Blue Gene/L performance tools

X. Martorell
N. Smeds
R. Walkup
J. R. Brunheroto
G. Almási
J. A. Gunnels
L. DeRose
J. Labarta
F. Escalé
J. Giménez
H. Servat
J. E. Moreira

Good performance monitoring is the basis of modern performance analysis tools for application optimization. We are providing a variety of such performance analysis tools for the new Blue Gene®/L supercomputer. Those tools can be divided into two categories: single-node performance tools and multinode performance tools. From a single-node perspective, we provide standard interfaces and libraries, such as PAPI and libHPM, that provide access to the hardware performance counters for applications running on the Blue Gene/L compute nodes. From a multinode perspective, we focus on tools that analyze Message Passing Interface (MPI) behavior. Those tools work by first collecting message-passing trace data when a program runs. The trace data is then used by graphical interface tools that analyze the behavior of applications. Using the current prototype tools, we demonstrate their usefulness and applicability with case studies of application optimization.

Introduction

The Blue Gene*/L (BG/L) supercomputer is a new massively parallel system being developed by IBM in partnership with Lawrence Livermore National Laboratory (LLNL). BG/L uses system-on-a-chip (SoC) integration [1] and a highly scalable architecture [2] to assemble 65,536 dual-processor compute nodes. When operating at its target frequency of 700 MHz, BG/L will deliver 180 or 360 teraflops of peak computing power, depending on its mode of operation.

Each BG/L compute node can address only its local memory, making message passing the natural programming model for the machine. This paper discusses the current ongoing work on performance analysis tools to support the analysis of the execution of programs in BG/L. We are currently developing and porting such tools, and, at the same time, helping application programmers to port their applications to BG/L.

This paper is organized as follows: We first present a discussion of BG/L hardware, followed by a description of the implementation of the Message Passing Interface (MPI) communication library for this machine. The performance analysis tools on which we are working are introduced, followed by descriptions of the experiences and lessons learned after using our tools in a set of experiments with microbenchmarks and real applications. Related work is briefly described, and conclusions drawn.

A short discussion of Blue Gene/L hardware

The Blue Gene/L hardware [2] and system software [3, 4] have been extensively described elsewhere. In this section, we remind the reader of the hardware features most relevant to the discussion to follow.

Blue Gene/L processors: The 65,536 compute nodes of BG/L are based on a custom SoC design that integrates embedded low-power processors, high-performance network interfaces, and embedded memory. The low-power characteristics of this architecture permit very dense packaging. One air-cooled BG/L rack contains 1,024 compute nodes (2,048 processors) with a peak performance of 5.7 teraflops.

The BG/L chip incorporates two standard 32-bit embedded IBM PowerPC* 440 (PPC440) processors with private L1 instruction and data caches, a small (2-KB) L2 cache and prefetch buffer, and 4 MB of embedded dynamic random access memory (DRAM), which can be partitioned between shared L3 cache and directly addressable memory. A compute node also incorporates 512 MB of double-data-rate (DDR) memory.

Cache coherency: The standard PPC440 cores are not designed to support multiprocessor architectures: The L1

0018-8646/05/$5.00 © 2005 IBM

caches are not coherent, and the processor does not implement atomic memory operations. Software must take great care to ensure that coherency is correctly handled at the granularity of the L1 cache lines of the central processing units (CPUs)—32 bytes. This means that objects not delimited by 32-byte boundaries cannot be shared by the CPUs. To mitigate these limitations, BG/L provides a variety of custom synchronization devices in the chip, such as the lockbox (a limited number of memory locations for fast atomic test-and-sets and barriers) and 16 KB of shared static random access memory (SRAM).

Floating point: Each processor is augmented with a dual floating-point unit (FPU) consisting of two 64-bit floating-point units operating in parallel (termed a *double-hummer* FPU). The dual FPU contains two 32×64-bit register files and is capable of dispatching two fused multiply–adds in every cycle, i.e., 2.8 Gflops per node at the 700-MHz target frequency. When both cores are used, the peak performance is doubled to 5.6 Gflops.

The torus network: The torus network is the main network for user communication. Each compute node is connected to its six neighbors through bidirectional links. The 64 racks in the full BG/L system form a $64 \times 32 \times 32$ three-dimensional torus. The network hardware guarantees reliable, deadlock-free delivery of variable-length packets. Routing is done on an individual basis using one of two routing strategies: a *deterministic* routing algorithm, whereby all packets between two nodes follow the same path along the x, y, z dimensions (in this order); and a minimal *adaptive* routing algorithm that permits better link utilization but allows consecutive packets to arrive at the destination out of order.

Network efficiency: The torus packet length is between 32 and 256 bytes in multiples of 32. The first 16 bytes of every packet contain destination, routing, and software header information. Therefore, at most, 240 bytes of each packet can be used as payload. For every 256 bytes injected into the torus, 14 additional bytes traverse the wire with cyclic redundancy checks (CRCs), etc. Thus, the efficiency of the torus network is at most $\eta = 240/270 = 89\%$.

Link bandwidth: Each link delivers two bits of raw data per CPU cycle (0.25 bytes per cycle), or $\eta \times 0.25 = 0.22$ bytes per cycle of payload data. This translates into 154 MB/s/link at the target 700-MHz frequency.

Per-node bandwidth: Adding up the raw bandwidth of the six incoming and six outgoing links on each node, we obtain $12 \times 0.25 = 3$ bytes per cycle per node. The corresponding bidirectional payload bandwidth is 2.64 bytes per cycle per node.

Network reliability: The network guarantees reliable packet delivery. In any given link, it resends packets with errors, as detected by the CRC. Irreversible packet losses are considered catastrophic and stop the machine. The communication library considers the machine to be completely reliable.

Network ordering semantics: Adaptively routed network packets may arrive out of order, forcing the message layer to reorder them before delivery. Packet reordering is expensive because it involves memory copies and requires packets to carry additional sequence and offset information. On the other hand, deterministic routing leads to more network congestion and increased message latency, even on lightly used networks.

CPU/network interface: The torus network is mapped into user-space memory. Packets are read and written with the special 16-byte single-instruction multiple-data (SIMD) load-and-store instructions of the custom FPUs. These require that memory accesses be aligned to a 16-byte boundary. The communication software does not have control over the alignment of user buffers. In addition, the sending and receiving buffer areas can be aligned at different boundaries, forcing packet realignment through memory-to-memory copies.

Hardware performance counters: The CPU core used in the system has no hardware performance analysis capabilities. Instead, performance counters have been implemented as a separate unit of the die, the universal performance counter (UPC) unit. Further, the double-hummer FPU has its own performance counters. As a consequence of the design, hardware performance counters are available for a large number of events, with the exception of events internal to the CPU cores.

The UPC unit consists of 16 control registers used to manage the behavior of 48 32-bit counter registers. In total, 311 UPC events are available, exposing the behavior of all aspects of the BG/L die outside the CPU cores. This includes the prefetch unit, the L3 cache controller, and the collective and the torus network controllers. Additionally, one control register in each of the double-hummer units manages two counter registers for events related to this unit. Finally, a 64-bit timestamp register is available. The timestamp register can be read by user-level code, while the UPC and FPU registers are available only in privileged mode.

The UPCs can be individually controlled to count the rising or falling edge of an event, or the duration (in CPU cycles) of an event state being either active or inactive. The UPC counters can individually be set to generate interrupts on user-selectable count thresholds. The FPU counters are divided into one floating-point arithmetic operation counter and one load/store counter. Each FPU counter is user-programmable to count the occurrence of a subset of operations, such as, for example, arithmetic trinary operations and quadword stores.

Architecture of Blue Gene/L MPI

The BG/L MPI is an optimized port of the MPICH2 [5] library, an MPI library designed with scalability and portability in mind. **Figure 1** shows two components of the MPICH2 architecture: message passing and process management. MPI process management in BG/L is implemented using system software services. We present the architecture of the message-passing component as it is relevant to the performance analysis tools.

The upper layers of the message-passing functionality are implemented by MPICH2 code. MPICH2 provides the implementation of point-to-point messages, intrinsic and user-defined data types, communicators, and collective operations, and it interfaces with the lower layers of the implementation through the Abstract Device Interface Version 3 (ADI3) layer [6]. The ADI3 layer consists of a set of data structures and functions that have to be provided by the implementation. In BG/L, the ADI3 layer is implemented using the BG/L message layer, which in turn uses the BG/L packet layer.

ADI layer

The ADI layer is described in terms of MPI requests (messages) and functions to send, receive, and manipulate these requests. The BG/L implementation of ADI3 is called *bgltorus*. It implements MPI requests in terms of message-layer messages, assigning one message to every MPI request. Message-layer messages operate through callbacks. Messages corresponding to send requests are posted in a send queue. When a message transmission is finished, a callback is used to inform the sender. Correspondingly, there are callbacks on the receive side to signal the arrival of new messages. The callbacks match incoming message-layer messages to the list of MPI posted and unexpected requests. This implementation is the equivalent for BG/L to that usually implemented in CH3 over sockets in Transmission Control Protocol/Internet Protocol (TCP/IP) networks.

BG/L message layer

The BG/L message layer is an active message system [7–10] that implements the transport of arbitrarily sized messages between compute nodes using the torus network. It can also broadcast data using special torus packets that are deposited on every node along the route they take. The message layer breaks messages into fixed-size packets and uses the packet layer to send and receive the individual packets. At the destination, the packets may arrive out of order, and the message layer is responsible for reassembling them into a message. The software structure of the message layer is shown in **Figure 2**.

The message layer addresses nodes using the equivalent of `MPI_COMM_WORLD` ranks. Internally, it translates these ranks into physical torus x, y, z coordinates, which are used by the packet layer. The mapping of ranks to torus coordinates is programmable by the user and can be used to optimize application performance by choosing a mapping that supports the logical communication topology of the application.

Message transmission in the message layer is implemented using one of multiple available

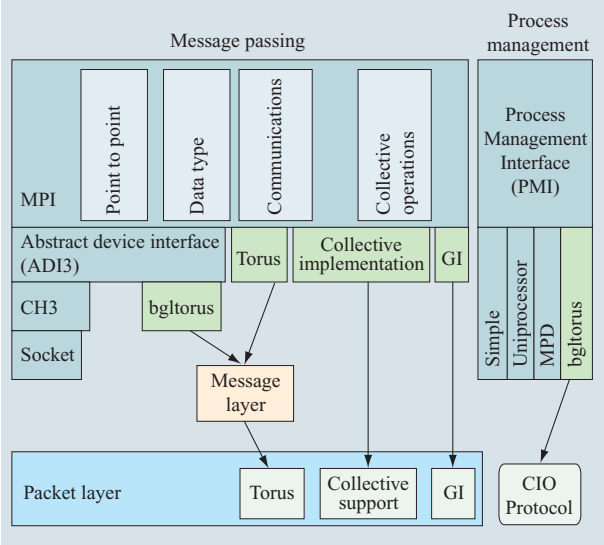


Figure 1

Blue Gene/L MPI software architecture. (GI = global interrupt; CIO = Control and I/O Protocol. CH3 is defined as the primary device distributed with MPICH2 for communication; MPD = multipurpose daemon.)

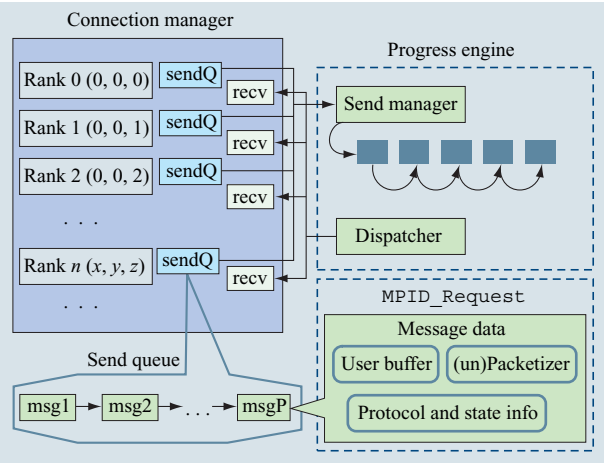


Figure 2

Blue Gene/L message layer software architecture.

communication protocols, roughly corresponding to the protocols present in more conventional MPI implementations, such as the eager and rendezvous protocols.

The message layer is able to handle arbitrary collections of data, including noncontiguous data descriptors described by MPICH2 data loops. The message layer incorporates a number of complex data packetizers and unpacketizers that satisfy the multiple requirements of 16-byte aligned access to the torus, arbitrary data layouts, and zero-copy operations.

Packet layer

The packet layer is a very thin stateless layer of software that simplifies access to the BG/L network hardware. It provides functions to read and write the torus and collective hardware, as well as to poll the state of the network. Torus packets typically consist of 240 bytes of payload and 16 bytes of header information. Collective packets consist of 256 bytes of data and a separate 32-bit header. To help the message layer implement zero-copy messaging protocols, the packet layer provides convenience functions that allow software to "peek" at the header of an incoming packet without incurring the expense of unloading the whole packet from the network.

PMI component

The Process Management Interface (PMI) component for process management is also implemented on top of the bgltorus in BG/L. In this case, the bgltorus component provides the capability to load the application from the input/output (I/O) nodes into the compute nodes using the Control and I/O (CIO) Protocol over the collective network.

Performance analysis tools for BG/L

Several parallel applications are currently being ported to BG/L; in the near future, the performance of these applications running on BG/L will require analysis [11].

HPM

We ported libHPM [12, 13] to run on the BG/L system simulator (BGLsim; see the next section). This port was done by extending the library to use the BGLcounters application program interface (API), adding support for new hardware counters and derived metrics that are related to the BG/L architecture, such as the two-element vector FPU, and by exploiting the possibility of counting both at user mode and at supervisor mode during the same execution of the program.

BGLsim

We have used a pseudo cycle-accurate simulator based on BGLsim, an architecturally accurate complete system simulator for parallel machines [13–15]. BGLsim exposes all key features of the hardware, including processors, FPUs, caches, memory, interconnection, and other supporting devices. This approach allows the user to run complete and unmodified code, from simple self-contained executables to full Linux** images. The simulator supports interaction mechanisms for inspecting detailed machine state, thus providing monitoring capabilities beyond what is possible with real hardware. BGLsim was developed primarily to support the development of system software and application code in advance of hardware availability. It can simulate multinode BG/L machines, but we restrict our discussion in this paper to the simulation of a single BG/L node system.

The BG/L pseudo cycle-accurate simulator [15] offers higher performance than traditional cycle-accurate simulators. Our model runs 100 to 1,000 times faster than a cycle-accurate simulator. The idea behind the pseudo cycle-accurate simulator is to attribute timestamps for all relevant processor resources (such as registers, internal pipelines, FPUs, memory subsystem, etc.); the model checks all of the operand dependencies, updating the corresponding timestamps. Although this is not 100% accurate because the queuing effects on memory buses are ignored, the obtained accuracy (error smaller than 15% compared with the hardware) is enough to validate most optimizations.

BGLperfctr

The large number of available events in the BG/L CPU design and the rather complex mapping of events onto possible physical counters is handled through a user-level API, BGLperfctr. This API includes a set of predefined mnemonics for each available event and provides the user with an abstraction of 52 counters, unifying the UPC and FPU counters and extending them to 64-bit counters.

Since the system design is based on a single active thread per CPU, the bookkeeping of occupied compared with free counter registers is all provided in this API. The setup of the counters is transaction-based in that the user registers a number of intended changes to the running register configuration through the API. If no collisions are detected, these changes are committed through a separate call to the library, which finally results in a kernel invocation in which the content in the affected control registers is modified and the counters start counting the desired events.

The API has full support for all capabilities of the UPC counters and offers simplicity to the end user, such as the ability to generate interrupts at an arbitrary count threshold and find an appropriate free counter for each user-selected event.

PAPI

As most APIs intend to expose a maximum of capabilities of the hardware counter design to the end user, BGLperfctr has the disadvantage of being system-specific. Writing application code that uses hardware counter information is generally a highly nonportable task. The high cost of maintaining such codes is addressed by PAPI, the Performance counter API [16]. This specification provides a platform-independent interface to control and read hardware counters on a large variety of platforms widespread in high-performance computing. The API provides mechanisms for portably naming commonly used events, setting up sets of events, and starting, stopping, and reading these events. BG/L provides a functional implementation of PAPI using the BGLperfctr abstraction of events as its implementation substrate.

An interesting aspect of the implementation of PAPI on BG/L, when compared with other platforms, is the large number of events unique to this particular system. All 319 events defined on BG/L can be programmed using PAPI through its so-called "native events" interface. In the BG/L PAPI implementation, a native event is described by its BGLperfctr mnemonic and a bit pattern describing the counting behavior requested (rising edge, falling edge, duration high, or duration low). To the extent possible, existing BG/L hardware events have been mapped to PAPI predefined event names. However, several events typically available on other platforms are not available in the BG/L hardware. These include events related to the CPU core internals, such as instructions completed, branch prediction information, and level 1 cache events. Although such events are technically possible to count in the BG/L simulator framework [15], the PAPI implementation of BG/L has not incorporated such events into its event map.

Paraver

Paraver [17] is a parallel program visualization and analysis tool that supports both shared and distributed memory applications. Paraver has three major components:

- *Tracing facility:* For MPI, a library called MPItrace is used to collect traces of the application during execution. This library intercepts the calls to the MPI primitives and records events, generating a single file for every process involved in the application. In addition, this tool can collect hardware performance counters that appear as Paraver events in the trace.
- *Trace merger:* The individual trace files are then merged into a single Paraver trace file/ citeparavertrace, using the mpi2prv tool.
- *Visualizer:* Paraver traces are visualized using the Paraver tool, which allows the visualization of the information collected and derives new metrics from it.

The existing MPItrace package has been ported to the BG/L. The package uses the PAPI interface to obtain hardware counter values and emit them into the trace file. Until now, work has focused on the basic functionality of the tool and its use to understand the behavior of different Linpack and MPI library implementations.

Scalability of the tool is one of the areas to which major effort will be devoted in the future. As of this writing, we have been able to obtain and analyze traces on up to 1,024 processors.

BGLnodes

We are developing a simple tool to display how a scalar value varies over a BG/L partition. This tool is being fed by a text file containing the coordinates of the BG/L nodes and the value to be represented for each one. Values are translated to colors, each color indicating an intensity of the value. That way, it is very easy to represent in three dimensions the values of various performance counters or other metrics derived using Paraver, showing where the hot spots are in the BG/L partition.

Experiences and lessons learned

BGLsim

In this subsection, we present two different experiences carried on the BGLsim simulator analyzing the performance of the BG/L processors. These experiments were executed in the simulator and in the first version of the hardware chip, running at 500 MHz.

IS benchmark case study

In this case study, we present a performance improvement made on the NASA Advanced Supercomputing (NAS) Integer Sort (IS) Benchmark [18] (serial version, class S) using our set of tools. The IS benchmark performs an integer sort. The goal here was to find a bottleneck in IS with enough resolution to enable optimizations that would lead to performance improvements.

To find the bottleneck in the program, we started by using the BG/L version of libHPM [13]. We reduced the number of iterations in the benchmark to one and instrumented the `rank()` function, which is called at each pass of the loop, in order to decompose its effects on the cycle count. We did this by inserting `hpmStart()` and `hpmStop()` calls around each of the `rank()` regions to identify which of them were the heaviest contributors. By doing this, we identified that two particular regions of the `rank()` function were responsible for 84% of the total

Table 1 Loops 1 and 2 iteration execution times (cycles).

Iteration	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*
Loop 1	44	17	17	17	17	17	17	17	44	17	17	17	17	17	17	17	19	17	17	17	17	17	17	17	19
Loop 2	79	50	23	52	23	50	50	50	79	23	52	50	52	52	52	50	27	25	50	52	27	27	50	23	54

Table 2 Execution times for loop 2 iterations before and after optimization.

Iteration	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*	*23*	*24*
Original	50	23	52	23	50	50	50	79	23	52	50	52	52	52	50	27	25	50	52	27	27	50	23	54
Optimized	50	21	48	21	48	50	50	48	21	48	50	50	50	25	48	21	48	50	48	23	50	23	21	50

number of cycles for that iteration. These regions were the "copy keys into work array," which is referred to as loop 1, and the "count key population," which is referred to as loop 2.

Loop 1 has a simple operation in the form

```
for ( i = 0; i < NUM_KEYS; i++ )
    key_buff2[i] = key_array[i];
```

Loop 2, on the other hand, has a double-referenced operation in the form

```
for ( i = 0; i < NUM_KEYS; i++ )
    key_buff_ptr[key_buff2[i]]++;
```

For class S, `NUM_KEYS` is 2^{16}, which means that there are 2^{16} loads and as many stores in loop 1, which is responsible for approximately 30% of the total number of cycles of the whole iteration. Loop 2 performs two loads and one store instruction and is responsible for approximately 53% of the total number of cycles.

Knowing the total number of cycles required by one loop to execute, we can estimate the number of cycles consumed by each iteration. However, this is just an average value, and we know that actual iterations may vary from one another. To find out what was going on inside these loops and to be able to optimize them, we obtained an instruction-level trace of the execution that shows every instruction performed by the CPU along with its timestamp. From this trace, we obtained the actual number of cycles each iteration of the loop takes to run (**Table 1**).

It is clear that loop 1 has a very regular and predictable behavior, taking 17 cycles at each iteration, with the exception of the first iteration in each block of eight iterations, where it takes 44 or 19 cycles. This can easily be explained by the fact that the cache size for the BG/L machine is 32 bytes, which means that eight integers fit into one cache line. The first element to be loaded forces a miss in the cache and takes a longer time, while the others have a very predictable behavior. Note that even when a miss occurs, the number of cycles it takes is not always the same. This is due to the L2 prefetching that occurs after the second L1 cache miss. The 19 cycles are explained by a misprediction by the branch predictor; at the last time the loop is executed, the branch is not taken, which increases the instruction fetch latency by two cycles.

Loop 2 has such an unpredictable behavior because it makes a load using a random index to the `key_buff_ptr` array, which is stored in the `key_buff2` array. The next index in the sequence (that is, the next value in the `key_buff2` array), is likely to be in cache. The next element in the `key_buff_ptr` array, however, is not, due to the inherent randomness of the index array. This leads to a high cache-miss probability, which results in the noticeably higher cycle times loop 2 takes at each iteration.

It is possible to optimize this kind of memory access by using an explicit prefetching technique that consists of loading the next element of `key_buff_ptr` one iteration before it is going to be used, therefore hiding its load latency. We implemented this as

```
prefetch = key_buff2[0];
for ( i = 0; i < NUM_KEYS - 1; i++ ){
    index = prefetch;
    prefetch = key_buff2[i + 1];
    key_buff_ptr[index]++;
}
index = prefetch;
key_buff_ptr[index]++;
```

The results of this optimization can be seen clearly in **Table 2** and in **Figure 3**, where we observe that the higher cycle-count spikes have disappeared and that a lower baseline has been set. Furthermore, there was considerable improvement in the IS benchmark overall performance: The original main loop (one execution of

the rank function) took 0.007553 seconds to be executed, while the optimized version executed in only 0.006233 seconds, an improvement of 17.5%. The total number of Mops measured by IS also jumped from 25.78 to 31.12.

DGEMV case study

DGEMV is the name of a subroutine that performs the matrix–vector operation $y = \alpha \cdot A \cdot x + \beta \cdot y$, where α and β are scalars, x and y are vectors, and A is an $M \times N$ matrix. In this case study, we describe the optimization process of this dense linear algebra kernel using the BGLsim timing model as a performance-tuning tool. The following example shows the optimization process of a level 2 basic linear algebra software (BLAS) kernel, which performs operations in the form $y = y + A^T x$, where y and x are vectors and A^T is a matrix; A^T is the transposition of A. The routines defined by BLAS are commonly called by a wide range of scientific software and have become a *de facto* standard for elementary linear algebra operations [19]. Therefore, a high-performance implementation of the BLAS kernels has been developed as part of the math library that will be delivered with the BG/L supercomputer.

As the main goal is to achieve the highest performance in a single-processor computation, some of the BLAS kernels are written in assembly language and then hand-tuned such that an efficient pipelined execution is created for the kernel. As we show here, the results produced by the BGLsim timing model helped us identify inefficient sequences of code that were leading to stalls in the pipeline and consequently degrading the execution performance. The simulator also gives us the total number of cycles needed for the execution of a piece of code under a specific workload. In the optimization process, we change the scheduling of the instructions on the basis of the information provided by the simulator timing model. After the changes, the new version is tested again, and a new output is generated. Therefore, each consecutive version is improved on the basis of the time information provided by the BGLsim.

The first version of the code was implemented and executed in the simulator. The output produced by the simulator gives pseudo cycle-accurate information on the instructions issued. The output of the execution of the code related to the inner loop of the DGEMV kernel is shown in **Figure 4**. Every iteration of this loop computes the product $y[j] = y[j] + A[i][j] \cdot x[i]$ for every element of an 8×2 block of elements of the matrix A. As seen through the execution output, every iteration takes 12 cycles to complete. Therefore, the ratio of elements computed to CPU cycles is 4/3. Moreover, the output of the simulation tells us that every fused multiply–add (FMA) instruction is paired with a load instruction; consequently, both instructions are issued in the same cycle. However, many cycles are spent with just a load instruction being issued, which means that the computation pipeline is idle in that cycle.

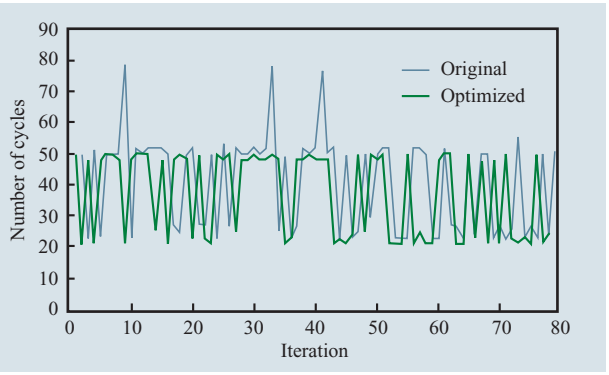


Figure 3

Loop 2 execution times before and after optimization.

```
Inst Cycle PC       Opcode   Decoded Instruction
1936 7586  00100EF0 7dce8bdc lfpdux f14, r14, r17
1937 7586  00100EF4 01705de6 fxcsmadd f11, f16, f11, f23
1938 7587  00100EF8 7e5093dc lfpdux f18, r16, r18
1939 7588  00100EFC 7e7093dc lfpdux f19, r16, r18
1940 7588  00100F00 01916624 fxcpmadd f12, f17, f12, f24
1941 7589  00100F04 7dee8bdc fpdux f15, r14, r17
1942 7589  00100F08 01b16e66 fxcsmadd f13, f17, f13, f25
1943 7590  00100F0C 7e9093dc lfpdux f20, r16, r18
1944 7591  00100F10 7eb093dc lfpdux f21, r16, r18
1945 7591  00100F14 014e54a4 fxcpmadd f10, f14, f10, f18
1946 7592  00100F18 7e0e8bdc lfpdux f16, r14, r17
1947 7592  00100F1C 016e5ce6 fxcsmadd f11, f14, f11, f19
1948 7594  00100F20 7ed093dc lfpdux f22, r16, r18
1949 7595  00100F24 7ef093dc lfpdux f23, r16, r18
1950 7595  00100F28 018f6524 fxcpmadd f12, f15, f12, f20
1951 7596  00100F2C 7e2e8bdc lfpdux f17, r14, r17
1952 7596  00100F30 01af6d66 fxcsmadd f13, f15, f13, f21
1953 7597  00100F34 7f1093dc lfpdux f24, r16, r18
1954 7598  00100F38 7f3093dc lfpdux f25, r16, r18
1955 7598  00100F3C 015055a4 fxcpmadd f10, f16, f10, f22
```

Figure 4

Cycles of the inner loop in the initial implementation of the DGEMV kernel.

Considering the results of the first version of the code, after collecting execution traces from the simulator with the cycles for each instruction, we did some improvement on the instruction scheduling to reduce the bubbles on the pipeline. This yielded Version 2, and we repeated the same process testing different instruction schedules until the final version was produced.

```
Inst Cycle PC       Opcode   Decoded Instruction
7430 2438  00100F24 7e6e8bdc lfpdux f19, r14, r17
7431 2439  00100F28 7cf09bdc lfpdux f7, r16, r19
7432 2439  00100F2C 00340b66 fxcsmadd f1, f20, f1, f13
7433 2440  00100F30 7d108bdc lfpdux f8, r16, r17
7434 2440  00100F34 005413a6 fxcsmadd f2, f20, f2, f14
7435 2441  00100F38 7d308bdc lfpdux f9, r16, r17
7436 2441  00100F3C 00741be6 fxcsmadd f3, f20, f3, f15
7437 2442  00100F40 7d508bdc lfpdux f10, r16, r17
7438 2442  00100F44 00942426 fxcsmadd f4, f20, f4, f16
7439 2443  00100F48 7d708bdc lfpdux f11, r16, r17
7440 2443  00100F4C 00b42c66 fxcsmadd f5, f20, f5, f17
7441 2444  00100F50 7d908bdc lfpdux f12, r16, r17
7442 2444  00100F54 00d434a6 fxcsmadd f6, f20, f6, f18
7443 2445  00100F58 7ed08bdc lfpdux f22, r16, r17
7444 2445  00100F5C 02b4ade6 fxcsmadd f21, f20, f21, f23
7445 2446  00100F60 02809840 fpmr f20, f0, f19
7446 2446  00100F64 7db09bdc lfpdux f13, r16, r19
7447 2447  00100F68 003309e4 fxcpmadd f1, f19, f1, f7
7448 2447  00100F6C 7dd08bdc lfpdux f14, r16, r17
7449 2448  00100F70 00531224 fxcpmadd f2, f19, f2, f8
7450 2448  00100F74 7df08bdc lfpdux f15, r16, r17
7451 2449  00100F78 00731a64 fxcpmadd f3, f19, f3, f9
7452 2449  00100F7C 7e108bdc lfpdux f16, r16, r17
7453 2450  00100F80 009322a4 fxcpmadd f4, f19, f4, f10
7454 2450  00100F84 7e308bdc lfpdux f17, r16, r17
7455 2451  00100F88 00b32ae4 fxcpmadd f5, f19, f5, f11
7456 2451  00100F8C 7e508bdc lfpdux f18, r16, r17
7457 2452  00100F90 00d33324 fxcpmadd f6, f19, f6, f12
7458 2452  00100F94 7ef08bdc lfpdux f23, r16, r17
7459 2453  00100F98 02b3ada4 fxcpmadd f21, f19, f21, f22
```

Figure 5

Cycles of the inner loop in the optimized implementation of the DGEMV kernel.

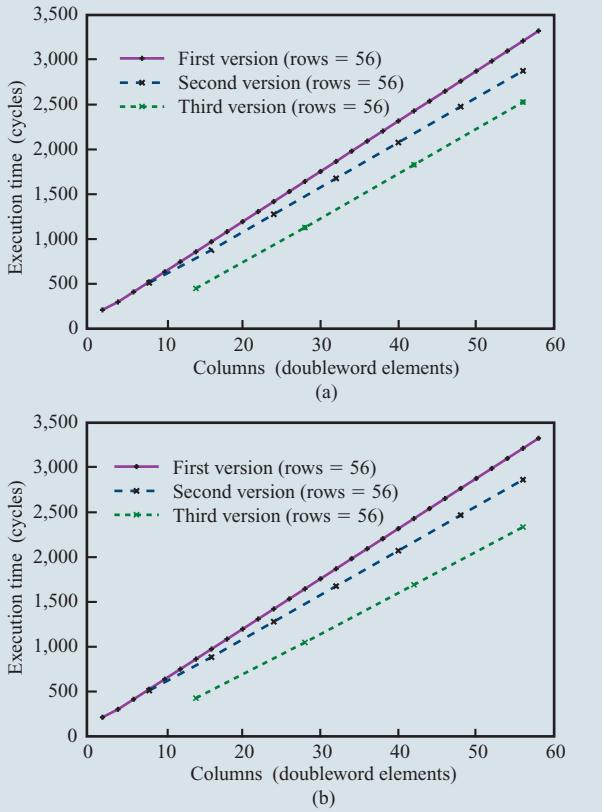


Figure 6

Running time of different versions of the DGEMV kernel on (a) BG/L hardware and (b) BG/L simulator.

The output of the inner loop of the latest version is shown in **Figure 5**. In this loop the product is computed for a 2×14 block of elements of the matrix A in each iteration, and the ratio of elements computed to CPU cycles is 28/15, which gives us a better utilization of the instruction pipeline. Moreover, more FMAs are interleaved with load instructions, which translates to a better use of processor resources, keeping the pipeline busy most of the time.

Figures 6(a) and **6(b)** respectively show the running times for different versions of a DGEMV kernel as run on hardware and as predicted through BGLsim. The DGEMV kernel has been optimized for best performance considering that the data is already in the L1 cache. As the plot shows, the better the instruction scheduling, the faster the kernel execution for a given workload. For both experiments, similar performance trends are observed as the DGEMV is improved. Hence, we observe that by using the BGLsim timing model, we were able to generate a version of the DGEMV kernel optimized for execution by the hardware. The BGLsim timing model does particularly well when simulating straight-line code that accesses primarily the L1 cache, as represented by the code fragments in Figures 4 and 5. The maximum discrepancy between simulation and hardware results that we observe in Figures 6(a) and 6(b) is 7.5%. The accuracy of the pseudo cycle-accurate timing model depends on the access pattern and the number of misses at L1. That is the reason why the different experiments have different accuracy with respect to the hardware execution.

Detecting message-passing overhead

One of the characteristics of the BG/L supercomputer is that it has a very fast network compared with the speed of the processor. During the project, we have developed some microbenchmarks to determine how much information the processor can deal with in and out of the network. Ideally, the processor should be able to manage six incoming and six outgoing links. In a machine running

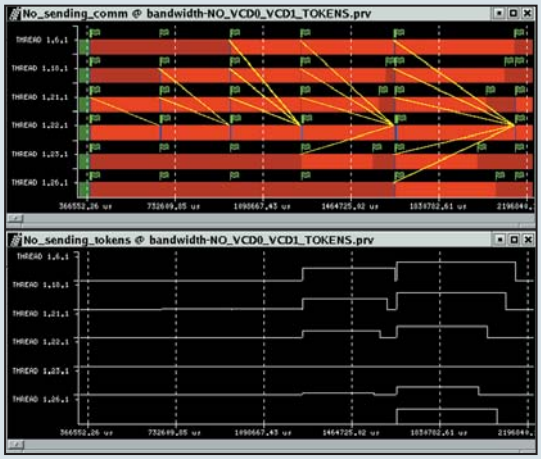
at 500 MHz, each link is able to sustain 110 MB/s, giving each node the capacity to move a total of 1,320 MB/s.

We have instrumented one of these microbenchmarks with Paraver and hardware counters. This section presents the results of this study. The microbenchmark first selects a node in the middle of the current BG/L partition and its closest neighbors. It then starts a set of iterations in which the number of senders and receivers is increased. In each communication phase, 30 messages of 1 MB of data each are sent. For the analysis, we collected the hardware performance counters which indicated that a link in each direction was available but there were no tokens available for the node to send. When this occurs, the specific link is full, and the destination node is not draining it at the proper speed to sustain the required bandwidth.

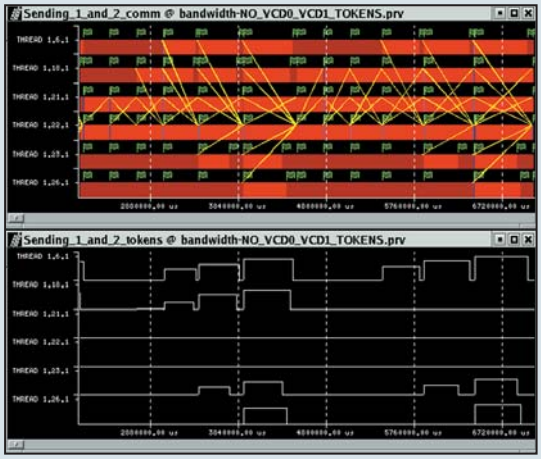
The microbenchmark was executed in a 32-node partition, in which any of the central nodes has up to five neighbors. The top plot of **Figure 7(a)** presents the behavior of the communication phases in which a single node (node 22—the node number is the middle term in the expression following "Thread") is receiving messages from one to five nodes simultaneously. The plot in the bottom shows the behavior of the counters indicating network congestion. These counters are incremented every cycle in which a network link is available but the hardware has no token to send data. Not having a token is usually caused by the fact that there are packets in transit, and the destination node is not able to collect them. As shown, the receiver (node 22) can deal with up to three incoming links without experiencing network congestion. As soon as a fourth sender becomes active, all senders start seeing a lack of tokens; this is because the receiver is not draining the links fast enough. Observe also that, as the counter value becomes higher, the execution time increases for these messages to be received. That is the effect of the sender being blocked while waiting for tokens.

Figure 7(b) shows the same information when, in addition to receiving messages, node 22 also sends messages to first one and then two destination nodes. Observe that when there is a single destination node to which node 22 is sending messages, node 22 is no longer able to deal with three incoming links. A lack of tokens appears first at nodes 6 and 18 and then later at nodes 23 and 26. Also observe that as soon as a node is busy receiving messages, it no longer has sending problems due to the lack of tokens. This is because it has to send more slowly. This happens to node 18 in Figure 7(b) and also to node 21, when the lower plots in Figures 7(a) and 7(b) are compared.

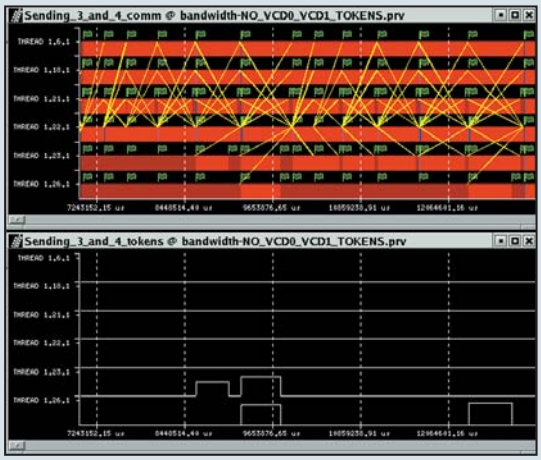
Figure 7(c) presents the behavior of the microbenchmark when the central node, in addition to receiving messages, sends to three and four destination



(a)



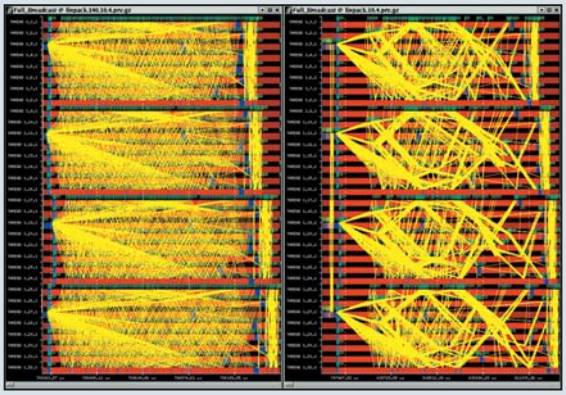
(b)



(c)

Figure 7

Lack of sending tokens when (a) receiving from a different number of nodes; (b) receiving from many and sending to one and two nodes; and (c) receiving from many and sending to three and four nodes.



(a) (b)

Figure 8

Comparison of the behavior of two versions of the manual broadcast: (a) First version (single outgoing message sent in each direction). (b) Alternative version (several messages sent round-robin to the same destination).

Table 3 Gigaflops obtained in Linpack with the two versions of the message layer.

Problem size	*Single message*	*Overlapping messages*
10,240	50.43	49.31
15,360	68.41	66.69
20,480	78.38	76.68
29,696	89.25	87.07

nodes. Observe that in this case, the detection of a lack of send tokens is reduced to nodes 23 and 26, which are the ones not receiving messages. Any node receiving messages is unable to deliver messages at enough speed to detect the lack of tokens.

Analysis of the behavior of the message layer

We used the Paraver tool to evaluate different implementation alternatives inside the MPI message layer. In this experiment, we compared the performance obtained using two possible implementations (developed as prototypes) of the low-level message layer on which the MPI implementation relies. The Linpack benchmark indirectly uses this portion of the message layer through the MPI library to implement a hand-coded version of the broadcast collective. This hand-coded version of broadcast performs better than the built-in MPI broadcast using any of the implementations. Currently, this broadcast is being implemented inside the MPI library.

The difference between the two implementations was in the way messages are sent:

- *One message at a time (first in, first out, FIFO, mode):* Considering that each node has six connections to its neighbors, the first implementation of the message layer allowed sending up to six messages at a time (one in each different direction). That way, the send queues in the connection manager (see Figure 2) contain a single outgoing message for each direction.
- *Overlapping messages:* We wanted to test whether allowing several outgoing messages for the same destination at the same time could improve communication performance, so we developed a version in which any outgoing message was immediately posted to be delivered to the network. In this case, packets are picked up in a round-robin fashion from all available messages in each direction.

After implementing both of these ways of dealing with messages, we evaluated the performance of the Linpack benchmark in 32 nodes. **Table 3** shows the results of the comparison. We observed that the performance obtained in this application was slightly worse with overlapping messages. Using Paraver, we were able to look inside the application and detect which part of it performed worse and why.

Figure 8 shows the behavior of one of the broadcasts that was hand-coded inside the Linpack benchmark using point-to-point communications. As can be observed, the transmission of the messages is different in the two versions of the message layer. The plot on the left in Figure 8 corresponds to the first version of the message layer, which sends a single outgoing message in each direction. The plot on the right corresponds to the alternative implementation, in which several messages are sent in a round-robin fashion to the same destination.

We can observe that in the plot on the left, the first message sent from nodes 3, 11, 19, and 27 reaches the destination earlier than that in the plot on the right. That is precisely because each single-link bandwidth is devoted to a single message. Instead, in the plot on the right, all outgoing messages from these nodes are sent in parallel, so the first and the last ones are complete at the destination nearly at the same time. Because of the way this broadcast is implemented, each destination node is going to retransmit the information to other nodes. The parallel implementation causes the retransmission of the first messages to be delayed because of the late arrival, and this causes the performance degradation. The actual degradation of the broadcast code was 30%.

In conclusion, time-sharing the links between MPI messages to the same destination results in all messages taking about the same time to arrive at their destination. Keeping a FIFO order in sending messages through the link also has the potential to keep the link fully used, but

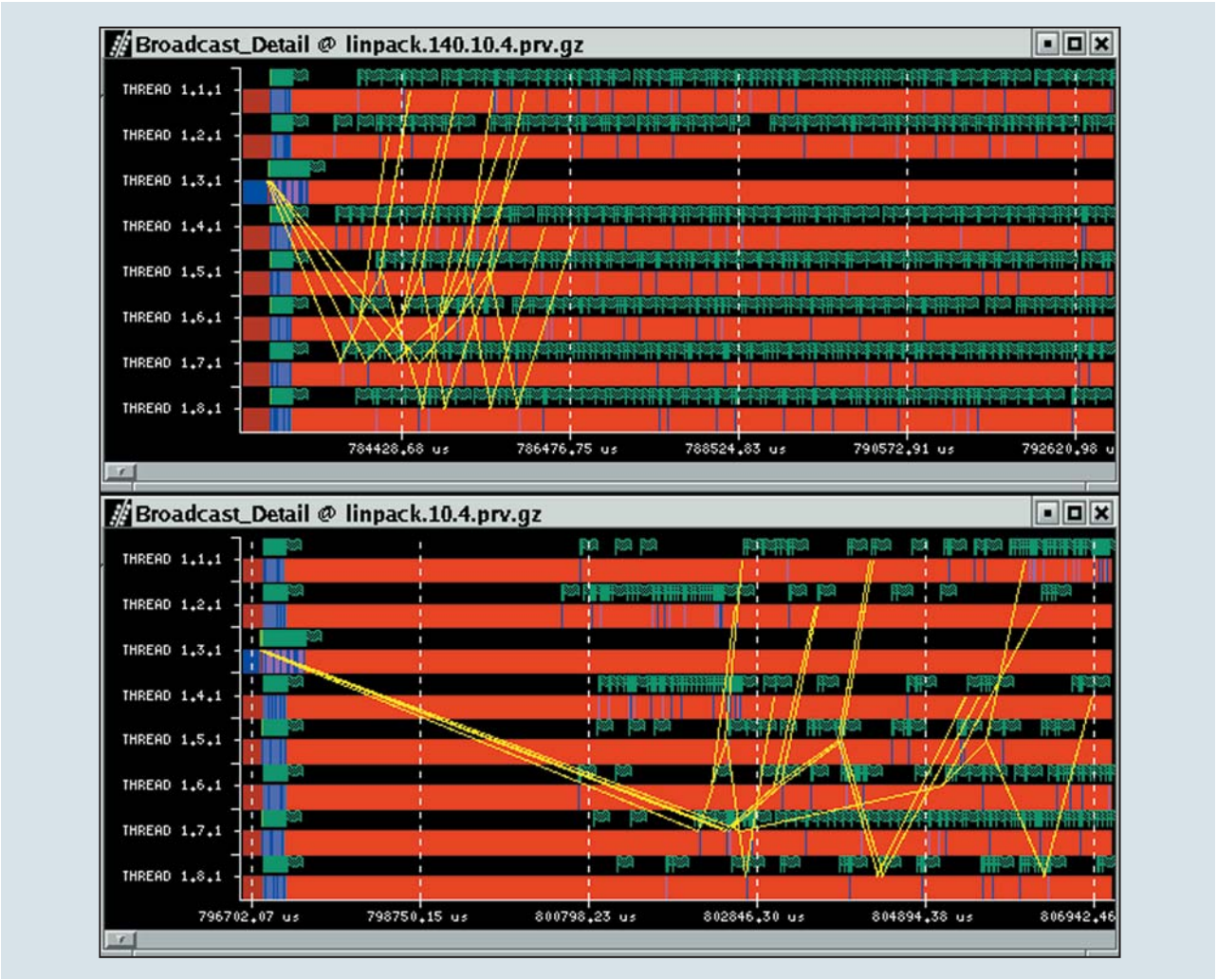


Figure 9

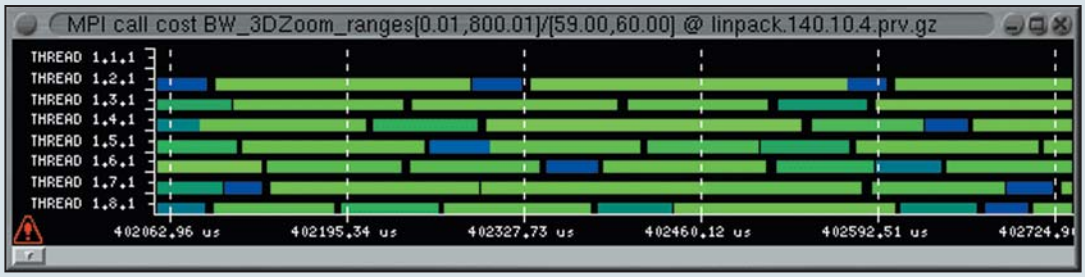
Comparison of specific messages sent from node 3 in broadcast. (The node number is the middle term in the expression following "THREAD.")

results in some messages arriving earlier than others. In a situation in which all of the messages are of similar length and most of them have to be retransmitted, the version that keeps the FIFO order has the potential for better performance. In this case, retransmissions will start earlier, increasing the number of simultaneously active links. This is visible in **Figure 9**, which presents, at the same timescale, a set of messages sent by node 3 and its retransmissions, clearly showing the benefits of having the link dedicated to a single message at a time.

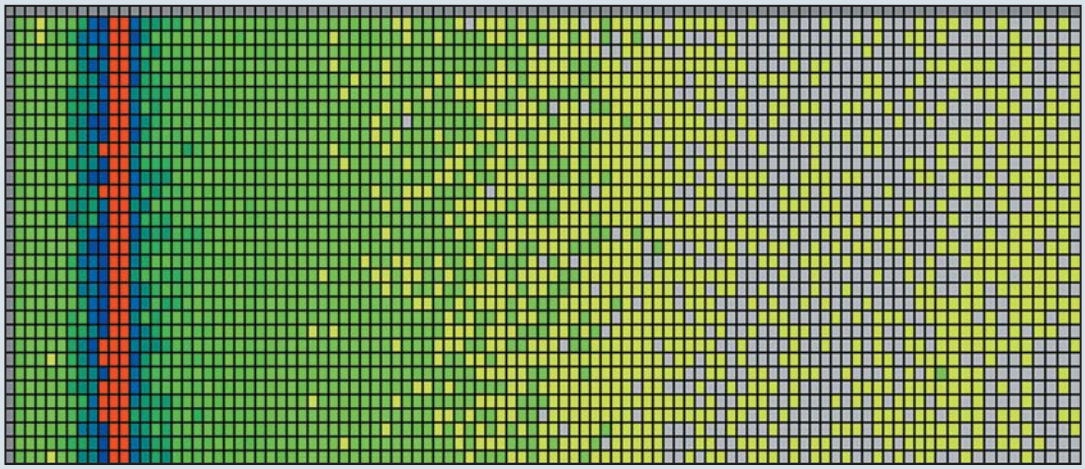
The Paraver traces helped identify this issue and provided a good understanding of its detailed impact in this situation. Conceptually, in other situations with messages of different sizes, the time-sharing version might be advantageous, depending on what the application does with different messages.

From the analysis of the traces, we also inferred some suggestions to the application developer about the way the broadcast is implemented. It might be useful, for example, first to send and receive messages that have to be retransmitted, and only at the end send messages that constitute the leaves of the broadcast.

Another suggestion comes from the observation that the broadcast is actually decomposed in four subtrees, where the root of the broadcast pipelines the message to the four roots of each subtree. Unfortunately, two of those trees end up being assigned to the same physical processor. In this case, the cause is that the neighbors in the $z+$ and $z-$ directions are the same nodes. This is due to



(a)



(b)

Figure 10

(a) Cost of the `MPI_Waitany` calls in a section of the broadcast. (b) Histogram of the `MPI_Waitany` calls in a broadcast: yellow if less than five instances, a linear gradient from five instances (light green) to 80 instances (dark blue), and red if more than 80 instances.

the $4 \times 4 \times 2$ topology in a partition containing 32 nodes. A more balanced topology would probably result in better performance.

A final observation is that the root of the broadcast pipelines the messages to each subtree root, but it finishes sending long before the end of the whole broadcast. This suggests that imbalanced tree approaches, where the root keeps transmitting during the whole operation, would potentially improve the utilization of the links.

We were also interested in a more detailed analysis of the FIFO version. **Figure 10(a)** shows a view in which the effective cost of each `MPI_Waitany` call is reported in MB/s. By effective cost, we mean the ratio between the number of bytes received by the call and the time taken for the wait to complete. For any MPI point-to-point call, this local bandwidth is a fair metric of how efficient the call has been in handling the data it had to deal with. The view focuses on a few threads and a short period of time, and a light green color represents 40 MB/s and dark blue 400 MB/s. Surprisingly, even if the size is the same for all of them, different instances of the `MPI_Waitany` take rather different amounts of time.

Figure 10(b) shows a histogram for the whole trace of such effective cost. For each process (row), the column represents a range of 10 MB/s (up to a total of 1,000 MB/s). The color of each entry corresponds to the total number of times an `MPI_Waitany` call achieved the particular local effective bandwidth. As can be seen, there is a major mode around 100 MB/s that corresponds to the link bandwidth. There are a significant number of instances that achieve less than the link bandwidth. Finally, it is interesting to see some instances achieving close to a GB/s. Nevertheless, averaging over the whole duration of a broadcast, each processor performs `MPI_Waitany` calls at the rate of 100 MB/s. This value is still far from the physical limits of the interconnect.

The interpretation for this behavior is related to the fact that the data is sent to and drained from the network interface by the main node processor through polling. The processor becomes the bottleneck because it is not

capable of feeding and draining the six I/O links at their full speed. Additionally, there are issues related to how to proceed if the processor is simultaneously sending one message and receiving another. When, for example, the reception is finalized, should control be returned immediately to the user or should the transmission be finalized? In both cases, somebody (the local or the remote node) is going to be delayed. Independent direct memory access (DMA) engines would certainly help here. The extreme variation in bandwidth achieved by some calls can be explained if we consider that several messages may be arriving at a node simultaneously. If the node cannot cope with it, the whole process is slowed down, and the first reception to finalize perceives a low bandwidth. By that time, it is quite possible that the next incoming message has almost been received, so when the next `MPI_Waitany` call is invoked, control returns very soon, resulting in a huge local bandwidth perceived by such a call.

Analysis of the behavior of Linpack broadcasts

Figure 11 shows a set of views of the communication phase in a Linpack version that performs the broadcasts directly through the MPI broadcast call. The run is for a problem size of 40K on a 32-processor system. The upper view displays the MPI call (yellow: broadcast; red: barrier; blue: send; white: receive). In the second view, we see those processors that are the root of a given broadcast (the different colors represent the different communicators). The third and fourth views are derived from the hardware counter information emitted into the trace at the entry and exit of each MPI call. The third view is an estimate of the number of active links. Here, the important issue is that during most of the broadcast time, most nodes show only a single active link. Only the root processors achieve two active links. During the second broadcast region, which performs a column broadcast in Linpack, some nodes achieve three simultaneous active links (red). The fourth view is the equivalent bandwidth going out of a node along all links during the whole call. In accordance with the third view, light green is the predominant color across the first broadcast and large portions of the second. This means that the bandwidth achieved is far from the peak.

It is possible to compute a histogram of the bandwidth used during the major broadcast with message sizes above 7 MB. From the analysis of that histogram, we can see that the root processor achieves an effective total bandwidth of the order of 117 MB/s, while most other processors show either 78 MB/s or 39 MB/s.

Another capability of the tools environment is the possibility of using the powerful metric derivation and analysis capabilities of Paraver to generate the ASCII data for a locally developed tool, BGLnodes (discussed above). This tool displays a single scalar value for each processor in the physical topology. **Figures 12(a)**, **12(b)**, and **12(c)** respectively show the bandwidth obtained during the major broadcast, in the x, y, and z dimensions.

Performance counter limitations

From the experience with the current chip implementation of performance counters on BG/L, some lessons can be learned. BG/L is a machine targeted to address the grand challenges in high-performance computing. These applications typically amount to a large number of floating-point operations. In this context, the capabilities of the double-hummer FPU performance counters are limited. There is one counter in the FPU capable of registering arithmetic events. This counter counts operations that belong to any of the following groups: additions and subtractions, multiplications and divisions, trinary operations, and Oedipus operations. The first three groups relate to single-pipe operations. The trinary operations are operations of the form $a \pm b \cdot c$. This corresponds to two classical floating-point operations. The Oedipus operations are trinary operations that use both functional pipes in the FPU, using up to six operands and producing two results per instruction. Parallel single- and dual-operator instructions (such as, for example, `fpadd`) do not map into any of the countable groups of events. Thus, even with repeated runs of the same code, it is not possible to count the complete number of floating-point instructions performed. For the same reason, it is not possible to compute a corresponding number of floating-point operations of an algorithm by using only the performance counters.

End users doing advanced tuning of large applications would most likely gain from a CPU core implementation that incorporated a performance-counter infrastructure. The most noticeable events that are not possible to detect are issued loads and stores, L1 cache events, branch unit events (such as branches correctly predicted compared with mispredicted branches), and instruction issues. The impressive performance available in modern CPU design is highly dependent on the ability of the code developer and compiler to generate instruction sequences in which branch prediction is mostly correct and the instruction cache hit ratio is maximized. Without hardware performance counters capable of generating a view inside the units of the core that control these aspects of the CPU, the code developer has no accurate way to determine success in fully utilizing the inherent computational power of the platform.

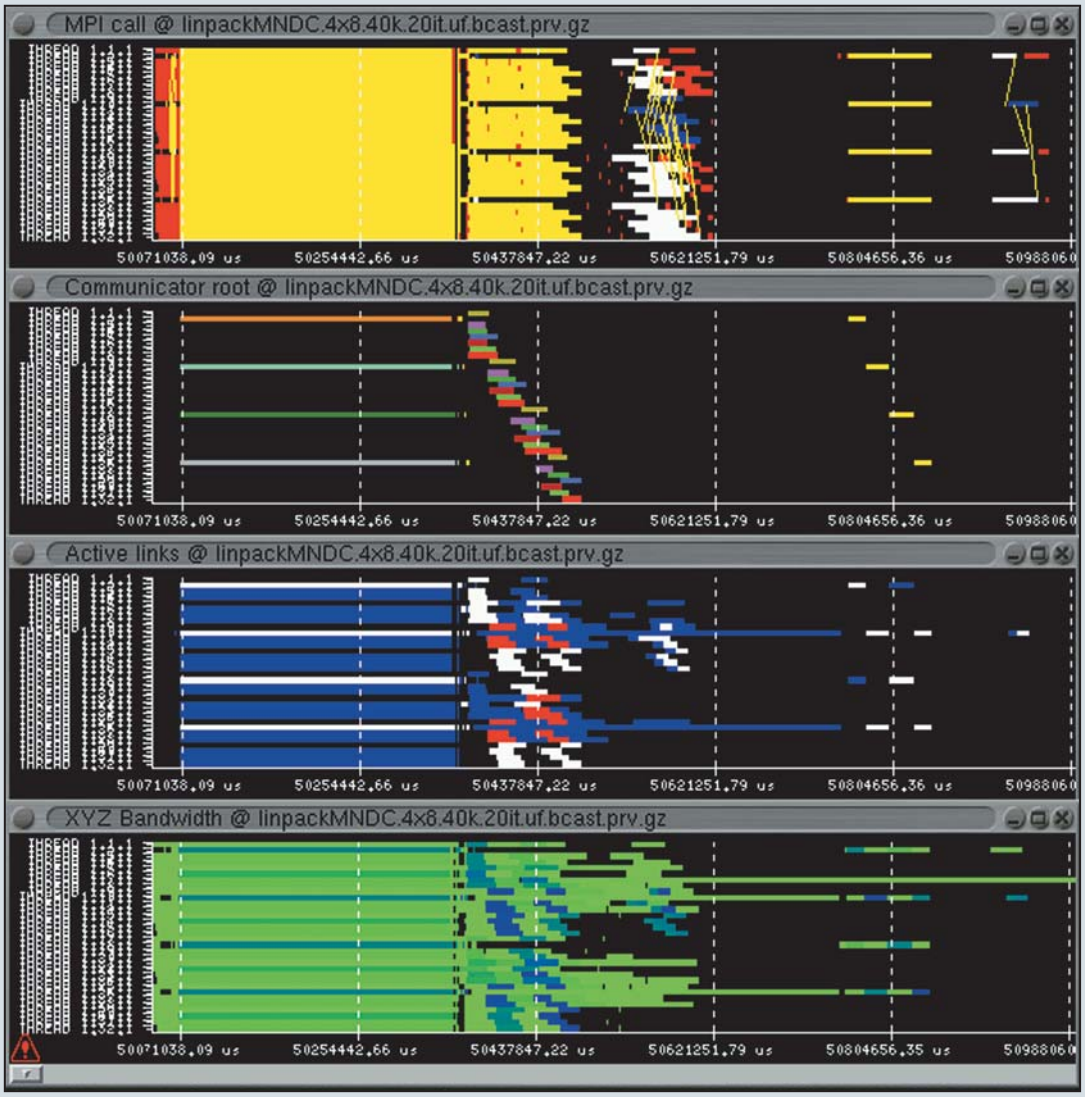


Figure 11

Representation of the execution of the Linpack broadcasts. The upper view displays the MPI call (yellow: broadcast; red: barrier; blue: send; white: receive). In the second view, the different colors represent the different communicators. The third view is an estimate of the number of active links (blue: 1; white: 2; red: 3). The fourth view is the equivalent bandwidth going out of a node along all links during the whole call (gradient from green to blue; dark blue: 200 MB/s).

Related work

Vampir [20] is a commercial product for performance analysis that allows tracing and analysis of MPI applications. Several execution environments such as ParaWise[1] [21] provide an interface for generating Vampir traces. Two research projects on performance analysis are Paradyn** [22], developed at the University of Wisconsin, and Aksum, part of the Askalon [23] project conducted at the University of Vienna. Both aim at the automatic detection of performance bottlenecks. Tuning and Analysis Utilities (TAU) [24] was developed at the University of Oregon. It is a set of tools for analyzing the performance of C, C++, Fortran, and Java** programs.

The advantage offered by Paraver is a high level of flexibility in computing performance indices and statistics. This usually allows the exploration of metrics of interest and the influence of the parallelization choices on them.

[1] Formerly known as CAPTools.

Conclusions

In this paper, we have presented a set of tools devoted to performance analysis of the Blue Gene/L supercomputer. The tools in this set range from a hardware simulator and low-level libraries to visualization and analysis tools. They are currently being ported and adapted to the BG/L environment, and should not be considered as finished work. We have made initial explorations of the possibilities this new architecture provides for performance analysis.

BGLsim is a pseudo cycle-accurate simulator that runs full-system simulations and provides monitoring capabilities beyond the level possible with real hardware.

LibHPM, BGLperfctr, and PAPI are user-level libraries capable of managing the hardware performance counters available in BG/L and extracting information during application runtime. The MPItrace library collects traces during execution for later visualization and analysis.

Paraver and BGLnodes are visualization tools that present the traces obtained (including performance counters) and allow the user to analyze in detail what is happening inside the application.

Finally, we have demonstrated the power of an environment for collecting information about the execution and using it to explain the performance obtained. We have also presented a set of experiences optimizing code using information obtained by simulation. We have used the specific hardware performance counters in the torus network to analyze the behavior of the communications and determine the limitations of the processor in each node when dealing with up to six incoming and outgoing links. We have also analyzed the implementation of the MPI message layer and a hand-coded broadcast in the Linpack application.

Acknowledgment

The Blue Gene/L project has been supported and partially funded by the Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Lawrence Livermore National Laboratory Subcontract No. B517552.

* Trademark or registered trademark of International Business Machines Corporation.

** Trademark or registered trademark of Linus Torvalds, Barton P. Miller, or Sun Microsystems, Inc., in the United States, other countries, or both.

References

1. G. S. Almasi, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, A. A. Bright, J. Brunheroto, C. Cascaval, J. Castanos, L. Ceze, P. Coteus, S. Chatterjee, D. Chen, G. Chiu, T. M. Cipolla, P. Crumley, A. Deutsch, M. B. Dombrowa, W. Donath, M. Eleftheriou, B. Fitch, J. Gagliano, A. Gara, R.

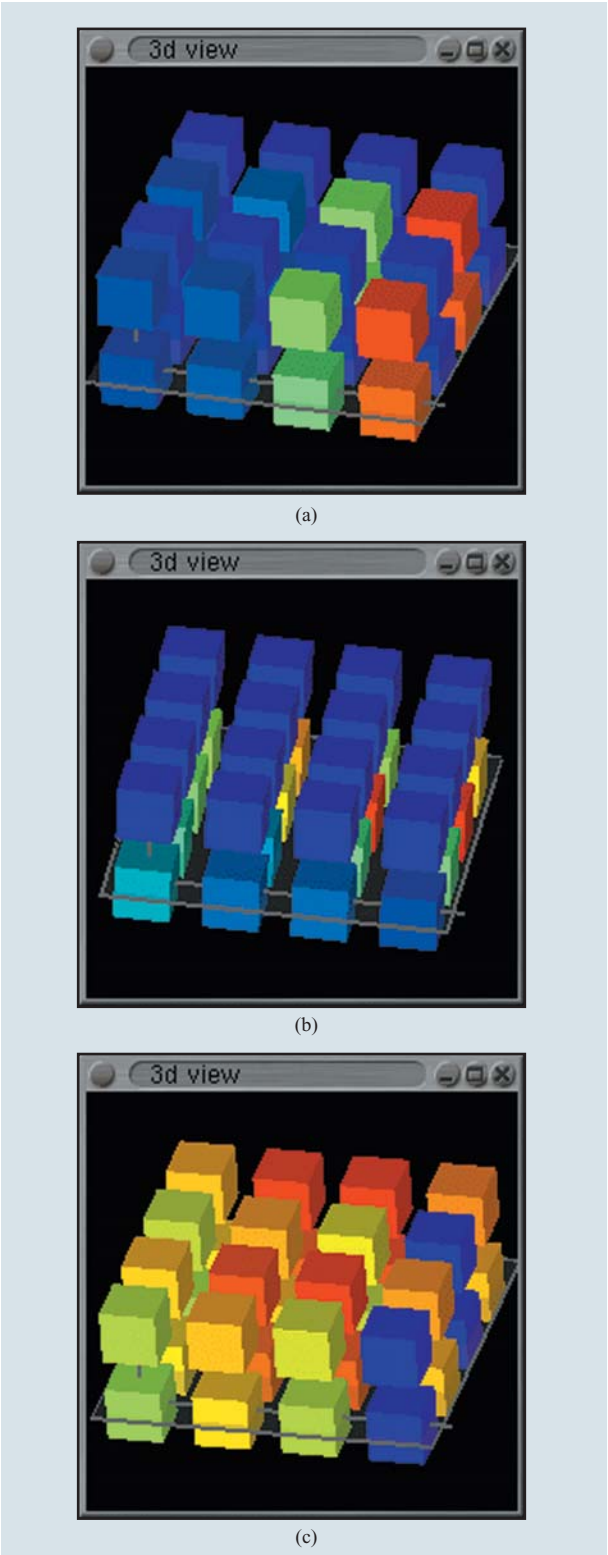


Figure 12

Linpack broadcast bandwidth in (a) *x* direction, (b) *y* direction, and (c) *z* direction.

Germain, M. E. Giampapa, M. Gupta, F. Gustavson, S. Hall, R. A. Haring, D. Heidel, P. Heidelberger, L. M. Herger, D. Hoenicke, R. D. Jackson, T. Jamal-Eddine, G. V. Kopcsay, A. P. Lanzetta, D. Lieber, M. Lu, M. Mendell, L. Mok, J. Moreira, B. J. Nathanson, M. Newton, M. Ohmacht, R. Rand, R. Regan, R. Sahoo, A. Sanomiya, E. Schenfeld, S. Singh, P. Song, B. D. Steinmacher-Burow, K. Strauss, R. Swetz, T. Takken, P. Vranas, T. J. C. Ward, J. Brown, T. Liebsch, A. Schram, and G. Ulsh, "Cellular Supercomputing with System-on-a-Chip," *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, Digest of Technical Papers, Vol. 2, 2002, pp. 152–153.
2. N. R. Adiga et al., "An Overview of the Blue Gene/L Supercomputer," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2002, pp. 1–22; see *www.sc-conference.org/sc2002/*.
3. G. Almasi, R. Bellofatto, J. Brunheroto, C. Cascaval, J. G. Castanos, L. Ceze, P. Crumley, C. Erway, J. Gagliano, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and K. Strauss, "An Overview of the BlueGene/L System Software Organization," *Proceedings of the 9th International EuroPar Conference*, 2003, pp. 543–555.
4. G. Almasi, C. Archer, J. G. Castanos, C. C. Erway, P. Heidelberger, X. Martorell, J. E. Moreira, K. Pinnow, J. D. Ratterman, N. Smeds, B. Steinmacher-Burow, W. Gropp, and B. Toonen, "Implementing MPI on the BlueGene/L Supercomputer," *Proceedings of the 10th International EuroPar Conference*, 2004.
5. MPICH and MPICH2 homepage; see *http://www-unix.mcs.anl.gov/mpi/mpich*.
6. W. Gropp, E. Lusk, D. Ashton, R. Ross, R. Thakur, and B. Toonen, MPICH Abstract Device Interface Version 3.4 Reference Manual, May 20, 2003 (draft); see *http://www-unix.mcs.anl.gov/mpi/mpich/adi3/adi3man.pdf*.
7. G. Chiola and G. Ciaccio, "Efficient Parallel Processing on Low-Cost Clusters with GAMMA Active Ports," *Parallel Computing* **26**, No. 2/3, 333–354 (2000).
8. S. Pakin, M. Lauria, and A. Chien, "High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet," *Proceedings of the International Conference on Supercomputing*, 1995, pp. 1–22.
9. T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995, pp. 40–53.
10. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," *Proceedings of the 19th International Symposium on Computer Architecture*, 1992, pp. 256–266.
11. K. Davis, A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, S. Pakin, and F. Petrini, "A Performance and Scalability Analysis of the BlueGene/L Architecture," *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2004; see *http://www.sc-conference.org/sc2004/schedule/pdfs/pap302.pdf*.
12. L. A. DeRose, "The Hardware Performance Monitor Toolkit," *Proceedings of the 7th International EuroPar Conference*, 2001, pp. 122–131.
13. P. Mindlin, J. R. Brunheroto, L. DeRose, and J. E. Moreira, "Obtaining Hardware Performance Metrics for the BlueGene/L Supercomputer," *Proceedings of the 9th International EuroPar Conference*, 2003, pp. 109–118; see *http://www.llnl.gov/asci/platforms/bluegenel/pdf/metrics.pdf*.
14. L. Ceze, K. Strauss, G. Almasi, P. J. Bohrer, J. R. Brunheroto, C. Cascaval, J. G. Castanos, D. Lieber, X. Martorell, J. E. Moreira, A. Sanomiya, and E. Schenfeld, "Full Circle: Simulating Linux Clusters on Linux Clusters," *Proceedings of the 4th LCI International Conference on Linux Clusters: The HPC Revolution*, 2003; see *http://www.llnl.gov/asci/platforms/bluegenel/pdf/linux.pdf*.
15. L. R. Bachega, J. R. Brunheroto, L. DeRose, P. Mindlin, and J. E. Moreira, "The BlueGene/L Pseudo Cycle-Accurate Simulator," *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2004, pp. 36–44.
16. J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and Lessons Learned with a Portable Interface to Hardware Performance Counters," *Proceedings of PADTAD Workshop, IPDPS Meeting*; see *http://icl.cs.utk.edu/papi/pubs/*.
17. European Center for Parallelism of Barcelona (CEPBA). More information on Paraver may be found at *http://www.ciri.upc.es/paraver/index.html*.
18. See *http://www.nas.nasa.gov/Research/Reports/Techreports/1994/HTML/node17.html*.
19. J. J. Dongarra, J. D. Croz, S. Hammarling, and I. S. Duff, "An Extended Set of FORTRAN Basic Linear Algebra Subprograms," *ACM Trans. Math. Software* **14**, No. 1, 1–17 (March 1988).
20. Intel GmbH, Software and Solutions Group, Hermuelheimer Str. 8a, 50321 Bruehl, Germany; see *http://www.pallas.com/e/products/vampir/*.
21. C. S. Ierotheou, S. P. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelization Tools (CAPTools)—Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes," *Parallel Computing* **22**, No. 2, 163–195 (February 1996); see also *http://www.parallelsp.com/*.
22. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapdam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools," *IEEE Computer* **28**, No. 11, 37–46 (1995); see also *http://www.cs.wisc.edu/~paradyn/*.
23. Askalon; see *http://www.par.univie.ac.at/project/askalon/*.
24. TAU: Tuning and Analysis Utilities; see *http://www.cs.uoregon.edu/research/paracomp/tau/tautools/*.

Received July 16, 2004; accepted for publication September 10, 2004; Internet publication April 12, 2005

Xavier Martorell *Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (xavim@us.ibm.com)*. Dr. Martorell received M.E. and Ph.D. degrees in computer science from the Technical University of Catalonia (UPC), in 1991 and 1999, respectively. He has been lecturing on operating systems and parallel runtime systems since 1992. He has been an Associate Professor in the Computer Architecture Department at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, and compilers for high-performance multiprocessor systems. Dr. Martorell has participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs, and also in collaboration with the IBM Thomas J. Watson Research Center.

Nils Smeds *Center for Parallel Computers, Royal Institute of Technology, S-100 44 Stockholm, Sweden (smeds@pdc.kth.se)*. Dr. Smeds has been working at the Center for Parallel Computers, the major national facility for academic high-performance computing in Sweden, since 1996. He received an M.A. degree in physics and a Ph.D. degree in solid mechanics from the Royal Institute of Technology. His special areas of interest are performance measurement and tuning of scientific codes. Dr. Smeds has an active interest in parallel programming paradigms such as OpenMP and MPI. He has been a contributor to the Performance Application Programming Interface (PAPI) project at the University of Tennessee at Knoxville for many years. Dr. Smeds has participated in several European Union projects. Through this work he has been on extended stays at the University of New South Wales, Australia, and the IBM Thomas J. Watson Research Center.

Robert Walkup *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (walkup@us.ibm.com)*. Dr. Walkup is a Research Staff Member at the IBM Thomas J. Watson Research Center. He received his Ph.D. degree in physics from the Massachusetts Institute of Technology. Dr. Walkup currently works on high-performance computing algorithms.

José R. Brunheroto *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (brunhe@us.ibm.com)*. Mr. Brunheroto is a Senior Software Engineer at the IBM Thomas J. Watson Research Center. He received a B.S. degree in electrical engineering from Escola Politécnica, University of São Paulo, Brazil. Mr. Brunheroto is currently working on his M.A. degree in computer science at Columbia University. His research interests are on computer architecture, simulators (single-node and distributed), and performance tools.

George Almási *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gheorghe@us.ibm.com)*. Dr. Almási is a Research Staff Member at the IBM Thomas J. Watson Research Center. He received an M.S. degree in electrical engineering from the Technical University of Cluj-Napoca, Romania, in 1991 and an M.S. degree in computer science from West Virginia University in 1993. In 2001 he received a Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign; his thesis dealt with ways of optimizing and compiling MATLAB code. For the last three years, Dr. Almási has been working on various aspects of the Blue Gene system software environment, including the MPI communication libraries.

John A. Gunnels *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (gunnels@us.ibm.com)*. Dr. Gunnels received his Ph.D. degree in computer science from the University of Texas at Austin. He joined the Mathematical Sciences Department of the IBM Thomas J. Watson Research Center in 2001. His research interests include high-performance mathematical routines, parallel algorithms, library construction, compiler construction, and graphics processors. Dr. Gunnels has coauthored several journal papers and conference papers on these topics.

Luiz DeRose *Cray Inc., 1340 Mendota Heights Road, Mendota Heights, Minnesota 55120 (ldr@cray.com)*. Dr. DeRose received a Ph.D. degree in computer science from the University of Illinois at Urbana–Champaign. He is a Senior Principal Engineer at Cray, where he leads the performance tools activities for all Cray platforms. He was the Tools Group leader at the Advanced Computing Technology Center at IBM Research before joining Cray. Prior to that, he worked at the National Center for Supercomputing Applications (NCSA), the Pablo Group at the University of Illinois, and the Center for Supercomputing Research and Development (CSRD). Dr. DeRose has more than 15 years of experience in high-performance computing, working in the areas of performance tools and software support for high-performance computation. He participated in the design and development of performance tools, such as FALCON, SvPablo, SIGMA, the HPM Toolkit, and the Cray Apprentice2.

Jesus Labarta *European Center for Parallelism of Barcelona, Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (jesus@cepba.upc.edu)*. Professor Labarta has been with the Computer Architecture Department at the Technical University of Catalonia (UPC) since 1990. He has lectured on computer architecture, operating systems, computer networks, and performance evaluation since 1981. His research interests have centered on parallel computing, including areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, GRID environments and performance analysis, and prediction tools. He has led the technical work of UPC in 15 industrial research and development projects. Since 1995 Professor Labarta has been the director of the European Center for Parallelism (CEPBA); he has been responsible for three technology transfer cluster projects.

Francesc Escalé *European Center for Parallelism of Barcelona, Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (fescale@cepba.upc.edu)*. Mr. Escalé received an M.A. degree in computer science from the Technical University of Catalonia in 2001. He is currently working at the European Center for Parallelism developing performance analysis tools. His research interests cover the areas of parallel applications and performance analysis tools. Mr. Escalé participated in the Distributed Applications and Middleware for Industrial use of European Networks (DAMIEN) project.

Judit Giménez *European Center for Parallelism of Barcelona, Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (judit@cepba.upc.edu)*. Ms. Giménez received an M.A. degree in computer science from the Technical University of Catalonia (UPC) in 1989. She began her work at UPC on the development of an operating system for distributed memory environments. She left UPC to work for an

English company in Spain, providing technical support for parallel computers. She returned to UPC in 1994 and since then has worked in the area of technology transfer, participating in the management of three European initiatives that grouped 28 projects involving the transfer of parallel technology to small and medium enterprises (SMEs). Since 2000, Ms. Giménez has been responsible for the development and distribution of the European Center for Parallelism of Barcelona (CEPBA) performance analysis tools, Paraver and Dimemas.

Harald Servat *European Center for Parallelism of Barcelona, Technical University of Catalonia, Campus Nord, Modul D6, Jordi Girona 1-3, 08034 Barcelona, Spain (harald@cepba.upc.edu)*.
Mr. Servat received an M.A. degree in computer science from the Technical University of Catalonia in 2003. He is working at the European Center for Parallelism in Barcelona, developing performance analysis tools. His research interests cover the areas of parallel applications and performance analysis tools. He is working in collaboration with the IBM Thomas J. Watson Research Center in the migration process of the instrumentation tool MPItrace to the BG/L system. Mr. Servat has participated in several European research projects and is currently participating in the HPC–Europa project.

José E. Moreira *IBM Systems and Technology Group, 3605 Highway 52 N., Rochester, Minnesota 55901 (jmoreira@us.ibm.com)*. Dr. Moreira received B.S. degrees in physics and electrical engineering in 1987 and an M.S. degree in electrical engineering in 1990, all from the University of São Paulo, Brazil. He received his Ph.D. degree in electrical engineering from the University of Illinois at Urbana–Champaign in 1995. Since joining IBM in 1995, he has been involved in several high-performance computing projects, including the teraflop-scale ASCI Blue-Pacific, ASCI White, and Blue Gene/L. Dr. Moreira was a manager at the IBM Thomas J. Watson Research Center from 2001 to 2004; he is currently the Lead Software Systems Architect for the IBM eServer Blue Gene solution. Dr. Moreira is the author of more than 70 publications on high-performance computing. He has served on various thesis committees and has been the chair or vice-chair of several international conferences and workshops. Dr. Moreira interacts closely with software developers, hardware developers, system installers, and customers to guarantee that the delivered systems work effectively and accomplish their intended missions successfully.