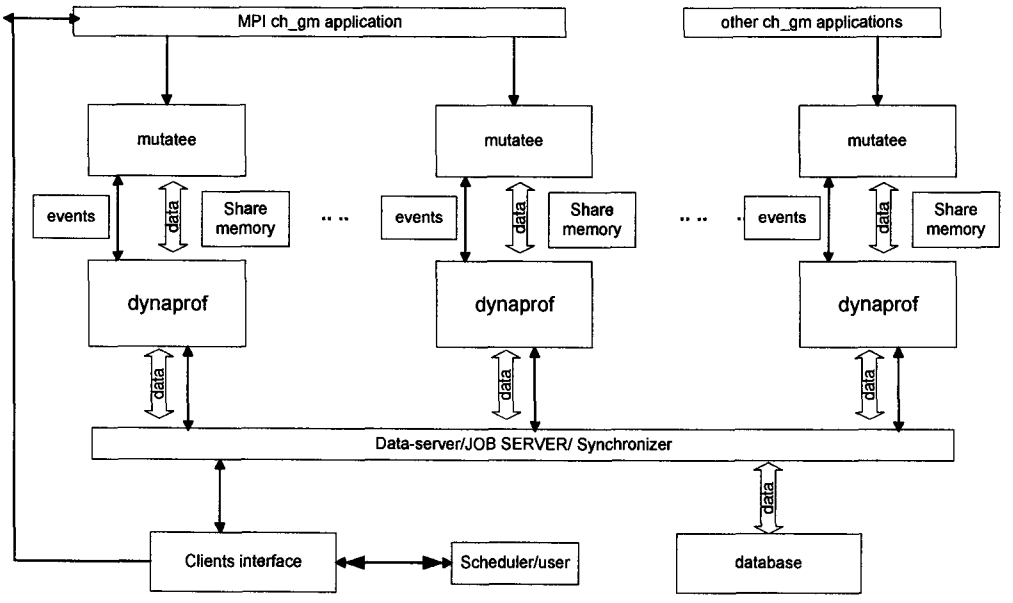


Fig2. The diagram of ScoPro

First, the scheduler (could be a user) sends a request to start a monitored parallel application by calling the client interface. The client interface will forward the request to the controller (Data-server/job-server/synchronizer) with

related information including a list of measured functions, a list of nodes, the information of monitored application, and the user's setting about measurement etc.

Note: To implement this step we did minor modifications for "mpirun", a start command for MPI [18] jobs, making it also accept a job ID as a parameter. An example of calling this script file would be "mpirun –np 4 –jb 2004121000007 dynaprof" from the client interface. The dynaprof(sensor) will contact the controller, getting the other information buffered in the session related to this job.

The controller, after acquiring the information, will create a job session and a job id with it, and send the id of this job back to the client interface.

The client interface, after acquiring the job-id, will start the parallel application. If it is an mpi-ch_gm application, it will start the MPI ch_gm daemon with dynaprof as a parameter in remote execution mode. The MPI ch_gm daemon will start dynaprof on different nodes with the MPI application name as a parameter and other related information as environment variable including the job id and the ch_gm magic id.

The started Dynaprof will finish the following steps one by one:

1). Set up communication with the controller and acquire the list of instrumented function description.

2). Claim a certain size of shared memory according to the number of measured metrics and other requirements.

3). Load the application executable in stopped status (mpi application stopped at the end of Mpi_init).

4). Make synchronization through controller to make sure every node has successfully initialized (including MPI initialization).

5). Find the instrumented point and insert instrumented code according to the list of measured functions. Generate the snippets and insert into both entry and exit point of instrumented functions.

6). Run one time code in the mutatee's space, including the following task, attaching shared memory, setting monitoring global variables etc.

7). Execute the mutatee.

After mutatee is running, the dynaprof will wait for two events: signal from mutatee, time out event.

When parallel application is running in monitored mode, it will save the monitoring data directly into shared memory. The monitoring data includes the following: the time total/detailed time spent in running different functions, the absolute time for entry of functions, the communication/IO volume, and number of calls of a certain function in a certain period of time etc.

If certain conditions defined by the user become true (the times a certain function is called reaches a predefined value), the mutatee will send a signal to dynaprof. Dynaprof will then stop the mutatee and remove the instrument code from the parallel application and continue the processes. From this moment on, the parallel application will run in un-instrumented mode.

Once the parallel application switches to run in un-instrumented mode, the Dynaprof will package the data saved in shared memory and transfer the data to the controller, which will then save the data to the database. Dynaprof will then block and wait for 3 events, including the finishing of application, the instrument request from the controller, and the timeout event.

The application can switch back to instrumented mode again whenever necessary. When the user sends a request to the controller for instrumentation,

the controller will send a signal to the corresponding Dynaprof process. The Dynaprof will stop the running process and insert the instrumented code into the application. The instrumented function list could differ each time when instrumentation starts, but must be a subset of initial instrumented function list, because we do not re-calculate the shared memory size once it is declared.

The controller in ScoPro provides three services, as shown in the following:

1) Listening for requests from users (via the user interface) for starting a job, or measuring. Once a job is created, a session related with that job will be buffered at the server side. On receiving a measuring request, it will forward this request to the corresponding process.
2) Provide synchronization service for different processes to ensure every process has properly initiated.
3) Data-collection service, the data from the dynaprof will be collected and combined with the information in the session of this job, and saved to the database as an integrated set of datum.

Note: If the job has no contact to the controller/server in a reasonable time, its session of this job will be removed.

7. API of ScoPro

The client interface of ScoPro provides 2 interfaces.

The First interface, called "MpiJobStart", starts an mpi job and instrumentation. There are four parameters included in this parameter:

1) MpiJobDesc: description of job and measurement, including executable name, path, running nodes number, location.

2) Confirmstruct: A monitoring handle for this job, including whether this job

is successfully started, a unique job id related to this job.

3) ifblocking: indicate if this function will return immediately once job is created or return after the monitored data has been acquired.

4) metricHeader: include the max time period to instrument, maximum number of measured functions, max number of calls recorded. Start condition to take the measurements.

5) metricList: a list of metric description, each of metric description include function name, library name, which parameter to be summed, whether to acquire detailed value of instrumentation etc.

The second interface, "mpi_measure", starts a measurement when the job is running in un-instrumented mode, and contains the following 2 parameters:

1) Description of this measurement: including the Jobid (which job to be instrumented), measuring time, which subset to be instrument in the initialized list, the event triggering this measurements etc.

2) Indicate if this function will return immediately once the instrument request is sent to the application or return after the monitored data has been acquired.

8. What ScoPro can provide

8.1 Instrument communication volume and calculation/communication ratio

Many parallel applications, especially the simulation applications, are

featured by having an iteration to control the progress. Examples are particle simulation application and GEOFEM [3]; each step of the iterations in these applications stands for a certain time stamp. There are calculations and communications within each time-step (or iteration). As a performance benchmark, all packages of NAS benchmark have a main iteration also.

If the application uses blocking function calls to communicate, by testing one or just several iterations and recording the time in calculation and communications inside the iteration, we can predict communication/ calculation ratio of the whole application because the application shows similar characteristics in each iteration.

However, to accurately mark the start of each step in the main iteration, we need to insert a function call into the source code with a specified name (Support of loop instrumentation and intelligent targeting of main iteration will be a future extension of ScoPro. Currently, dyninst_API [28] supports the searching and instrument outer-loop within a specified function).

The following codes show how easy it is to insert such functions into a FORTRAN application. The definition of "measuremark" is saved in "measuremark.f " provided by ScoPro, and a user can employ it by linking this file. Thus, the only part to be modified in the source code is to insert "call marksuremark" statement once at the entry of main iteration.

```
        User Loop condition
        {
            call measuremark()
            user code
        }
...
#   the definition of measuremark
```

```
subroutine measuremark()
!     implicit              logical (a-z)
      return
      end subroutine measuremark
```

By measuring the absolute time of the function which is specially inserted into the start of the loop, we can get the time elapsed for finishing one or several loops. Assuming t1 is the first absolute time the function is called and t2 is the last absolute time the function is called, the C is the number of times the functions is called, Tc is the application level communication time we calculated before. The communication/calculation ratio would be:

Tc / (t2 - t1-Tc)

8.2 The estimation of system-level communication time from metrics acquired by ScoPro

1). The difference between the application-level metrics and system-level metrics.

ScoPro can acquire application-level metrics by measuring the time elapsed for MPI routines, which essentially indicates the impact of communication to the overall performance of tested applications instead of the actual time of data transfer, which are to be instrumented at system level.

For both blocking and non-blocking communication routines, the results from application level measurement could be very different from the system level results. The blocking routines that communicate each other might initiate at different times due to an unbalanced workload or environment, the communication routines called earlier will have to wait until all other routines are started also. Thus, the time elapsed for communicating functions will

include both waiting time and the time for transferring the data.

The non-blocking function returns immediately after letting the system level to accomplish the data transfer. Usually the application will have to synchronize at a later point to ensure the message is delivered. In one case, if the communication has not yet finished, the synchronization function will block wait. But otherwise, if synchronization happens after the communication finished, it is impossible to acquire the actual communication time.

However, the application level communication time measured for parallel application using non-blocking function calls is meaningful because it tells the extent to which the performance of application is affected by communication.

2). The estimation of system-level communication time

ScoPro can catch the parameter value of the communication function call. In MPICH for example, every function in MPICH (collective or point to point) will ultimately call one of the MPID_Sendcontig, MPID_ISendcontig, MPID_Recvcontig, MPID_IRecvcontig, the third parameter of these functions indicates the transfer size. By catching and accumulating these values, we can know how much data was transferred. Assuming we know the bandwidth of each link between the processors involved in calculation, this information, together with the knowledge of the bandwidth of each link, enable us to estimate how much communication time is spent at system level.

According to logPC model [1] (a model that extends the LogP [15] and LogGP [7] models to account for the impact of network contention), the time for transferring a message is equal to the following:

$$T=Ost + L + (B - 1) * G. \quad (1)$$

Here Ost is the time for the sender to initiate the message, L is the average time for the message header to travel through the network, B is the message length (in bytes) and G is the network "Gap" (in cycles per byte).

Because ScoPro can acquire how much data was transferred for each link , (e.g. node1 to node3), we will be able to estimate what percentage of time was spent for transferring in any specified link using this model, assuming we know the bandwidth of each link.

Also, by instrument and adding the communication volumes of different applications running on the same node, we can get the total communication load of that node, which is useful information for load balancing.

3). The issue for estimation of bandwidth for data-transfer from the application level timing result and communication volume acquired by ScoPro

Although, in the ideal blocking point-to-point communication situation, the relationship between time for the MPI function call and the data size to be transferred matches the logGP model. However, in the case of dealing with real applications, we are currently unable to give a common formula that makes an exact relation between the time elapsed for blocking MPI communication functions and the data volume transferred. Through our analysis, we found the following difficulties that need to be solved:

- The optimizations in MPI communications: collective MPI functions take different implementation approaches to maximize the performance

depending on the number of nodes related and the message size to be transferred.

- When Mpi_send /Mpi_recv pair start at different time, if the message to be involved is larger than the available buffer(16k or set by user in mpich), the function that start earlier will have to be blocked until the corresponding function starts also. While for smaller messages, the mpi_send will send the data to the buffer directly instead of waiting for the corresponding receiving function.

However, in order to accurately get the absolute communication bandwidth from the elapsed time of MPI collective communication call, it is necessary to build a set of knowledgebase, each of which corresponds exclusively to one specified communication function and take the number of nodes involved, the message size into consideration (However this functionality is not yet implemented by ScoPro, but could be a future extension). The following figures show the test results tested by [25] on a Cray T3E-512, indicating the relationship for function MPI_Alltoall (MPI_Scatter at right side's Figure) between communication bandwidth, number of processors(nodes) and message size.

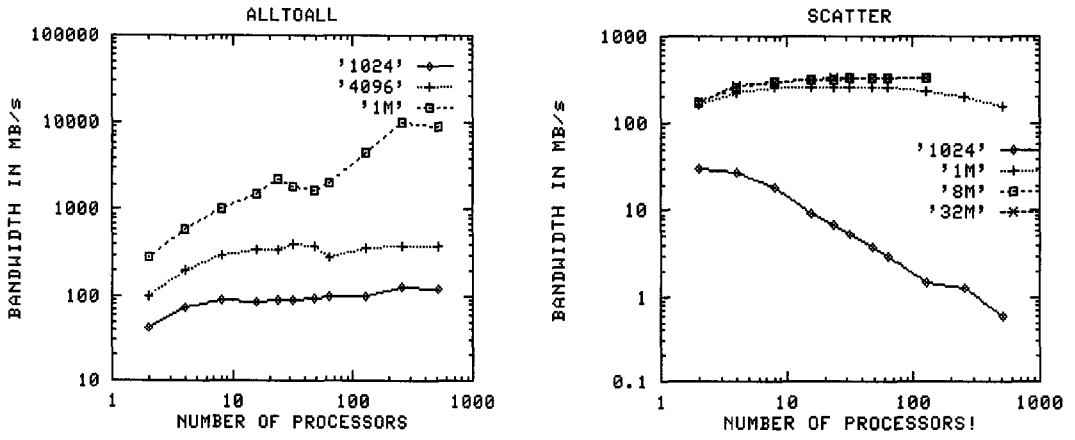


Figure 3: Bandwidth in MB/s for varying numbers of processors for an MPI_alltoall (left) operation and MPI_Scatter (right) operation. Three different message sizes are used, tested by

[25].

8.3 Acquire application run time/waiting time

Also, Application processes switch between CPU usage (busy) and non-usage (idle) phases during normal execution. ScoPro can acquire application busy-time/idle-time ratio in certain processes of the parallel application running on different nodes by measuring the blocking IO functions, communication functions and some other functions (e.g. sleep) in a certain period of time, which is meaningful to evaluate the calculation performance change due to the sharing of CPU resources.

In other words, the historic monitoring data of parallel applications which run without sharing CPU resources can be used to evaluate the performance potential when 2 different applications are co-scheduled. According to [16], when two applications share the same CPU, the "busytime" is the total of the busy time of the 2 applications; the "idletime" is the total of their idle time. In cases of "busytime" being less than "idletime", there will be no increase in the execution time. In cases of "busytime" greater then "idletime", the increase of the execution time is given in formula 2. Depending on this, the scheduler can assign application resources more reasonably and optimize the efficiency of CPU resource usage.

$$(\text{busytime}-\text{idletime}) / (\text{busytime}+ \text{idletime}) \qquad (2)$$

However, for parallel application, idletime is a variable which may be affected by the running progress of other nodes, CPU capability and communication link bandwidth, communication volume of other application etc. Thus, it is hard to make accurate predictions using this formula, but this information is useful for scheduling decision making.

8.4 Acquiring the heterogeneity characteristics where parallel applications run

Some parallel applications are symmetric, i.e. each process of such a parallel application have same amount of calculation workload. By measuring different processes of such kinds of applications, we can compare the capability of different nodes.

Parallel applications are characterized by intermittently synchronizing each other after a period of calculation. Nodes with more time to synchronize indicate that they are running faster, therefore they either have more calculation power or less workload to do than other nodes. If we are monitoring a symmetric parallel application, we can conclude the reason is the former one. If we have known that the environment is homogeneous, then the reason must be the latter one.

On the other hand, nodes that consistently have more synchronized communication time indicate the communication speed is affecting the running speed of application. In other words, we should avoid assigning 2 applications for which the running speed is largely affected by the communication. Such cases apply for both synchronous and asynchronous communications.

8.5 Instrument the progress of whole applications

Application monitoring data can be used to predict the performance of parallel application.

ScoPro can instrument the number of times a certain function is called in a certain time and the time spent when a certain function is called for certain

times. As a result, we can measure progress of a certain application running on different environments. We will know whether this application relatively runs faster or slower by monitoring the application in short period time and comparing the monitored data with its history record, also we could predict the running time of the whole process by using these data.

Related work is Prophesy [26], which emphasizes automatic performance analysis and modeling process, and use that model to predict the performance of the application under different system configuration. ScoPro, using dynamic instrumentation, can be applied to that infrastructure also. However, in the test we describe in 10.6, we use the time per iteration, a simplified but comprehensive indication of performance, to evaluate and predict the performance of parallel applications. Our extension also includes analysis of the relationship of multiple co-scheduled parallel applications in the context of co-scheduling approach [19], which provides better possibilities for resource utilization but also involve potential competition on resources, leading to slowdowns per individual application.

9. Overhead analysis

Overhead is unavoidable for any instrumentation systems. However, ScoPro uses several methods to control and reduce the overhead as shown in the following:

1). Measurement is taken only for a short period of time. In most other times there is almost no overhead asserted.

2). There is a maximum limit for the times of instrument code being called. When this limit is reached, the process will remove the instrumented code, run in un-instrumented mode.

3). Monitored data is transferred by dynaprof after application switch to run in un-instrumented mode.

4). Dynaprof blocking waits for events instead of using a poll for listening for the state change of mutatee.

5). Decease the data transfer size by buffering the job-related information at the server-side.

The main extra running time include:

Number of switch * (running time of remove instrumentation code + running time of insert instrumenting code) + instrumenting code time * times of function calls + slowndown factor.

The slowdown factor is because of the CPU activity of dynaprof when parallel application is running, but it is very small.

Other overhead includes the memory, network overhead, which is also ignorable, because data is summarized in ScoPro before transfer or saved to shared memory.

10. Experimental result acquired by ScoPro

We tested ScoPro on the Horus cluster which has 1 master node with 4 CPUs and 16 processing nodes. Each node of processing nodes from node1-node14 has one 2.0 GHz Xeon CPU while node 15-16 use 2.4 GHz CPU. For the coscheduling, we employ the fact that 2 applications can be run simultaneously (without process/thread switches) on a hyperthreaded CPU. This means we apply a special form of coscheduling that does not need any process switches [22]. Considering that the applications run simultaneously, they also issue communication simultaneously. The cluster has a Myrinet

interconnect, 512 Mbyte memory per node, and 512 Kbyte cache per CPU. We have used MPICH 1.2.5 with ch_gm device, i.e. MPICH-GM.

As test programs we have used a simple self-written particle simulation on a partitioned mesh with a 5-point stencil. The program uses nearest-neighbor communication (up to 4 sends and receives per iteration step, depending on the number of nodes employed and the position of the partition in the overall mesh). This program is very regular and loosely synchronous. The program is also very fine-grain, i.e. each iteration(simulation) step takes very short time. We have implemented both a blocking and a nonblocking version, with the latter having the potential to hide the communication latency.

Furthermore, we are using several of the NAS [17] benchmarks, Class B, including the Fast Fourier Transform (FT) benchmark, LU Decomposition (LU) Benchmark, Integer Sort (IS) Benchmark, Embarrassingly Parallel (EP) Benchmark, and Conjugate Gradient (CG) Benchmark. Each of these packages has different communication characteristics. LU package has only blocking point-to-point communication. FT package employs collective all-to-all communication, CG has a mixture of blocking and non-blocking communication, and EP is embarrassingly parallel. IS employs integer operations only whereas the other benchmarks involve floats. This is important for coscheduling on the hyperthreaded CPU as the two threads share the CPU resources.

10.1 The test for overhead of ScoPro

As we stated before the CPU overhead is mainly composed of the time to dynamically insert/remove instrumentation and the running time of measuring function call.

We have first measured the basic overhead introduced by profiling, which is implemented by measuring the extra running time of mutatee caused by profiling activity and dividing this value by number of times the measured functions is called.

The test result shows that the overhead for sampling is between 0.385 μsec and 0.5 μsec per monitored function call, depending on the complexity of the action taken. This overhead is low enough to potentially monitor a full application run if it is important to get detailed information of the overall program execution. The overhead is high enough to make it worthwhile to dynamically instrument and un-instrument the code if monitor information from short time windows is sufficient, especially considering that our goal is to monitor production-level code that may run for hours or days.

Our test result shows that dynamic instrumentation (placing the instrumentation) takes in the range of 0.22 sec, and un-instrumentation takes a similar amount of time. This was measured on 16 nodes for the blocking particle simulation.

The time is dependent on the number of nodes involved because the monitor processes have to be activated. The time, to a lesser extent, is also dependent on the number of functions to be monitored. The fact that distributed processes have to be activated leads to some skew in the reaction time which by itself accounts for approximately 0.15 sec out of the 0.22 sec. An important consequence of the skew is that the actual collection of monitor data should be delayed to start several iteration steps after the instrumentation has been inserted.

The overhead of dynamic removal and insertion of measuring-code is measured by time-stamping the start and end of it. The skew, however, is due to the different respond time from the controller to mutators(dynaprof) in various nodes plus the respond time when the mutator manipulates the mutatee, which we acquired by measuring the extra running time of the few iteration steps right before and after the time when dynamic removal or insertion of instrument code happens. A future possible extension is to use more efficient method to notify the mutator instead of using the expensive "rsh" call that we currently use.

10.2 Test for the accuracy of wall-time data produced by ScoPro

Application-level communication time stands for the impact of communication to the overall performance of applications as we stated in 8.2. To verify the correctness of this metrics acquired by ScoPro, we compared the results with the equivalent metrics acquired by MPI_Wtime [18] functions which were manually inserted into the source code of the tested applications.

particle simulation, 40,000 iterations	N_{nodes}	$T_{program}$	$T_{compute}$	T_{comm} real	% T_{comm} real	$\%T_{comm}$ measured by ScoPro	% T_{comm} error
particle simulation, blocking	4	125.90	107.63	18.27	14.51%	14.12%	2.7%
	16	38.10	23.65	14.45	37.90%	38.60%	1.9%
particle simulation, nonblocking	4	125.75	107.56	18.18	14.45%	14.12%	2.3%
	16	40.43	23.85	16.58	41.00%	40.07%	2.3%
FT	4	115.40	84.56	30.47	26.49%	25.87%	2.3%
	16	28.19	7.81	20.38	27.69%	28.23%	1.9%

Table1. Accuracy for measuring the full program run, using a simple particle simulation and the NAS benchmark FT. $T_{program}$ is overall runtime, $T_{compute}$ is computation time, T_{comm} is communication time, $\%T_{comm}$ is percentage of communication time, N_{nodes} the number of nodes employed.

The tested applications include the FT NAS benchmark, blocking particle simulation and non-blocking particle simulation.

As demonstrated in the table1, the accuracy errors are within the range of 3%.

Note: The reason for the communication time for the non-blocking version being higher is due to the fact that this version of MPICH does not actually exploit the latency hiding options but issues the communication with the wait instruction.

10.3 The test of calculation / communication ratio for blocking and non-blocking routines

The calculation/communication ratio tests include blocking and non-blocking particle simulation test case and NAS benchmark. Because both blocking simulation tested application and non-blocking simulation tested application have much shorter running time for each iteration and larger iteration numbers, we measured 100 iterations and 10 iterations respectively and compared the results with the results acquired by measuring the whole applications. Due to the issue of the skew, the measured data was begun to be recorded 500 iterations after the start of dynamic instrumentation. For NAS benchmark, the iteration for each is much longer and the whole application has much less iterations. Thus, we compare the test result of measuring 10 iterations and 5 iterations with the result of measuring the whole application. And measurement data for applications of NAS packages began to be recorded from 2 iterations after the start of dynamic insertion.

Application (N_{iter})	N_{nodes}	$T_{program}$	% T_{comm}, measured for full program	partial % T_{comm}, measured / % error	partial % T_{comm}, measured / % error vs. full	% T_{comm} error vs. real
				100 iterations	10 iterations	100 iterations
Particle simulation, blocking (40,000)	4	126.3 sec	14.12%	13.76% / 2.6%	14.68%/4.0%	5.3%
	16	38.6 sec	38.60%	39.03% / 1.1%	40.2%/4.9%	3.0%
Particle simulation, nonblocking (40,000)	4	126.8 sec	14.12%	13.69% / 3.0%	14.98%/6.1%	5.3%
	16	39.8 sec	40.07%	39.7% / 0.9%	40.72%/1.6%	3.2%
				10 iterations	5 iterations	10 iterations
LU (250)	4	577.2 sec	1.95%	1.98%/ 1.5%	1.87%/ 4.0%	
	16	142.2 sec	14.16%	14.95% / 5.5%	15.00% / 5.8%	
FT (20)	4	115.4 sec	25.80%	25.40% / 1.6%	24.88% / 3.9%	6.2%
	16	27.41 sec	28.23%	28.53% / 1.1%	27.90% / 1.2%	3.0%
CG (75)	4	131.0 sec	7.02%	6.78%/ 3.4%	6.76%/ 3.6%	
	16	37.2 sec	22.47%	23.18% / 3.2%	21.70% / 3.3%	
IS (80)	4	54.5 sec	48.94%	47.50% / 3.0%	47.30% / 3.2%	
	16	18.0 sec	49.10%	47.23% / 3.8%	47.17% / 3.9%	

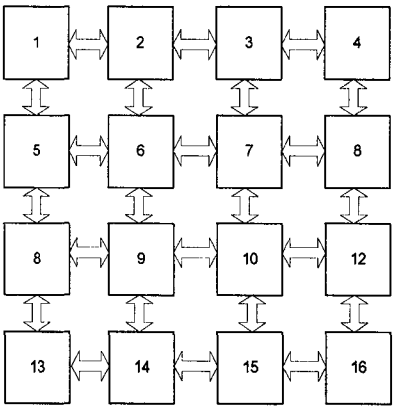
Table2. Dynamic monitoring of a window of iterations, using a simple partical simulation and several NAS benchmarks. $T_{program}$ is overall runtime, $T_{compute}$ is computation time, T_{comm} is communication time, $\%T_{comm}$ is percentage of communication time, N_{nodes} the number of nodes employed, and N_{iter} the overall number of iterations in the program.

The test results in table 2 verify our proposal that we can predict the application level calculation / communication ratio by measuring a window of only a small number of iterations.

10.4 The test of communication volume using ScoPro

To verify that ScoPro can correctly acquire the communication volumes in/out of any nodes involved in the application; we used the 16 nodes particle simulation application as the test case. In ScoPro, this metrics is actually

acquired by summarizing the communication volumes of all the links relating to the corresponding node.



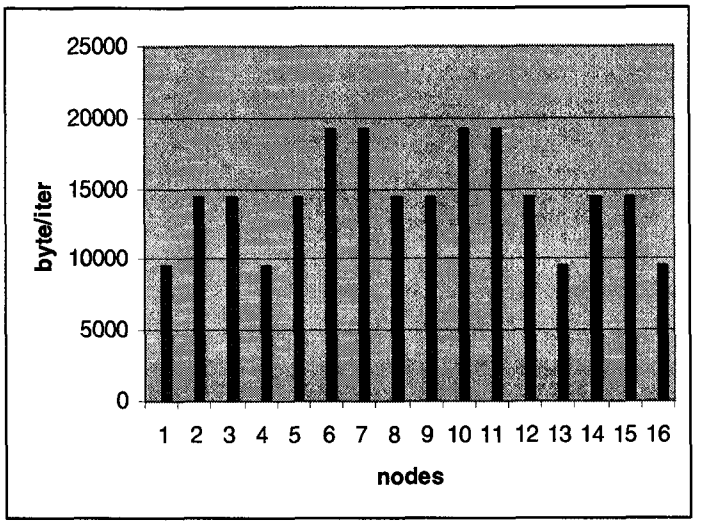


Fig4a. **Fig4b.**

Fig4a(right): The dataflow of particle simulation for 16 nodes.

Fig4b(left): The data volume sent/received by MPI functions of each node.

In the particle simulation application each nodes will communicate with its neighbors as we stated before. For 16 nodes test case, Fig4a shows how each node will communicate with its neighbors. Node2, for example, will communicate with node1, node3 and node6. Because the amount of data each node exchanges with one of its neighbor is the same, the communication volume sent by each node should be proportional to the number of neighbors it has. The test result demonstrated in Fig4b that exactly matches this proportion verifies the correctness of the data monitored.

10.5 Acquiring heterogeneity characteristic by monitoring symmetric parallel NAS benchmark

Both FT package of NAS benchmark and particle simulation application

are symmetric parallel applications which can be used to verify our claim stated in 8.3. First, we made a comparing test by putting the FT application running on 4 nodes (nodes 1,2,3,4 at 2.0Ghz) which have the same running speed; then we put the same application running on 4 nodes, two nodes (nodes15,16 at 2.4GHZ) of which are faster than another two (nodes1,2 at 2.0Ghz). In the latter test, result (in table 3 and right column of table 4a) shows that the faster nodes take much more time to finish blocking synchronous communication calls, while in the former test (results in left column of table 4a) each node generally spends the same amount of time in blocking synchronous communication calls.

	N_{nodes}	T_{comm} avg. per iteration on 2 Ghz nodes	T_{comm} avg. per iteration on 2.4 Ghz nodes
FT	4	1.47 sec	2.22 sec
	8	0.69 sec	1.02 sec
Particle simulation, blocking	4	0.45 msec	1.39 msec
	8	0.69 msec	1.33 msec

Table 3. Communication imbalances (indicating workload imbalance) measured with ScoPro. The data is based on dynamic monitoring of 10 iteration steps. T_{comm} is communication times, N_{nodes} is number of nodes.

We use the same method to test FT on 8 nodes and the simple blocking particle simulation application, and the result (in table 3, table 4a and table 4b) further verified our claim. Thus, we can conclude that for symmetric parallel application, the time spent on blocking MPI functions can reflect the heterogeneity of the nodes running the application.

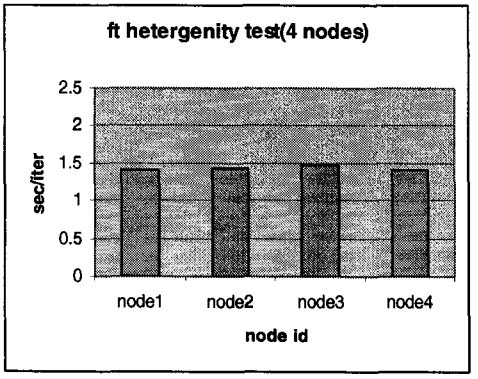
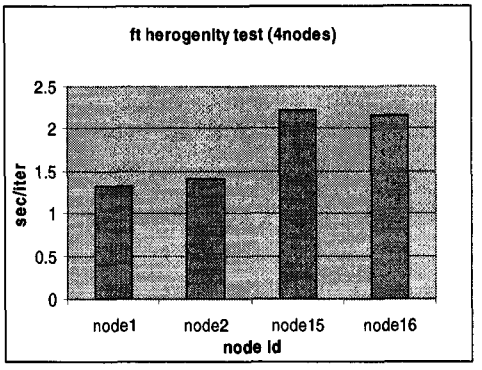
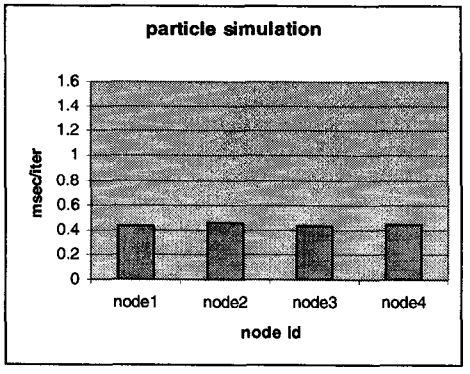
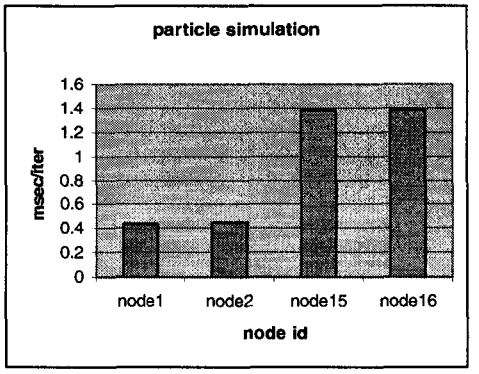
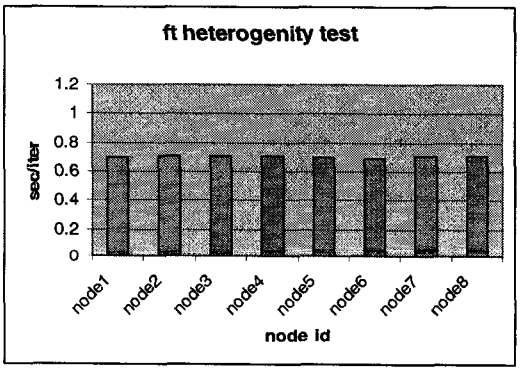
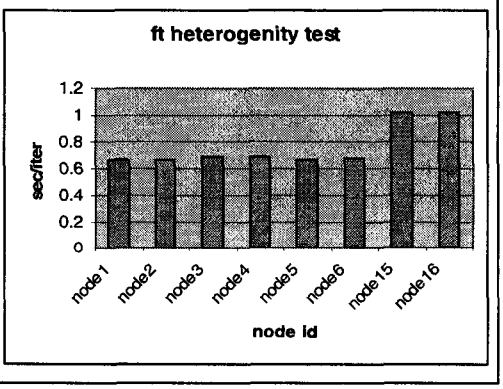
4 nodes test running on node1-4 (all 2Ghz)	4 nodes test running on node1-2(2Ghz), and 15-16(2.4Ghz)
	
	

Table 4.a Environment heterogeneity test for 4 node case, x-axis: blocking communication time per iteration, y-axis: node id

8 nodes heterogeneity test running on node1-8(all 2Ghz)	8 nodes heterogeneity test running on node1-6(2Ghz), and 15-16(2.4Ghz)
	

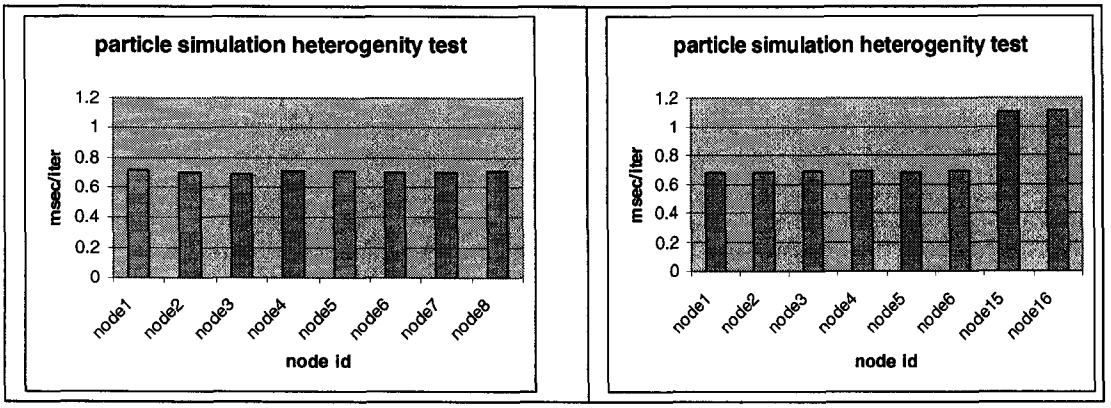


Table 4b. Environment heterogeneity test for 8 node case, x-axis: blocking communication time per iteration, y-axis: node id

In the following test, we monitored a constantly synchronized parallel application, for which the workload is imbalanced. To implement this test, we deliberately modify the blocking particle simulation application by making the calculation workload of node3 and node4 doubled (i.e. twice as much as in node1 and node2).

The test result in table 5 is consistent to our expectation that the synchronous communication time in node1 and 2, where the workload is comparatively lighter, is much more than in node3 and 4, demonstrating that by monitoring the synchronous communication functions, we are able to acquire the information of workload imbalance among processes of parallel application.

	Node1	Node2	Node3	Nodes4
Synchronous Communication time(msec/ iteration)	2.970	2.981	0.423	0.436
Total time(msec/ iteration)	5.681	5.681	5.681	5.682

Table 5. Test of calculation workload of imbalanced application (a modified version of blocking particle simulation).

10.6 Test result for progress of multiple parallel applications and making prediction using historic data

We first dynamically instrument the progress of the 2 parallel applications which are co-scheduled into the same set of resources, and then we compare this result with the historic progress data of the same application that is run without being co-scheduled. The calculated slowdown will be ratio of progress value of co-scheduled application vs. the progress value of the same application without being co-scheduled.

The combinations of co-scheduled applications we use to test include NAS benchmark IS, EP combination and NAS benchmark IS, FT combination. The following table shows the slowdown value we calculated through the instrument data comparing with the real slowdown the applications. To achieve an accurate result, the recording of test data was delayed for 2 iterations. The measurement of co-scheduling of IS and FT is omitted, because for this size, they exceed the available memory per node. Otherwise, we test 4, 8, and 16 nodes. The test result shows that the accuracy of slowdown of monitoring data is kept within the range of 2.5%.

Application	N_{nodes}	S_1 real	S_1 measured by ScoPro / % error	Sl real	Sl measured by ScoPro / % error
		run with IS	run with IS	run with EP	run with EP
IS	4			1.14	1.13 / 1.2%
	8			1.10	1.10 / 0.0%
	16			1.11	1.10 / 0.9%
EP	4	1.14	1.13 / 0.6%		
	8	1.14	1.17 / 2.5%		
	16	1.14	1.15 / 0.9%		
		run with IS	run with IS	run with FT	run with FT
IS	8			1.44	1.42 / 2.0%
	16			1.33	1.33 / 0.4%

FT	8	1.74	1.71 / 1.8%		
	16	1.71	1.73 / 1.5%		

Table 6. Measurement of slowdowns. S_l is slowdown, N_{nodes} is number of nodes.

When two parallel applications are co-scheduled together, one of the applications could finish earlier. If we consider the effect of total the running time of the application which finished earlier in predicting the total running time of the application which finished at a later time, we are able to acquire a more accurate expected running time of this application. Because ScoPro can monitor multiple parallel applications simultaneously and relate the information together, it could provide the necessary data to accomplish this.

Assuming parallel applications A and B are co-scheduled into the same set of resources. According to the data instrumented and history record, we can first predict the application using ScoPro which will finish earlier as we have done in the previous test. Assuming B will finish earlier, Tb is the predicted time of the application B. Assuming A was run previously solely in an environment with an observed Ta_1, we instrument a slow-down of Sa when 2 applications are run together, Tr is the predicted running time of application A after application B ends. Pa is the percentage finished of job A when job B finishes. So we have the following equations:

$$Tb / (Ta_1 * Sa) = Pa \quad (3)$$

$$Tr / Ta_1 = 1 - Pa \quad (4)$$

$$Tr = Ta_1 - Tb/Sa \quad (5)$$

$$Ta = Tr + Tb = Ta_1 + Tb - Tb/Sa \quad (6)$$

From the equations (3) (4), we can get equation (5) then (6). Using equation (6), we predict running the time application A and compare our result with the real observed result.

To demonstrate the effectiveness of our method, we compared the result from formula (6) with the result which has not used the information of another co-scheduled application. This compared method is calculated through formula (7), and it does not take Tb into account. Similar to what we stated previously, Ta1 is the observed running time which Application A runs without co-scheduling, Sa is the instrumented slowdown when 2 applications were run together.

$$Ta = Ta_1 / Sa \qquad (7)$$

The following tables (table7a, 7b) show the test result using formula (6) in comparing the result using formula (7). Because the original IS package is too short to get an accurate test value, we enlarged the iteration number of IS to 80 when used as application B. When IS was tested as application A, in order to make IS running longer than EP and FT, we enlarged its iteration number to 320.

	Ta1(sec)	Sa(sec)	Tb(sec)	Ta(6)(sec)/ error(%)	Ta(7)(sec)/ error(%)	Treal (sec)
IS(A)/EP (B)4nodes	226.22	1.12	148.00	242.5/1.4%	254.16/3.4%	245.77
IS(A)/EP (B) 8nodes	123.46	1.13	75.85	132.24/0.6%	139.6/4.9%	133.00
IS(A)/EP (B) 16nodes	74.26	1.08	38.00	76.95/1.3%	79.92/2.4%	78.01
IS(B)/EP(A) 4nodes	130.85	1.11	67.8	137.56/0.6%	145.24/6.2%	136.71
IS(B)/EP (A) 8nodes	65.43	1.15	36.22	70.09/3.7%	75.25/3.4%	72.8
IS(B)/EP (A) 16nodes	32.91	1.23	21.66	37.06/3.6%	40.71/5.8%	38.45

Table 7a: Execution time prediction test result of Co-scheduled jobs: IS and EP

	Ta1(sec)	Sa(sec)	Tb(sec)	Ta(6)(sec)/ error(%)	Ta(7)(sec)/ error(%)	Treal (sec)
IS(A)/FT(B) 16 nodes	74.13	1.31	56.00	87.50/1.0%	97.38/10.7%	88.46
IS(A)/FT(B) 8 nodes	123.36	1.40	104.83	153.18/1.1%	172.40/13.8%	151.41
IS(B)/FT(A) 16 nodes	32.36	1.75	26.23	43.62/0.0%	56.60/29.8%	43.59
IS(B)/FT(A) 8 nodes	61.42	1.71	47.48	81.03/0.4%	104.6/28.6%	81.36

Table 7b: Execution time prediction test result of Co-scheduled jobs: IS and FT

From the test result, we can conclude that the approach that uses the monitored information of the other co-scheduled application in predicting the execution time has more stable prediction accuracy. In fact the accuracy of this method gets better as the run-time of the applications are longer, while most of the real parallel applications tend to run for much longer period than the application we tested. The test result without relating the information of the other co-scheduled application generally has the same accuracy when the slowdown is low, but it becomes much worse when the slowdown performance get higher.

11. Conclusion and Future work

We have presented a tool—ScoPro, which can dynamically monitor multiple parallel applications. ScoPro is based on Dynaprof and Dyninst_API and can dynamically instrument and un-instrument the binary image of executing applications.

The ScoPro tool can be applied to check application characteristics such as the fraction of time spent in communication or I/O and to check slowdown under coscheduling. Most importantly, it can collect data from monitoring only

short time windows instead of a full program execution. This enables extraction of performance data from applications with little intrusion and makes the tool applicable to realistic job scheduling environments. We have shown that the error introduced by monitoring is small and that monitoring of small windows of iterations is feasible.

Moreover, ScoPro is able to acquire some intrinsic and static behaviors of parallel application including the communication size in/out of a specific node (process), the communication load on a specific link (e.g. node1 to node3), and average message size. This characteristic provides important information for load-balancing and its usage can be potentially expanded for acquiring other application behaviors including IO and memory allocation.

ScoPro is also applicable to checking progress of processes of parallel applications, which enable us to acquire the information of resources' heterogenity where the processes run.

By relating the monitored information of multiple parallel applications acquired ScoPro, we are able to make a more accurate run-time prediction of co-scheduled job.

The future work around ScoPro includes the following:

1. Automate the dynamic insertion of indicating function into main loops of the parallel applications.
2. Integrate other sensors into ScoPro including hardware counter (e.g. PAPI sensor [24]) to acquire more metrics (e.g.FLOPS).
3. Expand usage of ScoPro for monitoring the performance of IO/ Memory functions.

4. Apply ScoPro for performance prediction and monitoring for more complicated structured parallel application.

5. Considering the actual resource times by either estimating (from detailed communication traffic and parameter sizes) the actual time spent on the resource or directly extracting this time by monitoring lower-level libraries such as GM.

6. Improve the scalability of ScoPro by allowing multiple controllers to exist.

12. Reference

[1] Csaba Andras Moritz, Matthew I. Frank, "LoGPC: modeling network contention in message-passing programs", Joint International Conference on Measurement and Modeling of Computer Systems. Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems.

[2] Michael Golm, Christian Wawersich, Joerg Baumann, Meik Felser, and Juergen Kleinoeder, "Understanding the Performance of the Java Operating System JX using Visualization Techniques", Workshop on Software Visualization, OOPSLA 2001, Tampa, FL, October 15, 2001.

[3] Hiroaki Matsui (RIST), Hiroshi Okuda (University of Tokyo), "Thermal Convection Analysis in a Rotating Shell by Parallel FEM - Development of a Thermal-Hydraulic Subsystem of GeoFEM ", The 2nd ACES (APEC Cooperation for Earthquake Simulation) Workshop", October 15-20, 2000, Tokyo and Hakone, Japan."

[4] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, Tia Newhall, "The Paradyn Parallel Performance Measurement Tool", IEEE Computer 28, 11, (November 1995): 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems.

[5] Mucci, P., Dongarra, J., Kufrin, R., Moore, S., Song, F., Wolf, F. "Automating the Large-Scale Collection and Analysis of Performance," In Proceedings of the 5th LCI International Conference on Linux Clusters: The HPC Revolution, Austin, Texas, May 18-20, 2004.

[6] Philip J. Mucci, "Dynaprof Users Guide", http://www.cs.utk.edu/~mucci/dynaprof/dynaprof.html.

[7] A. Alexandrov, M. Jonescu, K.E. Schauser, and C. Scheiman, LogGP: Incorporating Long Messages into the LogP Model, Proc. SPAA '95, July 1995.

[8] Jerry C. Yan Performance Tuning with AIMS — An Automated Instrumentation and Monitoring System for Multicomputers, Proc. 27th Hawaii International Conference on System Sciences, Wailea, Hawaii, January 1994.

[9] Jeffrey Vetter, D. A. Reed Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids , International Journal of High Performance Computing Applications 14(4): 357-366. 2000.

[10] D. A. Reed: Performance Instrumentation Techniques for Parallel Systems. L. Donatiello and R. Nelson (eds), Models and Techniques for Performance Evaluation of

Computer and Communications Systems, Springer-Verlag, LNCS, 1993, pp. 463-490.

[11] Chandra Krintz, Rich Wolski "NwsAlarm: A Tool for Accurately Detecting Resource Performance Degradation (2000)" CCGRID'01, May, 2001.

[12] R. Wolski, "Dynamically Forecasting Network Performance Using the Network", Weather ServiceJournal of Cluster Computing, Vol. 1, No. 1, 1998, pp. 119-132.

[13] F.Vraalsen R.Aydt, C.Mendes D. A. Reed, "Performance Contracts: Predicting and Monitoring Grid Application Behavior" Proceedings of the 2nd Int'l. Workshop on Grid Computing, November 2001.

[14] Bryan Buck, Jeffrey K. Hollingsworth," An API for Runtime Code Patching (2000)", The International Journal of High Performance Computing Applications.

[15] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos,R. Subramonian, and T. von Eicken, "LogP: Towards a RealisticModel of Parallel Computation", Proc. Fourth ACM SIGPLAN Symp. Principles and Practices of Parallel Programming, May 1993.

[16] Shreenivasa Venkataramaiah, Jaspal Subhlok, "Performance Prediction for Simple CPU and Network Sharing", LACSI Symposium 2002, Santa Fe, New Mexico, October 2002.

[17] Bailey, D.H.; Barszcz, E.; Dagum, L.; Simon, H.D.; NAS parallel benchmark results Parallel & Distributed Technology: Systems & Applications, IEEE [see also IEEE Concurrency] , Volume: 1 , Issue: 1 , Feb. 1993
Pages:43 – 51

[18] MPICH-GM. Information available at http://www.myrinet.com. Retrieved 2004.

[19] MPI: A message passing interface standard. International Journal of Supercomputing Applications, 8(3/4), 1994.

[20] Angela C. Sodan. Loosely Coordinated Coscheduling in the Context of Other Dynamic Approaches for Job Scheduling—A Survey. Concurrency&Computation: Practice&Experience. Accepted for publication.

[21] Angela C. Sodan and Lin Han. ATOP—Space and Time Adaptation for Parallel and Grid Applications via Flexible Data Partitioning. 3rd ACM/IFIP/USENIX Workshop on Reflective and Adaptive Middleware, Toronto, Oct. 2004.

[22] Angela C. Sodan and Lei Lan. LOMARC—Lookahead Matchmaking in Multi-Resource Coscheduling. JSSPP (Workshop on Job Scheduling Strategies for Parallel Processing), New York / USA, June 2004, to appear in Springer.

[23] Angela Sodan and Xuemin Huang. SCOJO—Share-Based Job Coscheduling with Integrated Dynamic Resource Directory in Support of Grid Scheduling. Int. Symposium on High Performance Computing Systems (HPCS), Sherbrooke, Canada, May 2003, pp. 213-221.

[24] Dongarra, J., London, K., Moore, S., Mucci, P., Terpstra, D. "Using PAPI for Hardware Performance Monitoring on Linux Systems," *Conference on Linux Clusters: The HPC Revolution*, Linux Clusters Institute, Urbana, Illinois, June 25-27, 2001.

[25] Michael Resch, Holger Berger, Thomas Boenisch Dirk Sihling, High Performance Computing Center Stuttgart, "Performance of MPI on a Cray T3E-512", Third European CRAY-SGI MPP Workshop, Paris (France), Sept. 11 and 12, 1997

[26] Valerie Taylor, X. Wu, X. Li, J. Geisler, Z.Lan, M. Hereld, I. Judson, and R. Stevens, "Prophesy: Automating the Modeling process," in proceedings of 3rd International workshop on Grid computing, Baltimore, 2002

[27] Abdul Waheed, Diane T. Rover, Jeffrey K. Hollingsworth , "Modeling, evaluation, and testing of paradyn instrumentation system", Proceedings of the 1996 ACM/IEEE conference on Supercomputing,1996 , Pittsburgh, Pennsylvania, United States

[28] "DyninstAPI Programmer's Guide", Release 4.1 April 2004, http://www.dyninst.org

[29] Brian Tierney, William Johnston, Brian Cowley, Gary Hoo, Chris Brooks, Dan Gunter, "The NetLogger Methodology for High Performance Distributed Systems Performance Analysis ",In Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC 7), pages 260--267, 1998.

[30] Angela C. Sodan, Lun Liu, "Dynamic Multi-Resource Monitoring for Predictive Job Scheduling with ScoPro", Technical Report 05-002, University of Windsor, Computer Science, February 2005.

13. Annex

13.1 Interface definition

The definition of interfaces provided by ScoPro is given in the following. When the user (e.g. the scheduler) is calling these interfaces, these interfaces will forward the request to the controller, which start the measuring of an application. The definition of structures used by the client interfaces is described in chapter 13.2.

1) MpiJobStart:

```
#include "metcollect.h"
int MpiJobStart ( MpiJobDesc* mpiPtr, int ifBlocking, MetricDesc* m_list,
confirmStruct* cfm)
```

IN mpiPtr: A pointer to mpiJobDesc, which include description of job and measurement, executable name, path, running nodes number, location.

IN ifBlocking: Indicate if this function will return immediately once job is created or return after the monitored data has been acquired.

IN m_list: A list of metric description.

OUT cfm: returned handle to this job.

Return value: indicate whether the job is successfully created

2) mpiMeasure:

```
#include "metcollect.h"
int mpiMeasure(MeasureRequest *m_req,int ifBlocking )
```

IN m_req: including the Jobid (which job to instrument), measuring time,

which subset to be instrument in the initialized list, the event triggering this measurements etc.

IN ifBlocking: Indicate if this function will return immediately or return after the monitored data has been acquired.

13.2 Description of structures in ScoPro

All the structures used by ScoPro are defined in files "metcollect.h" or "metricstr.h". All the structures used by the interfaces of ScoPro are defined and described in the following tables.

1. struct metricHdr : settings controlling the measurement of aspecified job.

Variables	Descriptions
Int measuretime	The maximum time that the measuring will take place
Int metricNum	How many metrics will be measured.
Int detailNum	How many detail trace record will be generated
Int datasize;	The size of memory to be generated, calculated by API, no need to be set by user.
Int combinationCode	Controlling will subset of function group will be measured
Int nodesNum;	The number of process of monitored parallel application
Int m_times;	The number of iterations , in which the recording and measuring happens.
Int start;	The start number of iteration, where the recording of data begins
Int longwait	Whether the application will be monitored all the time when the application kept running(true/false)

2. struct metricDesc : the description of one detailed measured function

Variables	Descriptions
bool recorddetail;	Whether record the trace data or not for this function.
Int num_para1;	First parameter of function to be recorded(-1 if not used)
Int num_para2;	Second parameter of function to be accumulated,

	recorded (-1 if not used)
char functionName[30];	The name of this function
char libName[30];	The name of the library containing this function
Int combinationCode	The group number relating to this metrics

3. struct socketHdr: the header of data package

Variables	Descriptions
char kind;	'S' for synchronized request, 'C' for sending monitored data
char jobId[10];	The job id for the monitored application
Int memsize;	The size of data package followed
Int np;	The process id of parallel application starting from 0
Int batchId;	The batch id of data package.

4. struct mpiJobDesc : description of a job

Variables	Descriptions
char kind;	'C' create a job
struct metricHdr m_header	Description of measurement related to this job
long long starttime	The start time of this job
Int np	Number of processes
char executableName[30]	The executable name of this job
char executablepath[30]	The path of the executable of this job
char machinefilename[30]	Name of machine file
char machinefilepath[30]	The path of machine file

5. struct jobInfo: contains simplified information for a specified job

Variables	Descriptions
char executableName[30]	The executable name of the tested application
Char Date[11];	The date when the application is run
int nodeNumber	How many processes for this job

6. struct nodeSession: information for a specified process

Variables	Descriptions
int sockfd;	The connection number for this process
int processId;	The process id
Char nodeName[30];	The name of the node running this process

7. struct confirmStruct: the handle to access a specified job

Variables	Descriptions
int jobId	A unique id related to this job
Int slotNumber	Fast access number of this job, automatic generated.

8. struct measureRequest: description for the restarted measure request.

Variables	Descriptions
char kind	'M' for make a measurement request.
confirmStruct jhandle	The handle of this job
Int combinationCode	The subset of measured functions
Int eventId;	The event id triggering this measurement

9. struct BatchHdr: Description for this batch of data

Variables	Descriptions
int batchId	The ID number for this batch of data, starting from 0.
int datasize	The size of data in byte for this batch of data
int eventId	The event id trigger this batch of data

10. struct wallclock_metric_data_t: Monitored data for one specified function

Variables	Descriptions
Long long current	The most recent walltime when the function is called
Long long total	The total amount of time the monitored function consumes
Long calls	The total times the monitored function is called
Long long min	The walltime when the function is called for the first time after the record of monitored data begins
Long long max	The walltime when the function is called for the last time

	before the record of monitored data ends
Long long para_sum	The sum of the values of a specified parameter of the monitored function

11. Struct Linkdata: contain the the total communication volume sending from the monitored process to another process

Variables	**Descriptions**
Long long abso	The total communication volume sending from the monitored process to another process

12. Stuct detail_data_t: The tracing record for the monitored functions

Variables	**Descriptions**
long long abso	The wall time when the function is called
long long current	Indicate how long this function is called
Int para1	The value of the first parameter when the function is called
Int para2	The value of the second parameter when the function is called

13. wallclock_data_hdr_t : contain information of measurement settings where both mutator and mutatee can access.

Variables	**Descriptions**
Struct batchhdr batchinfo	Header information for current batch
Int Combinationcode	Indicate the subset of current monitored functions
Int pid	The process id of monitored process
Int nodeid	The id of this node
Int nodenum	The number of processes belongs to the monitored application
Int finished	Indicate whether the current process has been finished
Int measuring	Indicate whether the record of data is switched on

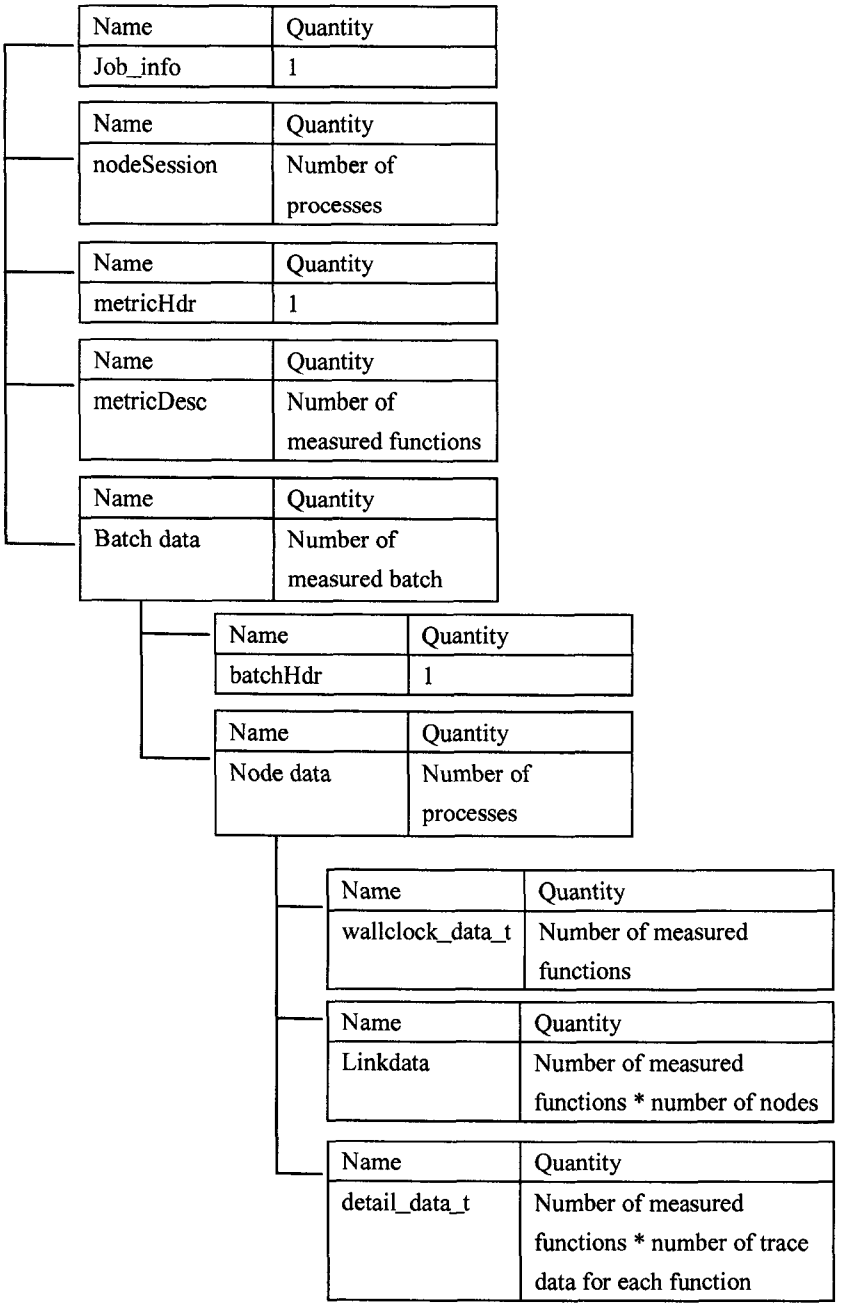
13.3 Description of data package sent from sensors to the controller

The data package sent from sensors to the controller is composed of the components listed in the following table. The order of the components in the data package is the same as the listed order in the table.

Sub component	Number of components included in the package
Socket_hdr	1
wallclock_data_hdr_t	1
wallclock_data_t	Number of instrumented functions
Linkdata	Number of processes
detail_data_t	Number of tracing records * number of traced functions

13.4 Data structure for measured data of each job

The following Figure shows the file format of monitored data saved in the persistent storage. ScoPro will generate a separate file whenever a parallel application is started and monitored.



13.5 How to invoke the tool

1. Set up the running environment

1) Set files ".bashrc" and ".bash_profile" to include the path of dyninst_API and the libraries of Dynaprof probes.

```
DYNINSTAPI_RT_LIB=/home/shared/dyninstAPI-v4.1.1/i386-unknown-linux2.4/lib/libdyn
instAPI_RT.so.1
DYNAPROF_PROBEDIR=/home/shared/usr/lib
LD_LIBRARY_PATH=/home/shared/usr/lib: $LD_LIBRARY_PATH
```

2) Copy "/home/liu/mpich*/ch_gm/bin/mpirun.ch_gm1.pl" to a public accessible path, and make sure the interface ("mpijobstart") can access this file

3) Traditional editing of the machine file of MPI.

2. Declarations and settings call the interface

1) In the source code to call the interface, include the following 2 header files.

```
#include "metCollect.h"
#include "probes/metricstr.h"
```

2) Declare the following variables in your source code.

```
MpiJobDesc myJob;
confirmStruct cfm;
metricDesc metList[8];   // larger than the maximum functions to be monitored
```

3) Set connection to the controller

```
set_connection("horus.newcs.uwindsor.ca");
```

4) Set of functions to be monitored

- Set function name:
  strcpy(metList[0].functionName,"measuremark_");
  // other functions
- Set function library
  strcpy(metList[0].libName,"DEFAULT_MODULE");
  // other functions
- Set whether generate trace record or not
  metList[0].recorddetail=true;
  // other functions
- Set combinationcode(which subset of functions to be monitored)

```
metList[0].combinationCode=0;
metList[1].combinationCode=1;
// other functions
```

- set 1st parameter to be monitored
  metList[0].num_para1=-1; // -1 stand for ignore, +2 for mpi_isendcontig
  // other functions
- set 2nd parameter to be monitored
  metList[0].num_para2=-1; // -1 stand for ignore, +6 for mpi_isendcontig

5) Set job descriptions, for example

```
strcpy(myJob.machinefilename, "mfile");          // machine file name
strcpy(myJob.executableName,"/home/liu/ft.B.4"); //excutable name, path
myJob.kind='C';
myJob.np=4;                                      // the # of processes of this parallel application
```

6) Set the properties of measurement which is common to all the functions, for example

```
myJob.m_header.m_times=3;       // How many calls to be monitored, 0 to be ignore
myJob.m_header.idletime=50;     // How long in second the monitoring will last
myJob.m_header.detailNum=20;    // The trace records to be generated
myJob.m_header.metricNum=4;     // The maximum functions to be monitored
myJob.m_header.start=0;         // The begin of recording data (depending on the
                                // 1st functions in the function list)
myJob.m_header.longwait=false;  // whether the monitoring will stopped once it
                                // begins
```

7) Add calls to the interface, for example

```
MpiJobStart(&myJob,true,metList,cfm);
```

8) Add following statement for restarted Measurement

```
Request m_req;                      // Declare an instance of measuring request
m_req.jobId=cfm.jobId;              // Part of the handle for this job
m_req.slotNumber=cfm.slotNumber;    // Part of the handle for this job
m_req.kind='M';
m_req.combinationCode=1;            // Determine the subset of monitored functions
m_req.eventId=2;                    // The event ID for this triggering this
                                    // measurement
sleep(SOME_SECONDS)                 // Make sure the last batch of application is
                                    // finished
mpiMeasure(&m_req,true);            // Call the interface
```

3. Compile the source code of user invocation.

4. If the server is not running, run the server by initiate the following command:

   /$Homedirectory/dynaserver 0 (or other start number)

5. Run the client executable

6. The generated file name is "currentdate"+jobID+".plog" (e.g. 20050415000010.plog)

7. Read the file by issuing the following command:

   logreader "the name of the generated plog file."

VITA AUCTORIS

NAME:	Lun Liu
PLACE OF BIRTH:	Beijing, China
YEAR OF BIRTH:	1967
EDUCATION:	Xi'an University of Electronic Science and Technology, 1986-1990 B.Sc. University of Windsor, Windsor, Ontario 2002-2005 M.Sc.
WORKING EXPERIENCE:	No. 14 Research Institute of China CALT Academy 1990-1996, Associate Engineer Beijing Branch, IBM Ltd., China 1997-2000, System Analyst