MPI Performance Analysis Tools on Blue Gene/L

I-Hsin Chung Robert E. Walkup Hui-Fang Wen Hao Yu

IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598
{*ihchung,walkup,hfwen,yuh*}@*us.ibm.com*

Abstract

Applications on today's massively parallel supercomputers are often guided with performance analysis tools toward scalable performance on thousands of processors. However, conventional tools for parallel performance analysis have serious problems due to the large data volume that needs to be handled. In this paper, we discuss the scalability issue for MPI performance analysis on Blue Gene/L, the world's fastest supercomputing platform. First we present an experimental study of existing MPI performance tools that were ported to BG/L from other platforms. These tools can be classified into two categories: profiling tools that collect timing summaries, and tracing tools that collect a sequence of time-stamped events. Profiling tools produce small data volumes and can scale well, but tracing tools tend to scale poorly. We then describe a configurable MPI tracing tool developed for BG/L. By providing a configurable method for trace generation, the volume of trace data can be controlled, and scalability is significantly improved.

Keywords: performance tool, performance tuning

1 Introduction

The Blue Gene[1]/L (BG/L) supercomputer is a massively parallel system developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). BG/L uses system-on-a-chip technology [Almasi et al. 2002] to integrate powerful torus and collective networks with the processors onto a single chip, and it uses a novel software architecture [Adiga et al. 2002] to support high levels of scalability. The BG/L system installed at LLNL contains 65,536 dual-processor compute nodes. Operating at a clock frequency of 700 MHz, BG/L delivers 180 or 360 Teraflop/s of peak computing power, depending on its mode of operation.

Solving a scientific problem on BG/L typically requires more processors than conventional supercomputers because BG/L has relatively low-power processors, both in terms of Watts consumed and Flops produced. Therefore, to effectively utilize the enormous computation power provided by BG/L systems, it is critical to explore scalability to the maximum extent. It has been shown that a significant number of important real-world scientific applications can be effectively scaled to thousands or tens of thousands BG/L processors [Almasi et al. 2005b]. The process of achieving such high level scalability and performance is often difficult and time-consuming, but can be guided with scalable parallel performance analysis tools.

Performance analysis tools utilize information collected at run-time to help program developers identify performance bottlenecks and in turn improve application performance. For massively parallel systems like BG/L where scalability is the key, scalable tools for analyzing communication are needed. On BG/L, communication is via the message-passing interface, MPI. The MPI specification includes a "profiling" interface, which provides entry points to intercept and instrument the message-passing calls. One can distinguish between two types of parallel performance tools: (1) profiling tools that generate cumulative timing information, and (2) tracing tools that collect and display a sequence of time-stamped events. It is clear that tracing tools face a fundamental difficulty when there are thousands of processes involved. The volume of trace data can easily get to be unmanageable. As a result, post-mortem data reduction schemes based on filtering the aggregate data may still be untenable. Instead, it is necessary to carefully control trace generation. In contrast, profiling tools are inherently more scalable, because they retain only cumulative data.

In this paper, we discuss the scalability of MPI profiling and tracing tools on BG/L. We first discuss existing MPI tracing tools and show that they have serious problems for large-scale BG/L systems. Specifically, we present an experimental study of existing tools that are ported onto BG/L from other platforms. We then present a configurable MPI tracing tool developed specifically for BG/L. Our novel tracing tool defines a simple and flexible interface allowing users to control the generation of trace data. This is much more effective

[1]Trademark or registered trademark of International Business Machines Corporation.

than post-processing large sets of trace data. For example, with this approach, it is possible to automatically detect and filter out repeating communication patterns at run time. The configurability of our tool makes it relatively easy to limit the generation of trace data, and this results in a significant improvement of the scalability of the tool.

The rest of the paper is organized as follows. Section 2 gives an overview of MPI on the BG/L. We study the scalability of existing MPI performance analysis tools on BG/L by using several real applications in Section 3 and 4. The new MPI tracing library we developed is introduced in Section 5, followed by descriptions of the experiences and lessons learned after using our new library in a set of experiments in Section 5.4. Related work is given in Section 6. Finally, the conclusion is drawn in Section 7.

2 MPI on BG/L

On BG/L, each process can directly access only its local memory, and message-passing is used for communication between processes. The current implementation of MPI on BG/L [Almasi et al. 2003] is based on MPICH2 [mpi a] from Argonne National Laboratory. The BG/L version is MPI-1.2 compliant [Almasi et al. 2005a] and supports a subset of the MPI-2 standard. There are parts of MPI-2, such as dynamic process management, that are not supported.

The MPI implementation on BG/L utilizes three high-speed networks: a three dimensional (3D) torus network for point-to-point communication [Adiga et al. 2005], a collective network for broadcast and reduction operations, and a global interrupt network for fast barrier synchronization. For the torus network, each compute node is connected to its six neighbors through bidirectional links, and the links can be "trained" to wrap around on partitions that are multiples of a midplane (a unit with 512 nodes in an 8x8x8 torus network configuration).

Another important architectural feature of BG/L is that each compute node has two processors. A compute node can operate in one of two modes. In *coprocessor* mode, a single-threaded MPI process occupies each node, and the second processor is used to aid in MPI communication. In *virtual node* mode, there are two single-threaded MPI processes per node, each with access to half of the node memory. The two processors on each chip do not have hardware support for cache coherency, so pthreads or OpenMP are not currently supported on BG/L. As a result, MPI has a very important role for parallel applications on BG/L.

Table 1: Existing MPI Performance Analysis Tools

Tool	Function Category	References
IBM HPCT	profiling and tracing	[act]
Paraver	tracing	[Pillet et al. 1995; par b]
KOJAK	tracing	[Mohr and Wolf 2003; koj]
TAU	profiling and tracing	[Malony et al. 2004; tau]
mpiP	profiling	[mpi b]

3 MPI Performance Analysis Tools for BG/L

A number of MPI performance analysis tools have been made available for BG/L users by joint efforts of IBM and collaborators. In this section, we first give brief introductions to these tools. Then, in Section 4, we present an experimental evaluation of the scalability, efficiency, and overhead of these tools when applying them on BG/L. Table 1 lists the MPI performance tools that were ported to BG/L and used in this study. These tools can be classified into two categories: (1) profiling tools that provide cumulative data, and (2) tracing tools[2] that provide time-stamped records of MPI events.

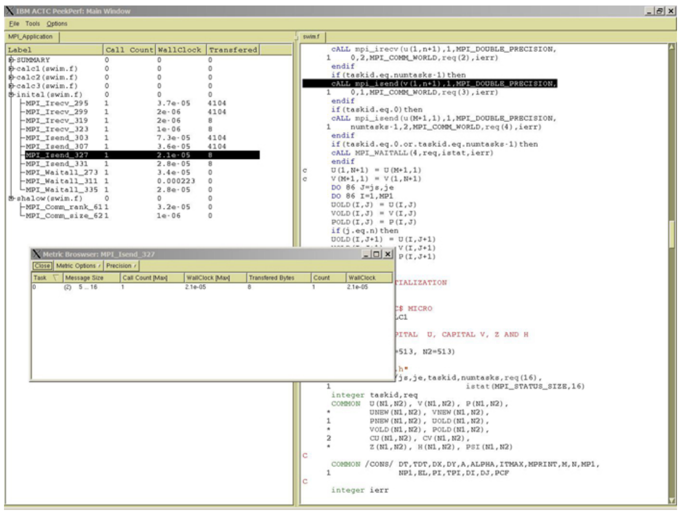
3.1 IBM High Performance Computing Toolkit

The IBM High Performance Computing Toolkit (IBM HPCT) contains MPI profiler and tracer libraries [act] to collect profiling and trace data for MPI and SHMEM programs. This toolkit was originally developed for AIX Power clusters and now is ported to BG/L. The IBM HPCT generates two types of output files that work with the visualization tools Peekperf and Peekview to identify performance bottlenecks. Peekperf integrates a source-code browser with cumulative performance metrics obtained with the profiler library (Figure 1(a)), and Peekview displays the time-stamped MPI trace file obtained via the tracer library (Figure 1(b)).

3.2 Paraver

Paraver [Pillet et al. 1995; par b] is a program visualization and analysis tool that supports both shared-memory and distributed-memory parallel applications. It has three major components: a tracing facility, a trace merge tool, and a visualizer. For MPI tracing, the MPItrace library is used to intercept MPI calls and save individual trace files during application execution. The individual files are then merged, and the merged trace file is displayed using the viewer, which has many display and analysis features. Paraver trace includes

[2]Tracing tools cause much less overhead for scalability. We describe them here for the completeness of MPI performance tools on BG/L.



IBM ACTC PeekPerf: Main Window

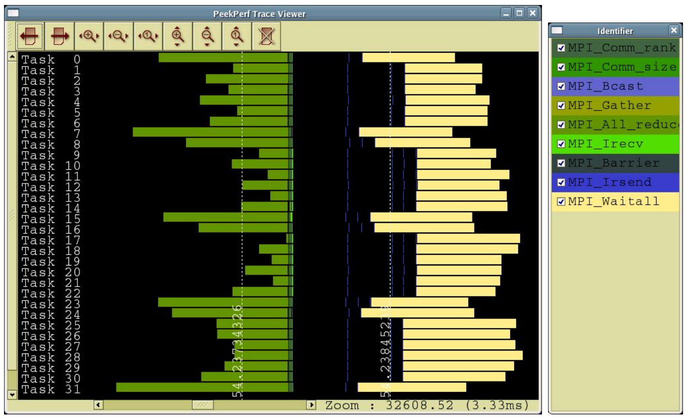
Label	Call Count	WallClock	Transfered
SUMMARY	0	0	0
calc1(swim.f)	0	0	0
calc2(swim.f)	0	0	0
calc3(swim.f)	0	0	0
inital(swim.f)	0	0	0
MPI_Irecv_295	1	3.7e-05	4104
MPI_Irecv_299	1	2e-06	4104
MPI_Irecv_319	1	2e-06	8
MPI_Irecv_323	1	1e-06	8
MPI_Isend_303	1	7.3e-05	4104
MPI_Isend_307	1	3.6e-05	4104
MPI_Isend_327	1	2.1e-05	8
MPI_Isend_331	1	2.8e-05	8
MPI_Waitall_273	1	3.4e-05	0
MPI_Waitall_311	1	0.000223	0
MPI_Waitall_335	1	2.8e-05	0
shalow(swim.f)	0	0	0
MPI_Comm_rank_61	1	3.2e-05	0
MPI_Comm_size_62	1	1e-06	0

Metric Broswser: MPI_Isend_327

Task	Message Size	Call Count [Max]	WallClock [Max]	Transfered Bytes	Count	WallClock
0	(2) 5 ... 16	1	2.1e-05	8	1	2.1e-05

```
      cALL mpi_irecv(u(1,n+1),1,MPI_DOUBLE_PRECISION,
     1    0,2,MPI_COMM_WORLD,req(2),ierr)
      endif
      if(taskid.eq.numtasks-1)then
      cALL mpi_isend(v(1,n+1),1,MPI_DOUBLE_PRECISION,
     1    0,1,MPI_COMM_WORLD,req(3),ierr)
      endif
      if(taskid.eq.0)then
      cALL mpi_isend(u(M+1,1),1,MPI_DOUBLE_PRECISION,
     1    numtasks-1,2,MPI_COMM_WORLD,req(4),ierr)
      endif
      if(taskid.eq.0.or.taskid.eq.numtasks-1)then
      cALL MPI_WAITALL(4,req,istat,ierr)
      endif
c     U(1,N+1) = U(M+1,1)
c     V(M+1,1) = V(1,N+1)
      DO 86 J=js,je
      DO 86 I=1,MP1
      UOLD(I,J) = U(I,J)
      VOLD(I,J) = V(I,J)
      POLD(I,J) = P(I,J)
      if(j.eq.n)then
      UOLD(I,J+1) = U(I,J+1)
                    V(I,J+1)
                    P(I,J+1)

                   TIALIZATION

                   C$ MICRO
                   LC1

                   PITAL  U, CAPITAL V, Z AND H

                   =513, N2=513)

                   .h"
                   /js,je,taskid,numtasks,req(16),
     1                             istat(MPI_STATUS_SIZE,16)
      integer taskid,req
      COMMON  U(N1,N2), V(N1,N2), P(N1,N2),
     *        UNEW(N1,N2), VNEW(N1,N2),
     1        PNEW(N1,N2), UOLD(N1,N2),
     *        VOLD(N1,N2), POLD(N1,N2),
     2        CU(N1,N2), CV(N1,N2),
     *        Z(N1,N2), H(N1,N2), PSI(N1,N2)
C
      COMMON /CONS/ DT,TDT,DX,DY,A,ALPHA,ITMAX,MPRINT,M,N,MP1,
     1              NP1,EL,PI,TPI,DI,DJ,PCF
C
      integer ierr
```

(a) Source Code Performance Metrics Mapping



(b) Trace Visualization

Figure 1: IBM HPCT Visualization for MPI Performance Analysis

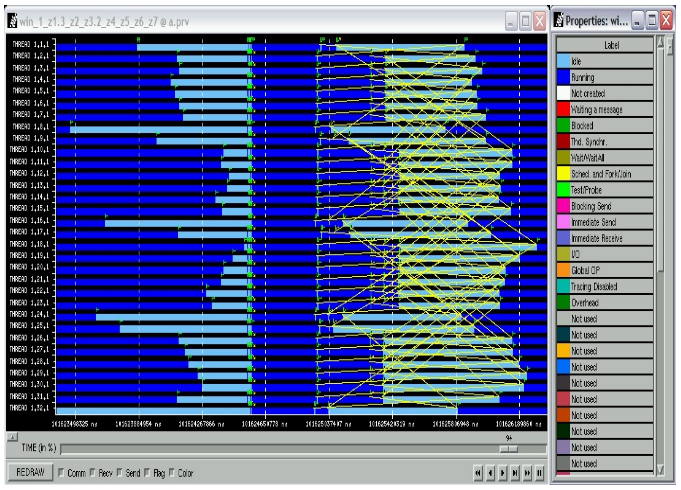


Figure 2: Paraver Visualization for MPI Tracing

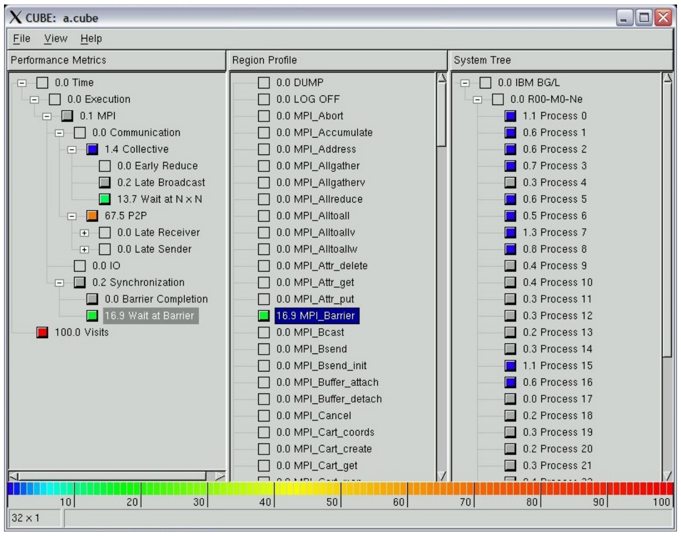
detailed information such as states, events and communications. An example of Paraver for MPI trace visualization is shown in Figure 2. Both the trace merge tool and the viewer run on BG/L front-end nodes, while trace generation is done from the BG/L compute nodes. Paraver is best suited for parallel applications at a modest scale by BG/L standards, because at large processor counts the trace files become large and hard to work with. This basic difficulty affects all tracing tools to some extent.

3.3 KOJAK

KOJAK (Kit for Objective Judgment and Knowledge-based Detection of Performance Bottlenecks) [Mohr and Wolf 2003; koj] is a collaborative research project aiming at the development of a generic automatic performance analysis environment for parallel programs. It includes a set of tools performing program analysis, tracing, and visualization. The instrumentation for MPI is obtained with the PMPI interface, which intercepts calls to MPI functions. KOJAK uses the EPILOG run-time library, which provides mechanisms for buffering and trace-file creation. On BG/L, the resulting trace files can be quite large. In terms of visualization, KOJAK provides several options including tree-style hot spot analysis (Figure 3(a)). The user can identify performance bottlenecks by exploring the tree. There is also a topology view (Figure 3(b)) to help the user map performance metrics back to the compute nodes in terms of the physical layout on the torus network.

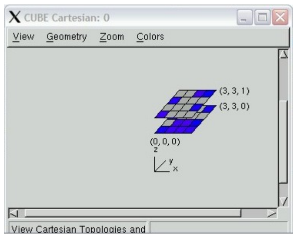
3.4 TAU

TAU [Malony et al. 2004; tau] is a program and performance analysis framework. It includes a suite of static and dynamic tools that form an integrated analysis environment for parallel applications. TAU includes automatic instrumentation to capture data for functions, methods, basic blocks, and program statements. In addition to automatic instrumentation, TAU provides an API for manual instrumentation. TAU can be used for either profiling (collecting cumulative data) or tracing (recording time-stamped events). TAU includes a visualizer, Paraprof, for profile data. For trace data, TAU does not include it's own trace viewer, but it can convert trace files for use with other visualization tools, such as Paraver. TAU has many features and has been ported to a variety of platforms. In our experiments with TAU, we used only the MPI profiling capability[3].



```
Performance Metrics
0.0 Time
  0.0 Execution
    0.1 MPI
      0.0 Communication
        1.4 Collective
          0.0 Early Reduce
          0.2 Late Broadcast
          13.7 Wait at N x N
        67.5 P2P
          0.0 Late Receiver
          0.0 Late Sender
      0.0 IO
      0.2 Synchronization
        0.0 Barrier Completion
        16.9 Wait at Barrier
100.0 Visits

Region Profile
0.0 DUMP
0.0 LOG OFF
0.0 MPI_Abort
0.0 MPI_Accumulate
0.0 MPI_Address
0.0 MPI_Allgather
0.0 MPI_Allgatherv
0.0 MPI_Allreduce
0.0 MPI_Alltoall
0.0 MPI_Alltoallv
0.0 MPI_Alltoallw
0.0 MPI_Attr_delete
0.0 MPI_Attr_get
0.0 MPI_Attr_put
16.9 MPI_Barrier
0.0 MPI_Bcast
0.0 MPI_Bsend
0.0 MPI_Bsend_init
0.0 MPI_Buffer_attach
0.0 MPI_Buffer_detach
0.0 MPI_Cancel
0.0 MPI_Cart_coords
0.0 MPI_Cart_create
0.0 MPI_Cart_get

System Tree
0.0 IBM BG/L
  0.0 R00-M0-Ne
    1.1 Process 0
    0.6 Process 1
    0.6 Process 2
    0.7 Process 3
    0.3 Process 4
    0.6 Process 5
    0.5 Process 6
    1.3 Process 7
    0.8 Process 8
    0.4 Process 9
    0.4 Process 10
    0.3 Process 11
    0.3 Process 12
    0.2 Process 13
    0.3 Process 14
    1.1 Process 15
    0.6 Process 16
    0.0 Process 17
    0.2 Process 18
    0.3 Process 19
    0.2 Process 20
    0.3 Process 21
```

(a) Hotspot Analysis



(b) Topology

Figure 3: KOJAK Visualization for MPI Profiling

3.5 mpiP

mpiP [mpi b] is a light-weight profiling library for MPI applications. It collects cumulative information about MPI functions. It profiles on the basis of call sites rather than just MPI commands. It uses communication only during report generation, typically at the end of the program execution. Since it collects only cumulative information, the output size is very small compared to MPI tracing tools, and the execution time overhead is normally small. However, the detailed time history of communication events is not available with this tool.

4 Performance Study

In this section, we study the scalability, efficiency, and overheads of the existing MPI performance analysis tools on BG/L.

4.1 Applications

We used a collection of scientific applications on BG/L to examine the strengths and limitations of parallel performance tools. The applications are briefly described below.

SAGE [Kerbyson et al. 2001] is an Adaptive Grid Eulerian hydrodynamics application from Science Applications International. SAGE uses blocks of cells in a three-dimensional Cartesian framework. SAGE has many features including the ability to do automatic mesh refinement. Two input sets were used: one that does significant computational work but has a static mesh, and another that exercises the automatic mesh refinement capability.

FLASH [fla] is a parallel adaptive-mesh multi-physics simulation code designed to solve nuclear astrophysical problems related to exploding stars. The particular test case that

[3]The tracing part is not available on the system at the time of this writing.

we used was a two-dimensional weak scaling problem [Sod 1978], which includes an expanding shock wave and exercises the adaptive-mesh refinement capability.

SOR is a program for solving the Poisson equation using an iterative red-black SOR method. This code uses a two dimensional process mesh, where communication is mainly boundary exchange on a static grid with east-west and north-south neighbors. This results in a simple repeating communication pattern, typical of grid-point codes from a number of fields.

sPPM [spp] is a simplified version of the Piecewise Parabolic Method (PPM), a hydrodynamics algorithm which is particularly useful when there are discontinuities such as shock waves . The application uses a three-dimensional process mesh, and communication is by exchange with nearest neighbors in +/-x, +/-y, and +/- z directions. This communication pattern fits nicely onto the 3D torus network, and as a result sPPM scales almost perfectly on BG/L.

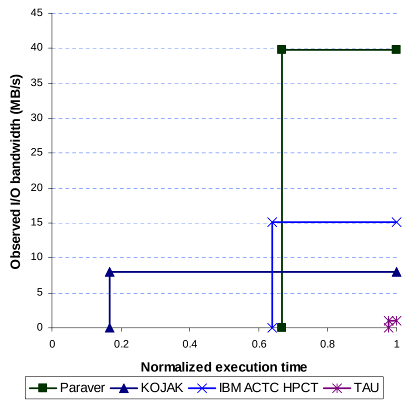
SWEEP3D [swe] is a simplified benchmark program that solves a neutron transport problem using a pipelined wavefront method and a two-dimensional process mesh. Input parameters determine problem sizes and blocking factors, allowing for a wide range of message sizes and parallel efficiencies.

4.2 Performance Measurement

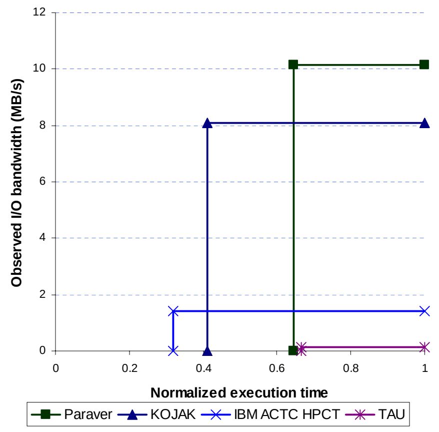
In this section we report measurements using the existing MPI performance tools with the collection of applications listed in Section 4.1. Two metrics were chosen to characterize the tools: (1) execution-time overhead [4] $((elapsed\_time - reference\_time)/reference\_time$, where the $reference\_time$ is measured without profiling or tracing); in other words, we compare the application execution time with and without using a MPI performance analysis tool, and (2) the volume of collected performance data.

Figures 4(a) and 4(b) show the measurements, using weak scaling (work per processor remains constant) for a set of applications, with 512, 1,024, and 2,048 compute nodes on BG/L in coprocessor mode[5]. The execution-time overhead for the profiling methods provided by TAU and mpiP was less than 3% and is not shown in the figures. In our experience, profiling tools can provide useful cumulative data up to very large systems, including the full 64K-node Blue Gene/L at Livermore, with relatively little overhead. In contrast, the execution-time overhead for the tracing methods used by IBM HPCT[6], Paraver, and KOJAK grows faster than linearly with the number of MPI processes, and is already of order 100(factor of two slower performance) with 1,024 processes. Two possible reasons for this are: (1) The contention among nodes for system resources (e.g., network bandwidth for I/O activities) and (2) The artifact of weak scaling from the application. The overall performance is slowed down significantly with 2,048 or more MPI processes. Also, the volume of trace data collected, shown in Figure 4(b), can quickly grow to of order 100 GBytes, which is too large for efficient analysis or visualization. At very large process counts, it is simply not practical to save all time-stamped records from every MPI rank.

A closer examination reveals that the execution-time overhead is substantially affected by the generation of trace file output. The trace file I/O behavior is not uniform over the



(a) FLASH, 1,024 Compute Nodes



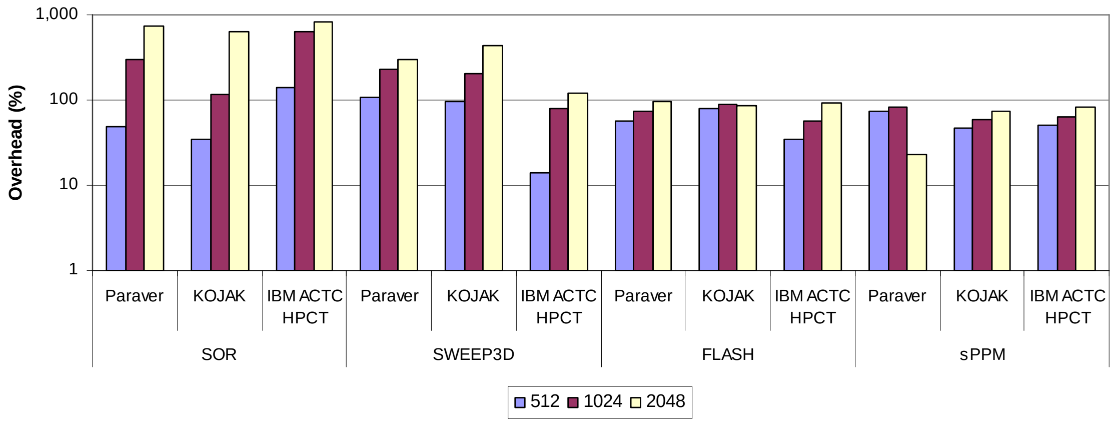
(b) sPPM, 1,024 Compute Nodes

Figure 5: Performance Analysis Tool I/O Distributions

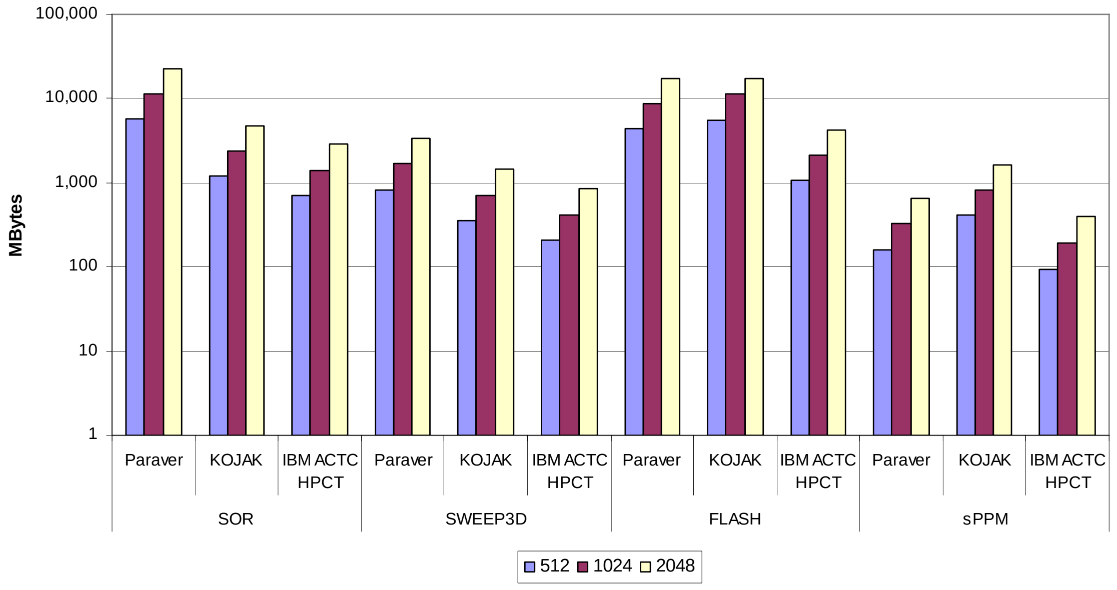
[4]It is preferred to have performance tools output performance data at the end of execution to minimize the performance perturbation. However, it is commonly seen that tools output performance data in the middle of execution due to limited buffer/memory space available to the tools.

[5]Only results for applications with tracing tools finish successfully for all three block sizes are shown.

[6]The IBM HPCT used in this section (Section 4) is the version developed on the AIX platform and ported onto BG/L. It does not include configurable functionality therefore every MPI rank will generate output files for all collected performance data.



(a) Execution Time Overhead



(b) Performance Data Size

Figure 4: MPI Performance Analysis Tools and Applications

course of program execution. For example, the I/O bandwidth obtained when tracing FLASH and sPPM is shown in Figure 5 as a function of time. The tracing tools use buffers in memory to save event records, but when the buffers are full it is necessary to write data to the file-system. In order to minimize the performance impact of tracing on systems like BG/L that only supports blocked I/O, it is advantageous to use large memory buffers, and delay file I/O until near the end of program execution. In our experiments, Paraver follows this approach and achieves somewhat better utilization of I/O bandwidth. The execution-time overhead for tracing could be kept to a minimum by keeping all trace data in memory, but that is not feasible for long-running applications with large numbers of events. The BG/L system that we used for these experiments did not have a parallel file system attached to it. The output files were written to an NFS file system mounted on the nodes. An aggressive parallel file-system could significantly reduce the execution-time overhead caused by trace file generation. However, the problem of managing the vast amounts of trace data would remain.

5 Configurable MPI Tracing Library

There are a number of ways to reduce the data volume associated with MPI tracing. For example, if the application does similar work for many time-steps, one could limit tracing to just one or two time-steps, either by inserting calls to start/stop tracing within the application, or by setting run-time parameters to limit the total number of time-steps in the numerical simulation. By selectively tracing just one part of the application, the data volume can be dramatically reduced. Also, the nature of large-scale parallel applications is that many MPI processes are doing very similar work, so one can limit trace generation to a subset of MPI ranks and still obtain useful insight into the time-dependent behavior of the application. Our experience has shown that good insight can be obtained by saving data from about 100 MPI processes, which can reduce the data volume by roughly a factor of 1,000 for large BG/L systems. One can imagine that there are a number of ways to select MPI ranks. For example, one could pick a contiguous range of MPI ranks, or one could define a communication neighborhood based on actual communication patterns, or one could choose a contiguous block of processes based on the topology of the network (a three-dimensional torus for BG/L).

In this section, we describe a more general way to make the tracing tool configurable, and thereafter allows users to focus on interesting performance points. For the evaluation, we use it to automatically detect repeating communication patterns, and save just the repeated pattern once. By providing a flexible mechanism to control the events that are recorded, the tracing tool can remain useful for even very large-scale parallel applications.

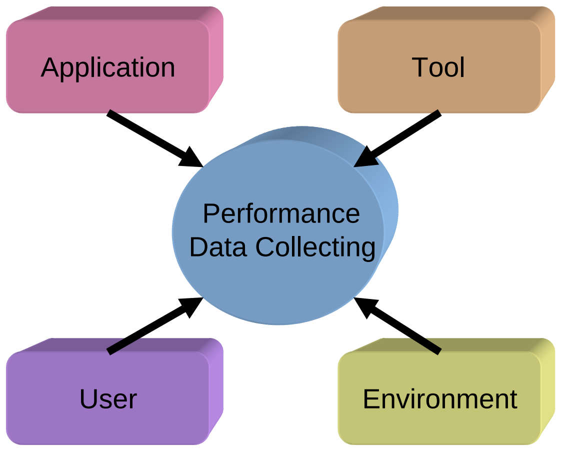


Figure 6: Configurable Profiler

5.1 Design

Conventional performance analysis tools usually record performance information that are pre-defined by the tool developers. The performance analysis tools usually collect all possible primitive performance metrics such as time and bytes. In addition to that, the tool developer also has to "imagine" what are possible useful derived performance metrics (e.g., bandwidth: bytes/seconds) based on those primitive performance metrics measured.

This mechanism works well with small scale of computations with limited number of processors. However, lacking of flexibility makes the scalability a critical issue as the computation scale (e.g, number of processors) goes up. The overhead of the information collection activities in a performance analysis tool can dominate the system resource usage (e.g., memory used for tracing). The transition (e.g., through the network) and the storage for the collected performance information is another challenge. These overheads can make the performance analysis tools unpractical (e.g., too much memory space required for detailed tracing). Even the system has sufficient amount of resource to handle these issues, it will be difficult to analyze the extraordinary large amount of performance information collected.

We believe that a scalable or an efficient performance analysis tool should take the factors shown in Figure 6 into consideration while collecting performance information. The four factors that affect the performance analysis tool are:

- **Application** The useful performance information may only reside in a few code segments (e.g., certain loops). Therefore, the performance analysis tool should be able to access such program/software-specific information during performance tracing. Otherwise, collecting performance information from all code segments would just cause unnecessary overhead and waste the system resource.

- **Tool** There are variables (such as available memory space for the trace buffer) from the performance analysis tool itself that can affect the performance information collection. On BG/L, if the performance analysis tool uses too much memory, application execution may fail because there are hard-limits of memory usage for each process.
- **Environment** The performance analysis tool needs to take the system environment into consideration. On BG/L, tool users may be only interested in the performance information for certain MPI ranks. Or users may expect different MPI ranks to collect different performance information.
- **User** Finally, tool users should be able to get involved in determining what performance information is the most of interest. For example, a derived performance metric can be a user-defined function that takes measured primitive performance metrics as its input. The capability would save the overhead for both the tool (to calculate derived metrics) and users (to analyze the collected performance information) compared to the post-mortem data analysis.

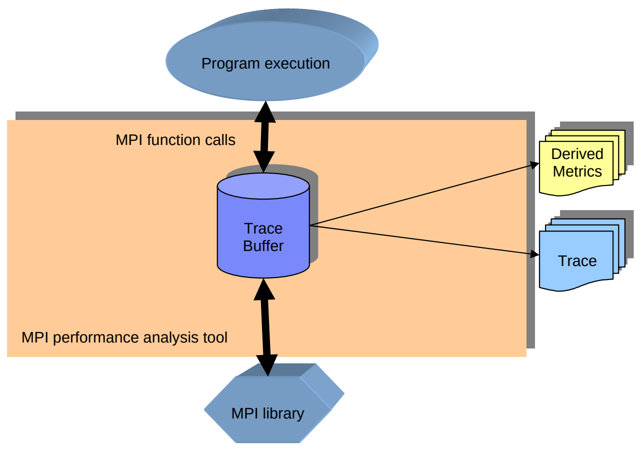
Exploration of above factors in the development process of a performance analysis tool would help the tool to achieve multiple goals in usability, efficiency, helpfulness, and finally, scalability. Particular, it would help to define the kind of information to be collected. By only collecting "useful" information, the overhead caused by the performance analysis tool can be reduced dramatically. And eventually, the overall performance analysis/tuning process would be improved significantly.

5.2 Implementation

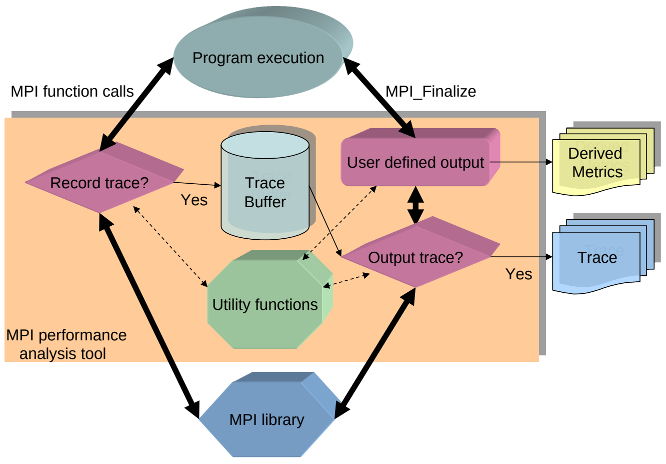
According to the design considerations in Section 5.1, we developed a configurable MPI tracing library based on the existing IBM HPC Toolkit described in Section 3.1.

The work flow of a typical MPI tracing tool, the MP_Tracer library in the IBM HPC Toolkit, is generalized as Figure 7(a). In this approach, each invocation of MPI functions is replaced with a call to a routine in the MPI performance tool library, which intercepts the MPI function call and do some bookkeeping in a "trace buffer" that resides in the memory. When the application execution finishes, the tracing library outputs performance/tracing data that satisfy certain pre-defined metrics.

The tools following this framework lack flexibility and often scale poorly if collecting excessive amount of tracing data.



(a) Existing



(b) Configurable

Figure 7: MPI Performance Analysis Tool Implementations

As our understanding, the primary reason for the lacking of flexibility of the framework is that the components of the tool library are all defined by the tool developer. To support certain level of flexibility, based on interaction with application developer, performance analyzer, tool developers usually put in hard-coded mechanism (from tool users' point of view) inside the library routines. There are a few direct drawbacks of the approach. First, the supported flexibility may never satisfy all users' requests. Secondly, while trying to support the "common" requirements, either the overhead of the tracing or accumulated data will grow and thereafter the tracing process does not scale.

Our approach is to open a few windows to tool users so that tool users can define, for the specific application, specific activities for the tracing/profiling process. The design of our configurable approach is illustrated in Figure 7(b). As shown in the figure, besides maintaining compatibility to the MP_Tracer library in the IBM HPC Toolkit, we augmented the previously approach with 4 components in the library (i.e., the big rectangle in the figure).

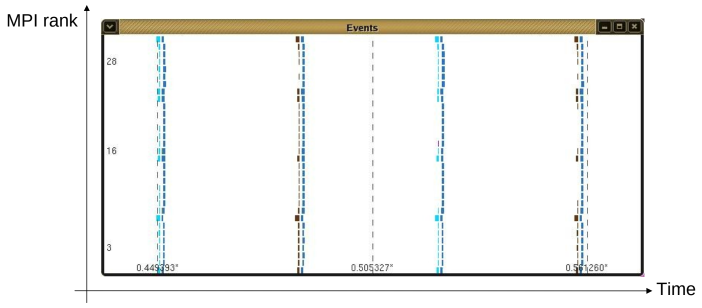
As indicated in the figure, we separate the processes for MPI_Finalize() and other MPI routines. Specifically, when a MPI routines (except MPI_Finalize()) is invoked, a function MT_trace_event() is called to decide if the specific invocation instance needs to perform any tracing activity and save the trace data into the "trace buffer". The function is implemented using weak symbol to allow the library users override the default implementation with their customized version (i.e., for beginning users, they can use default implementation without overwriting it). The functionality of MT_trace_event() corresponds to the component marked as "record trace?" in Figure 7(b). With the simply-defined and straight-forward scope of the function, it gives users a very flexible way to define various filter functions. The filter functions can range from simply returning true (as the default implementation) to perform complicated tasks such as information selection, trace-target selection, detection of specific activities and patterns, etc. Particularly, as a relative complicated example, this function can build-up a time-ordered list of the MPI routines that are called. After some training period, it could set the return value on/off to save events for one or two intervals when a repeated sequence of MPI calls is detected.

For the case when MPI_Finalize() is called, two other configurable functions are invoked. First, MT_output_trace() takes the MPI rank as input, and returns whether to output its local profiling/tracing data to a file or not. In addition, MT_output_text() is designed to handle user-defined performance metrics. The functionalities of the two functions correspond to the "output trace?" and "user defined output" components in Figure 7(b), respectively. The detailed information for those three functions is described in Appendix A.

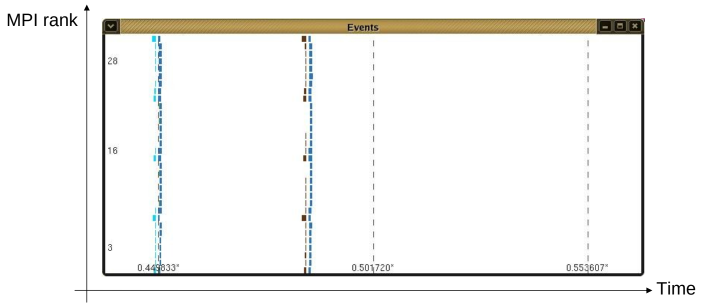
Above configurable functions provide flexible and powerful means for users to define many filtering heuristics or performing "intelligent" activities. Similarly, relatively simple implementations of the MT_trace_event() and MT_output_trace() functions can provide a great deal of selectivity in trace generation. For example, one could easily limit event records to a given time-window and/or a subset of MPI ranks, thus significantly reducing the volume of trace data.

Besides supporting above three configurable functions, our library provides a number of utility functions to help the user easily customize the three configurable functions. The detailed information for the utility functions is given in Appendix B. For example, we include utilities that return MPI message size, the number of $Hops$[7] a MPI message travels, the name of a MPI routine, caller information, etc. It also helps user to adjust tracing activity based on the dynamic usages of system resources and internal resources of the library (i.e., the trace buffer in Figure 7(b)).

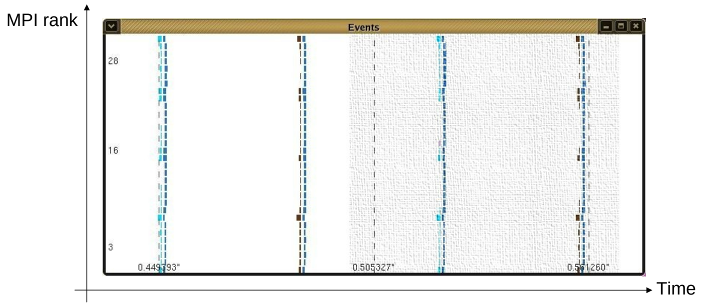
[7] $Hops$ we use is the Manhattan distance of two processors. For a pair of processors p, q with physical coordinates (x_p, y_p, z_p) and (x_q, y_q, z_q), their Manhattan distance is computed as $Hops(p,q) = |x_p - x_q| + |y_p - y_q| + |z_p - z_q|$.



(a) Original



(b) Collected

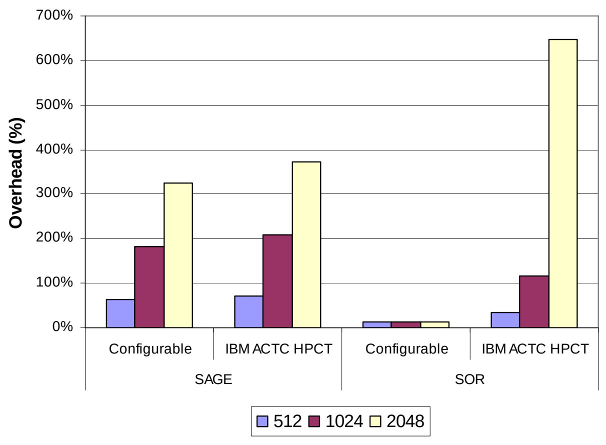


(c) Regenerated

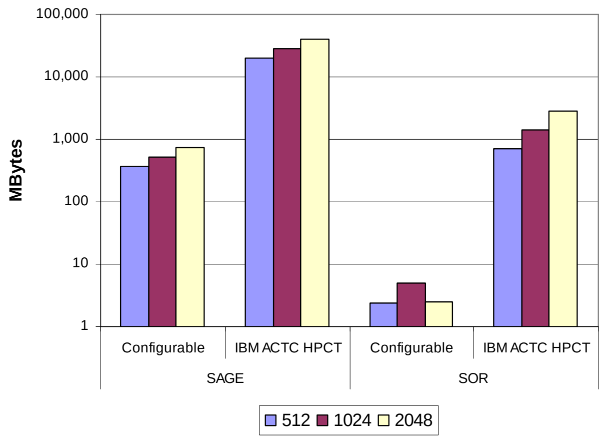
Figure 8: Trace Visualization

5.3 Evaluation

With the support of three re-writable functions, MT_trace_event(), MT_output_trace(), and MT_output_text(), users can inspect some specific run-time scenario without paying overhead for uninterested events. For instances, users can use the configurable library to collect data associated to communication hotspots. In order to do so, the user can rewrite MT_output_trace() using an utility function provided in our library, MT_get_allresults(), which only lets the node having the maximal communication time to output the trace. Users can also restrict the tracing activity to a small number of nodes that having certain physical coordinates. The library provides an utility function MT_get_layout() that returns the physical coordinates of a calling MPI process.



(a) Execution Time Overhead



(b) Performance Data Size

Figure 9: Configurable MPI Tracing Library Evaluation

```
int MT_trace_event(int id) {

  ...
  now=MT_get_time();
  MT_get_environment(&env);
  ...

  /* get MPI function call distribution */
  current_event_count=MT_get_mpi_counts();

  /* compare MPI function call distribution */
  comparison_result
  =compare_dist(prev_event_count,current_event_count);

  prev_event_count=current_event_count;
  /* compare MPI function call distribution */
  if(comparison_result==1)
  return 0; /* stop tracing */
  else
  return 1; /* start tracing */
}

int MT_output_trace(int rank) {

  if (rank < 32)
  return 1; /* output performance data */
  else
  return 0; /* no output */
}
```

Figure 10: Pseudo Code for MPI Tracing Configuration

The usability of our configurable library is rather broad and users can customize the library to perform complicated tasks. To demonstrate such usage, we implemented a simple pattern detection function to automatically capture the repeating sequence of MPI calls. Repeated invocations of MPI routines occur frequently in codes that simulate time-dependent phenomena. We re-write the MT_trace_event() and MT_output_trace() routines with about 50 lines of code as shown in Figure 10 (and use the default version of MT_output_text()). The function automatically detects the communication pattern and shuts off the recording of trace events after the first instance of the pattern. Also only MPI ranks less than 32 will output performance data at the end of program execution. As shown in the figure, utility functions such as MT_get_time() and MT_get_environment() help the user easily obtain information needed to configure the library. In this example, MT_get_time() returns the execution time spent so far and MT_get_environment() returns the process personality including its physical coordinates and MPI rank.

We tested this method using the SOR code, which iteratively solves the Poisson equation, using a repeating sequence of computations and communications. The time-dependent trace data for this case is shown in Figure 8 for a relatively small number of MPI ranks (32). The x-axis corresponds to elapsed time, and the y-axis represents MPI rank. In the graph, small rectangles indicate MPI function calls, and the background represents computation. Figure 8(a) shows the first four time-step iterations for the case of using the standard tracing method to record all events. Figure 8(b) shows the same iterations for using the customized method that recognizes repeating patterns automatically. For this case, the customized MT_trace_event() routine turns off tracing after the second wave of repeated MPI events. We have augmented the IBM HPCT trace visualization tool to re-generate the traces associated to the repeated communications that are cut-off. Figure 8(c) shows the automatically re-generated iterations (gray background area), which is almost identical to that in Figure 8(a). Thus, information is not lost because of cutting-off the recording of repeated communications.

Figure 9 shows the performance impact of exploring our configurable library, i.e., using the customized MT_trace_event(), on two applications. We use the same performance metrics described in Section 4.2: execution time with and without using tracing/profiling libraries, and the size of tracing/profiling data generated when using the libraries. For reference, we use the results of IBM ACTC HPCT performance tracing tool[8], which, as shown in Section 4.2, has similar overheads on execution time and tracing data to other performance tracing tools.

As shown in the graphs, the performance impact on SOR is much significant. For the execution time overhead, not only using the configurable library introduces extreme low (about 10%) overhead, but also the overheads keep the same when the number of processors increases. Similarly, in terms of extra tracing data collected and saved by the tracing process, when comparing to that of IBM ACTC HPCT tracing tool, using the configurable library gives negligible overhead, and it does not grow when the number of processors increases. The results on SOR give us strong evidence that if using our configurable library properly, highly scalable communication tracing can be achieved. Because the customized MT_trace_event() explores the repeated communications, the useful trace information is not jeopardized.

While the results on SOR are significant, the results on SAGE is rather moderate. Specifically, in term of execution time overhead, using the configurable library shows smaller improvement over that of the IBM ACTC HPCT tracing library. This is because that SAGE uses adaptive mesh refinement, and therefore there are not many repeated communication patterns captured by our customized MT_trace_event() routine. Nevertheless, the reduction of trace data is still significant, i.e., about 40 folds.

Overall, our experiment on exploring scalability of the configurable tool indicates that with good knowledge of the application under tuning and proper definitions of the three re-writable functions, scalable performance tracing is realizable. While the communication patterns explored in our particular example is fairly simple, our tool allows more complex and advanced methods to be defined within the framework to achieve scalable tracing. As the usage of our configurable tool is broad, besides filtering out trace data for repeated communication operations in time domain (i.e., above example), similar on-line filtering can be applied to the space domain. Further more, one can expect that with on-line filtering for both space and time domain, the tracing overhead will be reduced significantly and the size of the collected performance data will be well-controlled.

[8]Unimproved version as described in Section 3.1: it will generate output files for all collected performance data.

5.4 Discussion

For applications where the invocation patterns of the MPI functions change through the time, our configurable library provides a effective solution for the performance analysis. The MPI activities in those applications change along the execution time. By using the utility functions provided in the configurable library, users can easily control the performance data collection.

For example, users may have nodes with specific MPI ranks or within certain spatial domain of the physical topology to collect MPI tracing data, for a set of pre-selected MPI functions, in some given time periods. Work through the flexible and powerful user interface of our library, tool users have much flexible and better control on the collection of performance data. In other words, different from traditional tracing tools that collect and save all the performance data, our configurable library enables information interested by the users to be collected and saved, with minimal overhead. A lesson that we learned when using profiling/tracing tools on BG/L is that instead of collecting all performance information from all nodes, it is suggested to use profiling tool to collect summarized performance information first. For detailed traces, collecting a fixed number (e.g., 256) of MPI ranks will make the analysis easier since the data size is manageable.

The overhead for exploring configurability depends on the complexity of the configuration functions provided by the users and the density of MPI function calls in an application. The overhead corresponds to possible performance degradation of using re-configured functionalities, compared to that using the default implementation of the reconfigurable functions, depends on the user and the application. If users configure the library with rather complicated methods (e.g., containing a large number of calls to utility functions) and/or the application containing many calls to MPI functions, the overhead of configuration can be significant. In our experiment, the overhead associated to the configurable functions is much lower than that associated to the generation and saving of trace data, and thus is negligible.

6 Related Work

Dyninst [Buck and Hollingsworth 2000; dyn] provides a C++ class library for program instrumentation. Using this library, it is possible to instrument and modify application programs during execution. A unique feature of this library is that it permits machine-independent binary instrumentation programs to be written. Numerous projects including TAU are utilizing this library. Paradyn [Miller et al. 1995; par a], based on Dyninst, is a performance measurement tool for parallel and distributed programs. Performance instrumentation is inserted into the application program and modified during execution. The instrumentation is controlled by a Performance Consultant module. The Performance Consultant has a well-defined notion of performance bottlenecks and program structure, so that it can associate bottlenecks with specific causes and specific parts of a program. The instrumentation overhead is controlled by monitoring the cost of its data collection.

A configurable performance analysis tool is an important basis towards automatic performance tuning. Software like pSigma [Sbaraglia et al. 2004] can utilize the tool to minimize human intervention. pSigma is an infrastructure for instrumenting parallel applications. It enables the users to probe into the execution of an application by intercepting its control-flow at selected points. The configurable MPI tracing library we developed will be utilized by the pSigma in the near future.

The configurable MPI tracing library we developed focuses on MPI communication, a specific target in supercomputing. Performance tools mentioned here target on general framework for performance tuning for parallel application. In addition, the dynamic-instrumentation based tools, e.g. Dyninst and Paradyn, usually require at least one additional process for each normal computation process. This conflicts with Blue Gene/L's computational model, where each compute node only runs single computation process, and in turn may hinder their capability to scale.

7 Conclusion

Through an extensive, realistic empirical study on the scalability bottlenecks of tools that have been ported onto BG/L, we identified that handling and storing of ever increasing communication trace data being the most significant factor. Based on the study, we described a configurable approach that provides much flexible control of trace generation. By providing a small number of user-rewritable functions and a set of utilities functions, tool users can dictate much sophisticated, efficient, and eventually scalable performance tracing. We presented an example for the exploration of the configurability of our performance tracing approach. By configuring the tracing library to detect and utilize the repeated communication patterns, we show that our configurable library essentially facilitates highly scalable tracing of parallel applications.

At the time of this writing, tools like TAU, KOJAK and Paraver are also working to reduce the size of the performance data generated. We plan to followup this study with more comparisons relating to scalability. In the near future we also plan to further extend this approach to other performance analysis utilities, such as hardware performance counter libraries and tools for I/O activity tracing.

Acknowledgment

The authors would like to thank Manish Gupta, David Klepacki, and Jose Castanos for their supports and reviews.

References

IBM Advanced Computing Technology Center MPI Tracer/Profiler.

ADIGA, N., ALMASI, G., ALMASI, G., ET AL. 2002. An Overview of the Blue Gene/L Supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, 1–22.

ADIGA, N. R., BLUMRICH, M. A., CHEN, D., ET AL. 2005. Blue Gene/L Torus Interconnection Network. *IBM Journal of Research and Development 49*, 2/3, 265–276.

ALMASI, G. S., BEECE, D., BELLOFATTO, R., ET AL. 2002. Cellular Supercomputing with System-on-a-Chip. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 152–153.

ALMASI, G., ARCHER, C., CASTANOS, J. G., ET AL. 2003. MPI on Blue Gene/L: Designing an Efficient General Purpose Messaging Solution for a Large Cellular System. In *Proceedings of the 10th European PVM/MPI Users Group Meeting*, 252–261.

ALMASI, G., ARCHER, C., CASTANOS, J. G., ET AL. 2005. Design and implementation of message-passing services for the Blue Gene/L supercomputer. *IBM Journal of Research and Development 49*, 2/3, 393.

ALMASI, G., BHANOT, G., GARA, A., ET AL. 2005. Scaling physics and material science applications on a massively parallel blue gene/l system. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, ACM Press, New York, NY, USA, 246–252.

APPELBE, B., MAY, D., MORESI, L., QUENETTE, S., TAN, S.-C., AND WANG, F. Snark - a reusable framwork for geoscience computational models. *Submitted to Pure and Applied Geosciences*.

Appelbe, B., May, D., Quenette, S., Tang, S., Wang, F., and Moresi, L. 2002. Towards rapid geoscience model development - the snark project. In *Proceedings of 3rd ACES (APEC Cooperation for Earthquake Simulation) Workshop*.

Arabe, J. N., Beguelin, A., Lowekamp, B., Seligman, E., Starkey, M., and Stephan, P. 1995. Dome: Parallel programming in a heterogeneous multi-user environment. Tech. rep.

Balay, S., Eijkhout, V., Gropp, W. D., McInnes, L. C., and Smith, B. F. 1997. Efficient management of parallelism in object oriented numerical software libraries. In *Modern Software Tools in Scientific Computing*, Birkhäuser Press, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds., 163–202.

Balay, S., Buschelman, K., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H., 2001. PETSc Web page.

Balay, S., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. 2004. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory.

Benson, S. J., McInnes, L. C., Moré, J., and Sarich, J. 2004. TAO user manual (revision 1.7). Tech. Rep. ANL/MCS-TM-242, Mathematics and Computer Science Division, Argonne National Laboratory.

Berman, F., and Wolski, R. 1996. Scheduling from the perspective of the application. In *HPDC '96: Proceedings of the High Performance Distributed Computing (HPDC '96)*, IEEE Computer Society, 100.

Berman, F., Chien, A., Cooper, K., et al. 2001. The GrADS Project: Software support for high-level Grid application development. *The International Journal of High Performance Computing Applications 15*, 4, 327–344.

Browne, S., Dongarra, J., Garner, N., London, K., and Mucci, P. 2000. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, IEEE Computer Society, 42.

Bryan, K. 1997. A numerical method for the study of the circulation of the world ocean. *Journal of Computational Physics 135*, 2, 154–169.

Buck, B. R., and Hollingsworth, J. K. 2000. An API for Runtime Code Patching. *Journal of High Performance Computing Applications 14*, 317–329.

Chung, I.-H., and Hollingsworth, J. K. 2004. Automated Cluster-Based Web Service Performance Tuning. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC'04)*, IEEE Computer Society, 36–44.

Chung, I.-H., and Hollingsworth, J. K. 2004. Using Information from Prior Runs to Improve Automated Tuning Systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, 30.

Ţăpuş, C., Chung, I.-H., and Hollingsworth, J. K. 2002. Active harmony: towards automated performance tuning. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, IEEE Computer Society Press, 1–11.

Devine, K., Boman, E., Heaphy, R., Hendrickson, B., and Vaughan, C. 2002. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering 4*, 2, 90–97.

Dorland, W., Jenko, F., Kotschenreuther, M., and Rogers, B. N. 2000. Electron temperature gradient turbulence. *Physical Review Letters 85* (Dec.).

Dynamic Probe Class Library.

Dukowicz, J. K., and Smith, R. D. 1994. Implicit free-surface method for the Bryan-Cox-Semtner ocean model. *Journal of Geophysics Research 99* (Apr.), 7991–8014.

Dukowicz, J. K., Smith, R. D., and Malone, R. C. 1993. A Reformulation and Implementation of the Bryan-Cox-Semtner Ocean Model on the Connection Machine. *Journal of Atmospheric and Oceanic Technology 10*, 2 (Apr.), 195–208.

An Application Program Interface (API) for Runtime Code Generation.

FLASH homepage.

Frigo, M., and Johnson, S. G. 2005. The design and implementation of FFTW3. *Proceedings of the IEEE 93*, 2, 216–231. special issue on "Program Generation, Optimization, and Platform Adaptation".

Hollingsworth, J. K., and Keleher, P. J. 1998. Prediction and adaptation in active harmony. In *HPDC '98: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society, 180.

Keleher, P. J., Hollingsworth, J. K., and Perkovic, D. 1999. Exposing application alternatives. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, IEEE Computer Society, 384.

Kerbyson, D. J., Alme, H. J., Hoisie, A., Petrini, F., Wasserman, H. J., and Gittings, M. 2001. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, 37–37.

Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks.

KOTSCHENREUTHER, M., REWOLDT, G., AND TANG, W. M. 1995. Comparison of initial value and eigenvalue codes for kinetic toroidal plasma instabilities. *Computer Physics Communications 88* (Aug.).

LAN, Z., TAYLOR, V. E., AND BRYAN, G. 2001. Dynamic load balancing of samr applications on distributed systems. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, ACM Press, New York, NY, USA, 36–36.

libMesh:a c++ framework for the numerical simulation of partial differential equations on serial and parallel platforms.

MALONY, A. D., SHENDE, S., BELL, R., LI, K., LI, L., AND TREBON, N. 2004. Advances in the tau performance system. 129–144.

MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. 1995. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer 28* (Nov.), 37–46.

MOHR, B., AND WOLF, F. 2003. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications . In *Euro-Par 2003: Proceedings of the International Conference on Parallel and Distributed Computing*.

MPICH-A Portable Implementation of MPI.

mpiP: Lightweight, Scalable MPI Profiling.

NELDER, J., AND MEAD, R. 1965. A simplex method for function minimization. *Computer Journal 7*.

NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. 1997. Agile application-aware adaptation for mobility. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, ACM Press, 276–287.

Performance Application Programming Interface.

Paradyn:Parallel Performance Tools.

PARAVER homepage.

PILLET, V., LABARTA, J., CORTES, T., AND GIRONA, S. 1995. PARAVER: A tool to visualise and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, IOS Press, Amsterdam, vol. 44, 17–31.

The Parallel Ocean Program (POP).

RIBLER, R. L., VETTER, J. S., SIMITCI, H., AND REED, D. A. 1998. Autopilot: Adaptive control of distributed applications. In *HPDC '98: Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, IEEE Computer Society, 172.

RIBLER, R. L., SIMITCI, H., AND REED, D. A. 2001. The autopilot performance-directed adaptive control system. *Future Gener. Comput. Syst. 18*, 1, 175–187.

SBARAGLIA, S., EKANADHAM, K., CREA, S., AND SEELAM, S. 2004. pSigma: An Infrastructure for Parallel Application Performance Analysis using Symbolic Specifications. In *The sixth European Workshop on OpenMP - EWOMP'04*.

SCHOLZ, W. Magpar: Parallel micromagnetics code.

SCIRUN DEVELOPMENT TEAM. 2002. SCIRun: A Scientific Computing Problem Solving Environment.

SMITH, R. D., DUKOWICZ, J. K., AND MALONE, R. C. 1992. Parallel ocean general circulation modeling. In *Proceedings of the eleventh annual international conference of the Center for Nonlinear Studies on Experimental mathematics : computational issues in nonlinear science*, Elsevier North-Holland, Inc., 38–61.

SOD, G. A. 1978. A survey of several finite difference methods for systems of nonlinear hyperbolic conservation laws. *Journal of Computational Physics 27* (Apr.), 1–31.

The ASCI sPPM Benchmark Code.

STILLINGER, F. H., AND WEBER, T. A. 1985. Computer simulation of local order in condensed phases of silicon. *Physical Review B 31* (Apr.), 5262–5271.

The ASCI sweep3d Benchmark Code.

TAU: Tuning and Analysis Utilities.

TAYLOR, V., WU, X., AND STEVENS, R. 2003. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *SIGMETRICS Perform. Eval. Rev. 30*, 4, 13–18.

VUDUC, R., DEMMEL, J. W., AND YELICK, K. A. 2005. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Institute of Physics Publishing, San Francisco, CA, USA, Journal of Physics: Conference Series.

WHALEY, R. C., AND DONGARRA, J. 1998. Automatically tuned linear algebra software. In *SuperComputing 1998: High Performance Networking and Computing*.

WHALEY, R. C., PETITET, A., AND DONGARRA, J. J. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Computing 27*, 1–2, 3–35.

WOLSKI, R. 1997. Forecasting network performance to support dynamic scheduling using the network weather service. In *HPDC '97: Proceedings of the 6th International Symposium on High Performance Distributed Computing (HPDC '97)*, IEEE Computer Society, 316.

A Configuration Functions

There are three functions that can be rewritten to configure the profiling tool. During the runtime, the return values of those three functions decide what performance information to be stored, which process (MPI rank) will output the performance information, and what performance information will be output to files.

- `int MT_trace_event(int);` The integer passed into this function is the ID number for the MPI function. The return value is 1 if the performance information should be stored in the buffer; otherwise 0.
- `int MT_output_trace(int);` The integer passed into this function is the MPI rank. The return value is 1 if it will output performance information; otherwise 0.
- `int MT_output_text(void);` This function will be called inside the `MPI_Finalize()`. The profiling tool user can rewrite this function to customize the performance data output (e.g., user-defined performance metrics or data layout).

B Utility Functions

In this section we briefly describe the utility functions that are provided to help tool users configure the three configuration functions described in Appendix A.

- `long long MT_get_mpi_counts(int);` The integer passed in is the MPI ID and the number of call counts for this MPI function will be returned.
- `double MT_get_mpi_bytes(int);` Similar to the MT_get_mpi_counts(), this function will return the accumulated size of data transferred by the MPI function.
- `double MT_get_mpi_time(int);` Similar to the MT_get_mpi_counts(), this function will return the accumulated time spent in the MPI function.
- `double MT_get_avg_hops(void);` If the distance between two processors p, q with physical coordinates (x_p, y_p, z_p) and (x_q, y_q, z_q) is calculated as $Hops(p,q) = |x_p - x_q| + |y_p - y_q| + |z_p - z_q|$. We measure the *AverageHops* for all communications on a given processor as follows:

$$AverageHops = \frac{\sum_i Hops_i \times Bytes_i}{\sum_i Bytes_i}$$

where $Hops_i$ is the distance between the processors for ith MPI communication and $Bytes_i$ is the size of the data transferred in this communication. The logical concept behind this performance metric is to measure how far each byte has to travel for the communication (in average). If the communication processor pair is close to each other in the coordinate, the *AverageHops* value will tend to be

- `double MT_get_time(void);` This function returns the time since MPI_Init() is called.
- `double MT_get_elapsed_time(void);` This function returns the time between MPI_Init() and MPI_Finalize() are called.
- `char *MT_get_mpi_name(int);` This function takes a MPI ID and returns its name in a string.
- `int MT_get_tracebufferinfo(struct );` This function returns the size of buffer used/free by the tracing/profiling tool at the moment.
- `int MT_get_memoryinfo(struct );` This function returns the information for the memory usage on the compute node.
- `int MT_get_calleraddress(void);` This function will return the caller address in the memory.
- `int MT_get_callerinfo(int, struct );` This function takes the caller memory address and returns detailed caller information including the path, the source file name, the function name and the line number of the caller in the source file.
- `void MT_get_environment(struct );` This function returns its self environment information including MPI rank, physical coordinates, dimension of the block, number of total tasks and CPU clock frequency.
- `int MT_get_allresults(int, int, struct);` This function returns statistical results (e.g., min, max, median, average) on primitive performance data (e.g., call counts, size of data transferred, time...etc.) for specific or all MPI functions.