UCRL-CONF-215232

LAWRENCE LIVERMORE NATIONAL LABORATORY

Scalable Dynamic Instrumentation for BlueGene/L

Martin Schulz, Dong Ahn, Andrew Bernat, Bronis R. de Supinski, Steven Y. Ko, Gregory Lee, Barry Rountree

September 9, 2005

Workshop on Binary Instrumentation and Applications
St. Louis, MO, United States
September 18, 2005 through September 18, 2005

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

Scalable Dynamic Binary Instrumentation for Blue Gene/L

Martin Schulz[1], Dong Ahn[1], Andrew Bernat[2], Bronis R. de Supinski[1],
Steven Y. Ko[3], Gregory Lee[4], Barry Rountree[5]

[1]Lawrence Livermore National Laboratory, Livermore, CA, USA
{schulzm,ahn1,bronis}@llnl.gov
[2]University of Wisconsin, Madison, WI, USA, bernat@cs.wisc.edu
[3]University of Illinois, Urbana-Champaign, IL, USA, sko@cs.uiuc.edu
[4]University of California, San Diego, CA, USA, gllee@cs.ucsd.edu
[5]University of Georgia, GA, USA, rountree@cs.uga.edu

Abstract

Dynamic binary instrumentation for performance analysis on new, large scale architectures such as the IBM Blue Gene/L system (BG/L) poses new challenges. Their scale—with potentially hundreds of thousands of compute nodes—requires new, more scalable mechanisms to deploy and to organize binary instrumentation and to collect the resulting data gathered by the inserted probes. Further, many of these new machines don't support full operating systems on the compute nodes; rather, they rely on light-weight custom compute kernels that do not support daemon-based implementations.

We describe the design and current status of a new implementation of the DPCL (Dynamic Probe Class Library) API for BG/L. DPCL provides an easy to use layer for dynamic instrumentation on parallel MPI applications based on the DynInst dynamic instrumentation mechanism for sequential platforms. Our work includes modifying DynInst to control instrumentation from remote I/O nodes and porting DPCL's communication to use MRNet, a scalable data reduction network for collecting performance data. We describe extensions to the DPCL API that support instrumentation of task subsets and aggregation of collected performance data. Overall, our implementation provides a scalable infrastructure that provides efficient binary instrumentation on BG/L.

1 Motivation

Dynamic binary instrumentation is an attractive technique for the implementation of performance analysis tools. It requires no modifications to the source code or the linking process and enables tools and users to adjust the type and amount of measurements during program execution. Many performance analysis tools are layered on top of DynInst [4], an API to dynamically insert arbitrary code snippets into running applications, and using DynInst, several projects target dynamic instrumentation for parallel applications. This includes performance monitoring infrastructures like OMIS [8] or DPCL [6], and tools like Paradyn [10] or Tool Gear [9].

When applying dynamic binary instrumentation to new, large scale architectures such as the IBM Blue Gene/L system (BG/L), we face new challenges. The scale of such machines—with potentially hundreds of thousands of compute nodes—requires new and scalable mechanisms to deploy and organize binary instrumentation and to collect the resulting data gathered by the inserted probes. Further, many of these new machines don't support full operating systems on the compute nodes; instead, they use light-weight custom compute kernels that do not support DynInst's traditional *"mutator–daemon"* model.

In this paper we describe an implementation of dynamic instrumentation on BG/L that overcomes the limitations described above. We base our efforts on the Dynamic Probe Class Library (DPCL) [6] and provide a new, DPCL compatible library for BG/L. As part of this, we modify DynInst to control instrumentation from remote daemons and we deploy a scalable data reduction network to collect the acquired performance data. Users can access this new instrumentation infrastructure using the standard DPCL class hierarchy and API, which supports existing DPCL-based tools on BG/L.

The software is currently still under development, but first results are encouraging. BG/L's debug interfaces enable an efficient instrumentation of remote compute nodes using a modified version of DynInst as well as a clean integration into BG/L's job launch and control infrastructure. Further, the performance of data transfers from the compute node through the scalable data communication mechanism is low overhead and shows good scaling behavior.

2 Blue Gene/L

Blue Gene/L is large scale parallel system developed jointly by IBM and the Lawrence Livermore National Laboratory. In its final form, it will consist of 65536 compute nodes, each with a dual-core (PowerPC 440) ASIC, resulting in an overall peak performance of 367 TFlops. All compute nodes are connected by both a torus network (for point-to-point communication) and a tree network (for collectives). A full description of the architecture is available in an SC2002 paper [3].

The machine is divided into groups of 64 nodes[1] and each node is associated with a separate I/O node responsible for all external communication including file I/O and TCP client

[1] The concrete number of nodes per I/O node is machine specific: the BG/L architecture can support between eight and 64 nodes per I/O node.

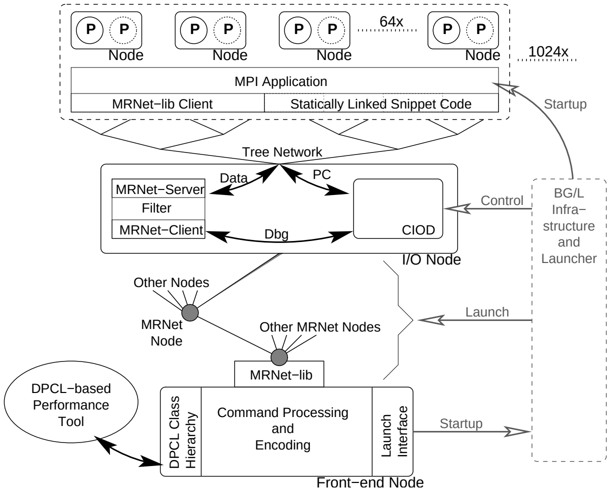

Figure 1: Architecture of DPCL on BG/L.

socket connectivity. I/O nodes, which are of the same node architecture as compute nodes, communicate with the compute nodes using the tree network and are connected to the outside world through Gigabit Ethernet. In addition to the actual system, the BG/L setup at LLNL provides 14 frontend (login and system) nodes, as well as a large disk array.

On the software side, each compute node runs a custom light weight kernel (CNK or Compute Node Kernel). This kernel does not include any support for scheduling or multi-tasking and only directly implements light-weight system calls. A larger subset of standard Unix system calls are "function shipped" to the I/O node, which executes them as a proxy for the compute node and returns the results. Since the CNK only supports one thread per core, tools as well as runtime systems can not rely on daemons in their software architecture.

On the I/O node, BG/L deploys a restricted operating system based on Linux, which provides limited multi-tasking and threading support. The central component running on every I/O node is the *CIOD*. This daemon manages the communication between frontend and compute nodes, executes system call requests from the compute nodes, and provides I/O access to applications. In addition, the *CIOD* allows tools to start and control one additional I/O node daemon, which in turn can communicate with the *CIOD* and control the compute nodes using a proprietary debugging interface provided by the *CIOD*. Debuggers, such as TotalView, use this additional daemon to implement their functionality [5].

3 Architecture

Our BG/L implementation of the DPCL interface consists of four major building blocks: the I/O node daemon support; a port of DynInst to BG/L; MRNet [11]; and an MPI job control mechanism based on BG/L's MPI debugger interface. We port DynInst to BG/L to provide remote node instrumentation. We use MRNet, which provides a scalable data collection network with data reduction capabilities, for the DPCL communication infrastructure. Finally, the debugger interface provides

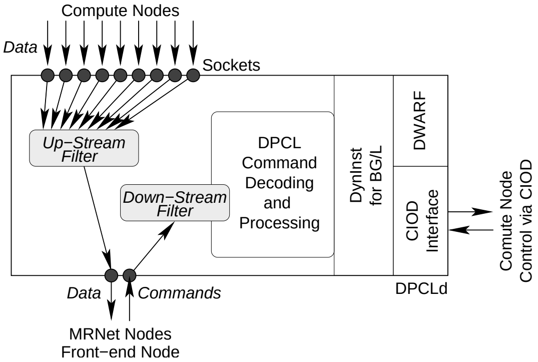

Figure 2: DPCL daemon architecture.

portable parallel job launch and attach facilities. The complete architecture is shown in Figure 1.

Tools run on one of the system's frontend nodes and access the BG/L DPCL library using the existing[2] DPCL class hierarchy. The library has access to BG/L's infrastructure through the launch interface, which it can use to start and terminate application processes on compute nodes. During job startup, the library also uses this interface to start a tool daemon on the set of I/O nodes associated with a job's partition [7].

Once both the application and daemons have been started, the system initializes a communication tree from the frontend to all participating I/O nodes. At the same time, the application connects to the communication tree through the respective I/O node (see also Figure 2). Note that the application must be transparently pre-linked with a thin communication client and potential DynInst code snippets since the BG/L CNK does not support dynamic linking.

After initialization, the tool can send instrumentation requests through the DPCL library and the MRNet network to the tool daemon on the I/O node. The daemon translates the requests into their respective DynInst calls and forwards them through BG/L's proprietary debugger interface to the compute nodes. Any data collected by the probes is sent directly to the tool daemon on the I/O node using socket communication and from there forwarded—after a potential data reduction or aggregation—to the consuming tool running on the frontend.

3.1 DynInst for BG/L

DynInst is an Application Program Interface designed to allow rewriting binary executables at runtime. Instead of instrumenting every potentially interesting point in the code, this approach allows instrumentation to be selectively inserted and removed while the program is executing, thus reducing the impact of measurement on the experiment.

DynInst first parses the binary image of the target application and provides users the capability to find potential instrumentation locations. At these spots users can then use DynInst calls to insert arbitrary code snippets. This is accomplished using trampoline code: an instruction at the appropriate location

[2] We extended the class hierarchy slightly to improve scalability; the API is, however, still backwards compatible.

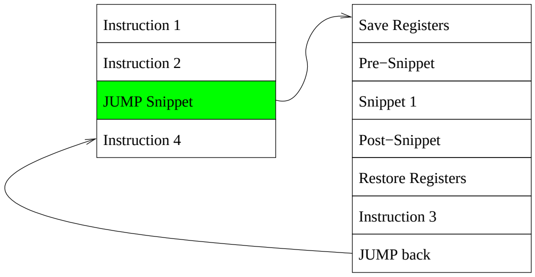

Figure 3: Inserting a code snippet using DynInst.

is replaced by a local branch to an unused space in memory where the new code has been written. The instruction that was replaced is executed as part of the patch, and the last instruction of the patch returns control back to the regular execution path. This is further illustrated in Figure 3.

A tool using DynInst is usually implemented using a separate process or daemon, called the *mutator*, and accesses the target application, the *mutatee*, using a debugging interface. On Linux, e.g., DynInst uses the *ptrace* interface and its capability to read and write address locations in other processes' address spaces to insert code snippets. Further, DynInst uses the same mechanism for process control.

Within IBM's original DPCL implementation, this *mutator* functionality is embedded into a daemon process executing on all target nodes. Such a daemon-based implementation is not possible on BG/L, since compute nodes only support one process at a time. We solved this problem by running the *mutator* process as tool daemons on all I/O nodes associated with the target application. Each *mutator* daemon controls the subset of compute nodes associated the respective I/O node. For this, we extend DynInst to control a large number of processes per DynInst instantiation and port the backend of DynInst to BG/L's proprietary debug interface, which enables the control and modification of remote processes running on the compute nodes.

The port of DynInst itself is based on a combination of the DynInst versions for Linux and PowerPC/AIX. The former is useful, since the I/O nodes run based on an embedded and stripped down version of Linux, while the latter covers most aspects of the instruction set architecture of the embedded PowerPC core.

3.2 Adapting MRNet for BG/L

The original DPCL implementation uses a socket-based communication infrastructure to connect all DPCL daemons directly to the tool running on the frontend node. The resulting star-shaped topology is, however, not scalable and hence not suited for systems like BG/L. When used on the full system, it would require 64K individual socket connections, exceeding the limit on open connections on most operating systems as well as representing a severe bottleneck.

We replaced the communication mechanism with MRNet [11], a scalable multicast/reduction network. MRNet uses a tree topology for all its communication needs (see Figure 4). The root of the tree is the consuming tool and is executed on the frontend; the leaves are the processes on all compute nodes participating in the target application; the second level nodes are implemented within the BG/L tool daemons on the I/O nodes; and the remaining internal nodes are supplementary tool daemons running either on additional frontends or other computational resources connected to the machine.

The first level of the tree topology (between the compute and the I/O node) is given by the machine architecture and the distribution of compute tasks onto the compute nodes. The topology of the remaining levels of the MRNet tree can be defined by the user or tool developer depending on the expected communication needs and the availability of external nodes.

At startup, our DPCL library gathers the topology information and creates a matching MRNet topology tree. It also launches the I/O node tool daemons and establishes connections between the compute nodes and the daemons and between the daemons and the remainder of the MRNet tree. Since the startup of the MPI application and the I/O node tool daemons is controlled by the BG/L infrastructure, we extend MRNet to connect to independently started components, including internal daemon nodes.

Once established, our DPCL implementation uses the MRNet tree to multicast commands selectively from the tool to the DynInst implementations running as part of the I/O node tool daemon, to receive acknowledgments from DynInst, and to collect the data produced by the inserted probes. For the communication with DynInst, we encode the intended command including all its parameters on the frontend into MRNet packets and send it as part of an MRNet stream. On the tool daemon side, we use a separate thread to wait for incoming packets, decode them, and invoke the matching DynInst routine. Any acknowledgment or return value is handled the same way in the reverse direction from the I/O node to the tool frontend.

Any data gathered by the inserted probes is sent from the application process to the tool frontend using a thin MRNet backend library as part of the inserted probe. To maintain backward compatibility with the original DPCL, we integrate this communication call into the existing *AIS_send* call. On the frontend node, the data streams from the compute nodes are handled by a separate thread that listens on the corresponding MRNet streams, receives all packets, and invokes the necessary tool callbacks.

3.3 Integrating with BG/L Job Start Mechanism

BG/L's job launching mechanism, which is accessed using the SLURM resource management system [2], provides the standard MPI debug interface [1]. This interface is commonly used by parallel debuggers to identify, locate and attach to the individual tasks of MPI applications.

We use the same services in our DPCL implementation, which will support portability to other systems that support the MPI debugger interface. Access to this interface is encapsulated in a generic *MPI_Application* class, similar to the

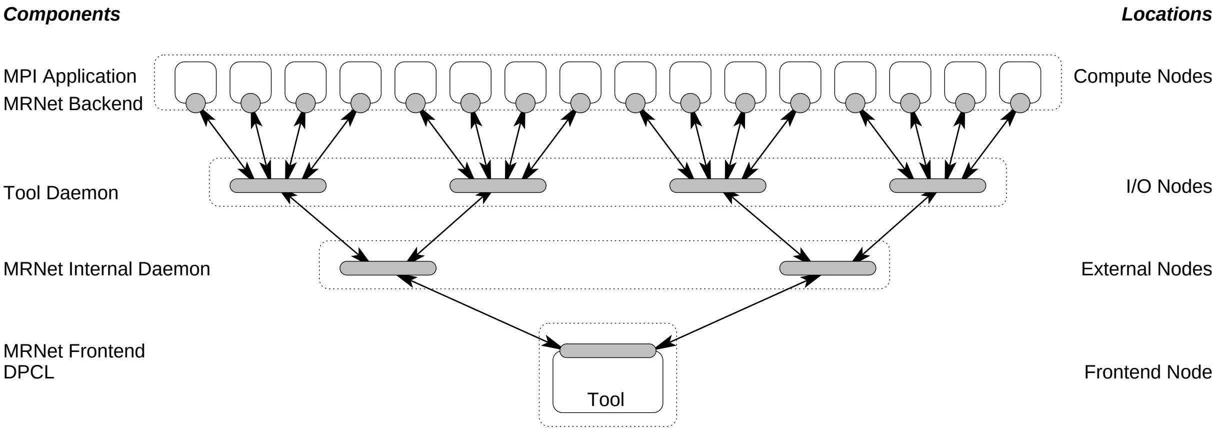

Figure 4: MRNet components and their locations.

DPCL *POE_Application* class for IBM systems. These classes provide a single abstraction for arbitrarily large MPI jobs using a single object on the tool frontend.

4 DPCL Interface Extensions

The current DPCL API lacks two concepts that are vital for an efficient implementation on large scale machines with hundreds of thousand processors: a) the ability to restrict instrumentation to a subset of nodes and b) the ability to specify data reduction filters for online data processing. In order to introduce these capabilities, we propose to extend the DPCL API with a *probe context* class.

4.1 Introducing Probe Contexts

Similar to MPI communicators, probe contexts specify subsets of nodes. Once associated with a context, a probe will then only be applied to the respective set of nodes. Further, users can associate data aggregation or reduction operations with each probe. These operations are installed as MRNet filters inside the reduction network nodes that are relevant to the probe's context when the probe is installed. All data originating from that probe is then filtered during the transfer by using the specified aggregation operation.

For downward compatibility, a default context is automatically associated with each probe when it is created. This default context comprises all nodes and specifies no data transformation, and hence behaves like a traditional DPCL probe.

4.2 Selecting Nodes

The DPCL user has two ways of creating a probe context: either a full context covering all nodes, or single context covering one rank specified as an argument to the constructor. Once created, users can use the context class's methods to copy or merge contexts, add and remove individual nodes or sets of nodes, invert node selections, or randomly sample a specified percentage of all nodes. In addition, users can specify an arbitrary function in the form of a function pointer that computes the membership based on a node's rank.

4.3 Selecting Aggregation Routines

By default, any value generated by DPCL is passed individually to the frontend through the MRNet infrastructure. Using MRNet's filter concept, values can optionally be combined on the fly while traversing the network. This reduces the amount of communication produced by the instrumentation as well as the computational requirements of the tool consuming the acquired data.

Users can specify aggregation routines as associative and commutative functions, which combine two values into one. These routines must either be predefined within MRNet (e.g., sum, average, min, max) or be part of a separate, dynamically loadable module. Once associated with a context and activated, the DPCL infrastructure dynamically loads the necessary modules into all MRNet daemons that are part of the specified context and uses the routines to combine all incoming data packets from the previous levels of the MRNet tree. The resulting data value is then passed to the next level of the tree for further processing. At the last level, the MRNet tree then passes a single value to the consuming tool.

5 Preliminary Results

Our experiments were conducted on *ubgl*, a 1024 node Blue Gene system at LLNL that functions as a development and test system. Its configuration is similar to the full BG/L except that each I/O node serves eight compute nodes (instead of 64).

Our experiments measure the cost of data transfer through our BG/L DPCL implementation. We manually instrumented an MPI application with data transfer routines that send a predefined number of messages after they receive a start command from the tool. Each message sends a number of integers, fixed for that experiment instance. We measure the time between sending the start command and completing the receive of the first message (*latency*) as well as the time between completing the receives of the first and the last message (*receive time*). We vary the number of messages, the number of nodes, the MRNet tree topology, as well as the use of aggregation routines. For all experiments we report the average of four runs.

We report measurements for messages with number of integers fixed as one, which represents a typical usage scenario for DPCL: probes send very limited amounts of data (often just consisting of a single value) to the frontend. We tested other message sizes and found trends similar to those we report here.

5.1 Communication Latency

To measure the latency for the communication between the compute nodes and the frontend, we apply the our benchmark to a single compute node. We measured a total roundtrip time of around 107.4 ms and consequently a latency of 53.7 ms. This time includes the communication from the compute node to the I/O node, the processing and forwarding of the packet within MRNet, the communication to the frontend, and the processing by the frontend MRNet library.

5.2 Receive Times

Figure 5 (left) shows the total *receive times* for varying number of messages and nodes. As expected, the time increases nearly linearly with the communication volume. In addition, the time increases with increasing node numbers, since the total number of messages received by the frontend is the product of messages and node count. To further distinguish the numbers Figure 5 (right) shows the *receive times* divided by the number of nodes. We measured the shortest times at 64 nodes, which seems to represent a sweet spot for the transfer of data from the compute nodes; smaller node counts performed significantly worse, larger number of nodes slightly worse. The latter indicates acceptable scaling properties.

5.3 Varying the MRNet Topology

In a second set of experiments, we vary the topology of the MRNet tree and hence the locations where the individual communication streams from the different nodes are merged. The results are shown in Figure 6 for runs using eight MPI tasks (left) and 64 MPI tasks (right).

The eight tasks case shows a clear trend: configurations that use fewer compute nodes per I/O node, and, thus, combine a larger number of streams at the frontend, perform better. This is not unexpected, since the computational resources on the frontend are larger. The same principle trend can also be seen for 64 tasks, although much less clearly. The performance differences between the varying topologies is reduced due to the higher pressure on the frontend, which now has to receive data for more nodes. This trend will continue for even larger partitions showing the importance of a tree-based approach.

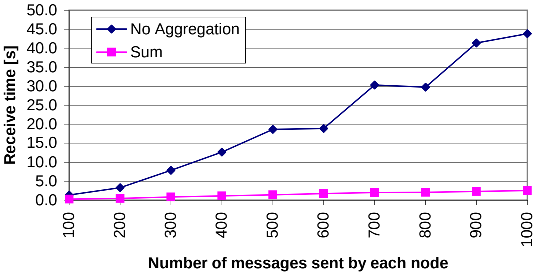

Figure 7: Receive time for varying number of messages on 256 MPI tasks with and without online data aggregation.

5.4 On-line Data Aggregation

We evaluated the impact of online data aggregation in the last set of experiments. We activated MRNet's sum filter on all MRNet nodes and compared the receive times with those from the previous runs without data aggregation. Figure 7 shows the results for 256 MPI tasks. As expected, the use of the sum filter significantly reduces the receive times due to the reduced number of messages and communication volume the frontend has to consume. Instead of 256 individual messages, which would have to be combined by the frontend, the MRNet infrastructure performs this aggregation on the fly and delivers a single, aggregated value to the frontend. The actual computational work involved in the aggregation is thereby negligible.

6 Conclusions and Future Work

Binary Instrumentation is a powerful mechanism for the implementation of performance analysis tools, but it comes with new challenges when applied in the context of large scale parallel systems. These systems often don't provide traditional operating system environments on the compute nodes and their scale demands new data collection and aggregation facilities.

We are implementing a DPCL compatible library for Blue Gene/L that tackles these new challenges. We are modifying DynInst to instrument remote application processes and we integrate MRNet [11], a scalable broadcast/reduction network. We also propose to extend the DPCL interface to include probe contexts, which enable users to select subsets of nodes for instrumentation, and to specify data aggregation operations.

Our DPCL implementation is close to being complete with most key components already working. We currently finishing the implementation of DynInst as well as the context class implementation. In the next steps, we will port existing tools using DPCL to BG/L and we will work on extending them to take advantage of the new probe context features. In addition, we plan on investigating various aggregation techniques implemented as MRNet filters and study their impact on performance as well as their usability for online performance analysis.

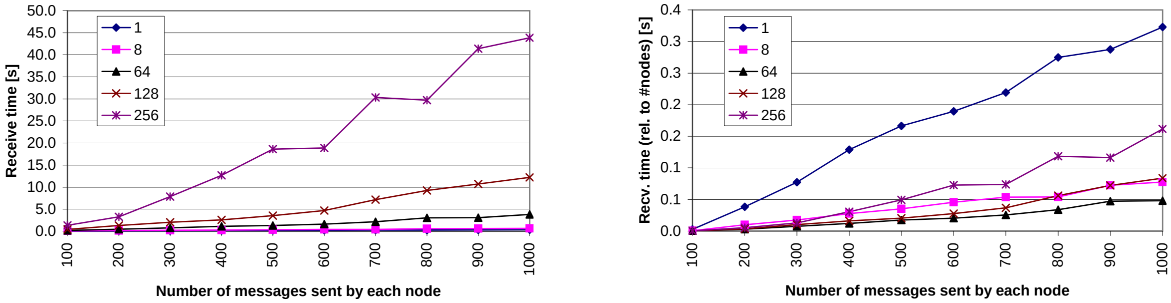

Figure 5: Receive time for varying number of messages and nodes: total (left), per node (right).

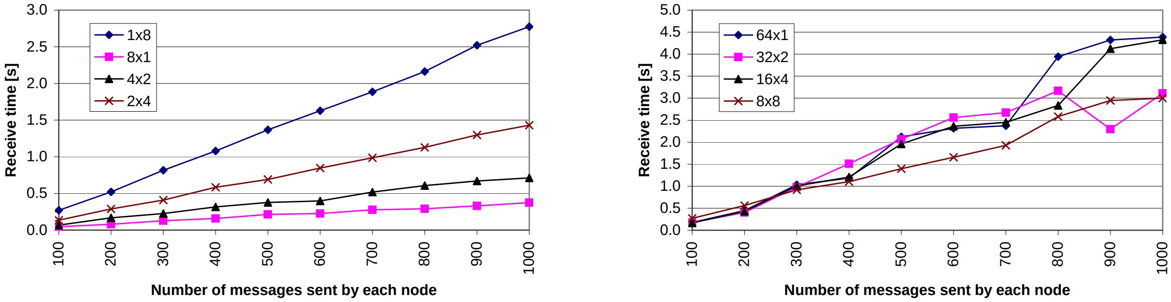

Figure 6: Receive time for varying number of messages and topologies ("number of I/O nodes"x"number of computes nodes per I/O node" : eight tasks (left), 64 tasks (right).

Acknowledgments

We would like to thank Phil Roth (Oakridge National Laboratory) for his help with MRNet and Michael Brutman (IBM) for his help with the BG/L debugger interface.

This work is supported in part by Department of Energy Grants DE-FG02-93ER25176 and DE-FG02-01ER25510, as well as the DOE NNSA ASC program and the DOE Office of Science SciDAC program PERC ISIC and was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-CONF-215232).

References

[1] MPI Debugging Interface. http://www-unix.mcs.anl.gov/mpi/mpi-debug/, Sept. 2005.

[2] SLURM: Simple Linux Utility for Resource Management. http://www.llnl.gov/linux/slurm/, June 2005.

[3] N. Adiga and et al. An overview of the bluegene/l supercomputer. In *Proceedings of IEEE/ACM Supercomputing '02*, Nov. 2002.

[4] B. Buck and J. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[5] J. DelSignore. TotalView on Blue Gene/L. Presented at "Blue Gene/L: Applications, Architecture and Software Workshop", presentation available at http://www.llnl.gov/asci/platforms/bluegene/papers/26delsignore.pdf.

[6] L. DeRose, T. Hoover, and J. Hollingsworth. The Dynamic Probe Class Library — An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, Apr. 2001.

[7] IBM. An Overview of the BlueGene/L Supercomputer. Whitepaper available at http://www-fp.mcs.anl.gov/bgconsortium.

[8] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[9] J. May and J. Gyllenhaal. Tool Gear: Infrastructure for Parallel Tools. In *Proceedings of the 2003 International Conference on Parallel and Distributed Techniques and Applications*, June 2003.

[10] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[11] P. Roth, D. Arnold, and B. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of IEEE/ACM Supercomputing '03*, Nov. 2003.