Unlocking the Performance of the BlueGene/L Supercomputer[1]

George Almasi, Sid Chatterjee, Alan Gara, John Gunnels,
Manish Gupta, Amy Henning, Jose E. Moreira, Bob Walkup
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{gheorghe, sc, alangara, gunnels, mgupta, amhennin, jmoreira, walkup}@us.ibm.com

Alessandro Curioni
IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
cur@zurich.ibm.com

Charles Archer
IBM Systems and Technology Group
Rochester, MN
archerc@us.ibm.com

Leonardo Bachega
LARC – University of São Paulo
São Paulo, Brazil
bachega@larc.usp.br

Bor Chan, Bruce Curtis
Lawrence Livermore National Laboratory
Livermore, CA 94550
{chan1, curtisb}@llnl.gov

Sharon Brunett
Center for Advanced Computing Research
California Institute of Technology
Pasadena, CA 91125
sharon@cacr.caltech.edu

Giri Chukkapalli, Robert Harkness, Wayne Pfeiffer
San Diego Supercomputer Center
La Jolla, CA 92093
{giri, harkness, pfeiffer}@sdsc.edu

Abstract

The BlueGene/L supercomputer is expected to deliver new levels of application performance by providing a combination of good single-node computational performance and high scalability. To achieve good single-node performance, the BlueGene/L design includes a special dual floating-point unit on each processor and the ability to use two processors per node. BlueGene/L also includes both a torus and a tree network to achieve high scalability. We demonstrate how benchmarks and applications can take advantage of these architectural features to get the most out of BlueGene/L.

1. Introduction

Achieving high sustained application performance in a scalable environment has been one of the chief goals of the BlueGene/L project [1]. The BlueGene/L (BG/L) system was designed to provide a very high density of compute nodes with a modest power requirement, using a low frequency embedded system-on-a-chip technology. As a consequence of this design approach, the BG/L compute node is targeted to operate at 700 MHz. To obtain good performance at this relatively low frequency, each node needs to process multiple instructions per clock cycle. This can be achieved through two main strategies. First, one can make use of both processors in each BG/L node. Second, each processor has a dual floating-point unit with fused multiply-add instructions, which can perform four operations per cycle using special SIMD-like instructions. We investigate two strategies for leveraging the two processors in each node: coprocessor mode and virtual node mode. We compare the relative merits of these strategies and show that they both achieve significant gains over single processor mode.

[1] 0-7695-2153-3/04 $20.00 ©2004 IEEE

BG/L was also designed to scale to very large sizes, with up to 65,536 compute nodes in the Lawrence Livermore National Laboratory installation. BG/L has multiple networks including a three-dimensional torus for point-to-point communication, and a tree for certain collective operations. For communication on the torus, the best performance will be obtained when there is good locality so that the average number of hops on the torus is small. We examine the communication properties of benchmarks and applications, and show that in some cases very good locality can be obtained by carefully mapping MPI tasks onto the torus. We also demonstrate that real applications do scale well in the BG/L supercomputer.

The rest of this paper is organized as follows. Section 2 gives a brief overview of the BG/L architectural features that are relevant for high application performance in BG/L. Section 3 describes the major techniques we rely on to leverage those features. Section 4 reports on our experiments with benchmarks and applications that demonstrate the effectiveness of those techniques and show that we can achieve high application performance on BG/L. Finally, Section 5 presents our conclusions.

2. BG/L architectural features

This section reviews some architectural features of BG/L that have a significant impact on performance and that are relevant to this study. A more detailed description of the architecture is available elsewhere [1]. The design of BG/L emphasizes both single node computational performance and scalability. For single node performance, we focus on the dual floating-point unit and on using both processors on each node.

2.1 Processor characteristics and memory hierarchy

Each BG/L node has two 32-bit embedded PowerPC (PPC) 440 processors. The PPC 440 processor is a low-power superscalar processor with 32 KB each of L1 data and instruction caches. The data cache is 64-way set associative with 32 byte lines, and a round-robin replacement policy for cache lines within each set. The BG/L nodes support prefetching in hardware, based on detection of sequential data access. The prefetch buffer for each processor holds 64 L1 cache lines (16 128byte L2/L3 cache lines) and is referred to as the L2 cache. Each chip also has a 4 MB L3 cache built from embedded DRAM, and an integrated DDR memory controller. A single BG/L node supports 512 MB memory, with an option to use higher-capacity memory chips.

The PPC 440 design does not support hardware cache coherence at the L1 level. However, there are instructions to invalidate a cache line or flush the cache, which can be used to support cache coherence in software. The memory system beyond the L1 caches supports sequential consistency with respect to the memory system and the hardware locks, with the exception of a fast shared SRAM only accessible from system space.

2.2 Double floating point unit

BG/L employs a SIMD-like extension of the PPC floating-point unit, which we refer to as the double floating point unit or DFPU. The DFPU essentially adds a secondary FPU to the primary FPU as a duplicate copy with its own register file. The second FPU is not an independent unit: it is normally used with a comprehensive set of special parallel instructions. This instruction set includes parallel add, multiply, fused multiply-add, and additional operations to support complex arithmetic. All of the SIMD instructions operate on double-precision floating-point data. There are also SIMD instructions that provide reciprocal estimates and reciprocal square root estimates. These form the basis for very efficient methods to evaluate arrays of reciprocals, square roots, or reciprocal square roots. In addition, the instruction set provides for standard single floating-point usage of the second FPU through extensions.

The special instruction set includes quad-word loads, which bring two consecutive double words (i.e., 16 bytes) from memory into a register pair in the primary and secondary floating-point units. Similarly, there are quad-word store operations. The processor local bus (PLB) supports independent read and write 128-bit data transfers, matching the requirements for efficient quad-word loads and stores.

2.3 Torus interconnect

BG/L has a three-dimensional torus network as the primary interconnect among compute nodes. The torus circuitry, including FIFOs for incoming and outgoing packets, routing and arbitration logic, is integrated within the compute node. Each node has six nearest neighbor connections. The raw hardware bandwidth for each torus link is 2 bits/cycle (175 MB/s at 700 MHz) in each direction. The torus network provides both adaptive and deterministic minimal path routing in a deadlock-free manner. Each message is broken up into one or more packets, with a maximum packet size of 256 bytes, for transmission. The hardware supports variable packet sizes in the range of 32 bytes to 256 bytes in 32 bytes increments.

3. Methods for obtaining high performance

Taking advantage of BG/L's special hardware features is important for achieving good application performance. We first describe how to exploit the double FPU for good uniprocessor performance. We then describe two different techniques for using both processors in each node: coprocessor mode and virtual node mode. We note that both approaches were used successfully on the ASCI Red machine [2]. Finally, we describe how to map applications more effectively onto the three-dimensional torus topology.

3.1 Using the double floating-point unit

The IBM XL family of compilers for Fortran, C, and C++ share a common back-end, TOBEY, which has been enhanced to support code generation for the DFPU core. TOBEY uses extensions to Larsen and Amarasinghe's superword level parallelism (SLP) algorithm [3] to generate parallel operations for the DFPU. In particular, TOBEY can recognize idioms related to basic complex arithmetic floating point computations, and exploit the SIMD-like extensions to efficiently implement those computations. The effectiveness of automatic code generation by the compiler is influenced by several factors. The compiler needs to identify independent floating-point operations using consecutive data on 16-byte boundaries in order to generate SIMD instructions. The issues are different for Fortran and C. For Fortran, the main issue is data alignment. Alignment assertions are required for cases where alignment is not known at compile-time. For C and C++ there can also be pointer aliasing issues. If there is a possible load/store conflict, the compiler cannot safely issue a load for two consecutive data items. In such cases it is necessary to guarantee that pointers are disjoint in memory using `#pragma disjoint` or `#pragma independent`. Alignment assertions are provided for Fortran: `call alignx(16, array_reference)`, and C/C++: `__alignx(16, pointer)`. In applications with statically allocated global data, the alignment and aliasing issues are known at compile-time, and the compiler can generate SIMD code without additional pragmas or alignment assertions. We are in the process of extending the compiler so that it can generate SIMD instructions by applying transformations like loop versioning, which use run-time checks for data alignment [4].

One can also specify SIMD instructions by using intrinsic, or built-in, functions. The compiler recognizes and replaces each such function with the corresponding DFPU instruction. For example, the intrinsic function for the parallel fused multiply-add operation is `__fpmadd()`. With intrinsic functions, one can control the generation of DFPU instructions without resorting to assembler programming.

3.2 Using the coprocessor for computation

By default, the coprocessor is used only to perform communication services. However, the compute node kernel supports a `co_start()` function to dispatch computation to the second processor, and a `co_join()` function to allow the main thread of execution on the first processor to wait for completion of that computation. This mode of using the coprocessor is referred to as coprocessor computation offload mode. Since the hardware does not provide cache coherence at the L1 level, the programmer needs to ensure cache coherence with software. The compute node kernel supports a number of calls to help manage cache coherence. The calls include operations to store, invalidate, or invalidate-and-store all cache lines within a specified address range, and a routine that can evict the entire contents of the L1 data cache. Further details

are described elsewhere [5]. There is significant overhead associated with using the coprocessor to offload computation. It takes approximately 4200 processor cycles to flush the entire L1 data cache; hence this method should only be used for code blocks of sufficient granularity. Also, the code section should be without excessive memory bandwidth requirements and free of inter-node communication.

Ensuring cache coherence in software is a complex and error-prone task. We expect computation off-load mode to be used mainly by expert library developers. We have used this method in Linpack and for certain routines in a subset of Engineering and Scientific Subroutine Library (ESSL) and Math Acceleration Subsystem Vector (MASSV) developed for BG/L.

3.3 Virtual node mode

Virtual node mode provides an easier way to harness the power of the second processor. Virtual node mode splits the available resources in the compute node, assigning half the memory to each processor, and running two separate MPI tasks on each processor. The two tasks share access to L3 and memory, and they share the use of the network. Communication between tasks in the same node happens through a special region of non-cached shared memory. The main disadvantage of virtual node mode is that the available memory for each MPI task is reduced by a factor of 2, and the requirement that the same processor handling the computation must now also handle network tasks such as emptying and filling network FIFOs.

3.4 Mapping MPI tasks onto the torus network

BG/L's main network for point-to-point communication is a three-dimensional torus. Effective communication bandwidth is reduced and the latency is increased as the number of hops on the torus increases, due to the sharing of the links with cut-through traffic. When optimizing the mapping of tasks to the torus, the objective is to shorten the distance each message has to travel. For a relatively small BG/L partition, such as an 8x8x8 torus with 512 nodes, locality should not be a critical factor because even for a random task placement the average number of hops in each dimension is $L/4 = 2$. The issue of task placement on the torus becomes more important for much larger torus sizes.

Many MPI applications have regular communication patterns, and in those cases one can map MPI tasks to torus coordinates to optimize communication performance. This mapping can be done from within or from outside the application. Within the application code, task layout can be optimized by creating a new communicator and re-numbering the tasks, or by using MPI Cartesian topologies. This is done in the Linpack code presented in the Section 4. The implementation of MPI on BG/L allows the user to specify a mapping file, which explicitly lists the torus coordinates for each MPI task. This provides complete control of task placement from outside the application.

4. Performance measurements

In this section we present performance measurements for benchmarks and applications. Some performance measurements were obtained with a 512-node BG/L prototype, running at a reduced frequency of 500 MHz, while other measurements were made on a 512-node system with second-generation chips running at 700 MHz. In our experiments, we explore mathematical kernels, benchmarks including Linpack and the NAS Parallel Benchmarks, and several parallel applications.

4.1 Mathematical kernels and benchmarks

Daxpy is a level-1 BLAS routine that updates one array using values from a second array: $y(i) = a*x(i) + y(i)$, $i=1, \ldots, n$. Similar array update operations appear in many applications, and the daxpy kernel provides a good test of the memory subsystem. There are two loads and one store for each fused multiply-add instruction, so daxpy is load/store bound. By making repeated calls to daxpy in a loop, one can map out the performance characteristics for data in different levels of the memory hierarchy. There are a total of 6 load/store operations for each pair of array updates, using 4 floating-point operations. Without special SIMD instructions, the theoretical limit would be 4 flops in 6 cycles for data in the L1 cache. With the quad-word

load and store operations, this limit becomes 4 flops in 3 cycles. Since there are two processors on each chip with private L1 data caches, the limit is 8 flops in 3 cycles using both processors. The measured performance for daxpy is shown in Figure 1. When one processor is used without SIMD instructions, the observed rate peaks at about 0.5 flops/cycle, using a simple loop and compiler-generated code. This is 75% of the limit of 2/3 flops/cycle. By turning on the `-qarch=440d` compiler option to generate SIMD instructions, the flop rate approximately doubles for data in the L1 cache: for a single processor with SIMD instructions, the measured rate is about 1.0 flops/cycle for array sizes that just fit in L1. Figure 1 also shows the combined floating-point rate for the node, using both processors in virtual node mode. The rate doubles again for data in L1 cache. For large array dimensions, contention for L3 and memory is apparent.

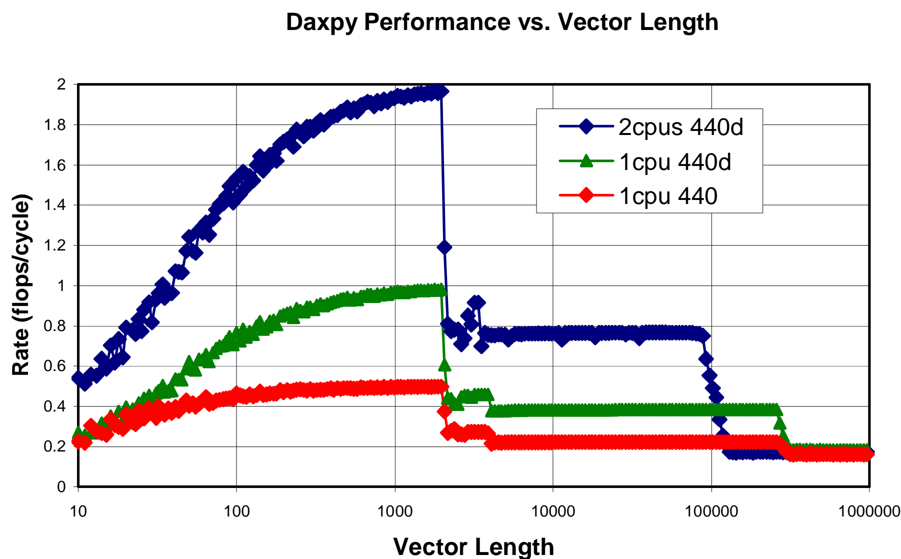

Figure 1. Performance of daxpy on a BG/L node is shown as a function of vector length. L1 and L3 cache edges are apparent. For data in the L1 cache (lengths < 2000), the performance doubles by turning on SIMD instructions (440d), and doubles again when using both processors on the node.

We have examined the effectiveness of virtual node mode by analyzing the performance of the NAS Parallel Benchmarks [6] on a 32-node BG/L system. Figure 2 shows the performance speedup of the NAS Parallel Benchmarks (class C) resulting from virtual node mode. The BT and SP benchmarks require a perfect square for the number of MPI tasks, so those benchmarks used 25 nodes in coprocessor mode, and 32 nodes (64 MPI tasks) in virtual node mode.

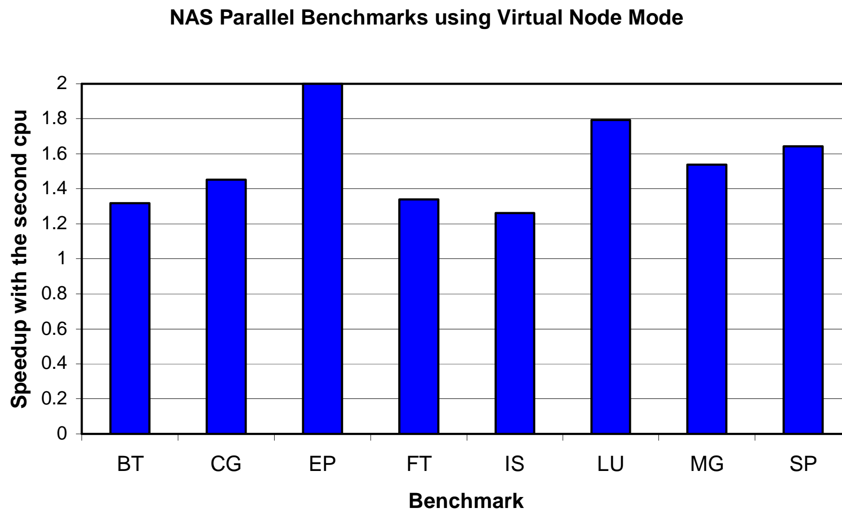

Figure 2. The performance speed-up using virtual node mode is shown for the class C NAS Parallel Benchmarks. The speed-up is defined as the ratio of Mops (million operations per second) per node in virtual node mode to Mops per node using coprocessor mode.

For every NAS benchmark, execution in virtual node mode leads to a significant performance improvement. The speed-ups vary from a factor of two for EP to a factor of 1.26 for IS. There are several reasons why the speed-up is typically less than a factor of two. First, there is a loss in parallel efficiency for each benchmark (except EP) as we increase the number of tasks. Second, there is sharing of physical resources between tasks when we execute two tasks in a compute node. Those resources include L3 cache, main memory, and interconnection networks. There are also changes in the basic computation rate due to a reduction in the problem size per task as the task count increases (the NAS benchmarks solve a fixed total problem size).

The other approach for utilizing both processors in a compute node is to offload some computation to the coprocessor using the `co_start()/co_join()` method. As previously described, this requires careful coordination between the processors, since cache coherence has to be managed entirely in software. Use of the second processor with `co_start()/co_join()` requires more programming effort than virtual node mode, but it can have performance advantages for several reasons. First, one does not need to increase the number of tasks in a parallel job, thus avoiding a loss of parallel efficiency. Second, each task has full access to all the resources of a compute node. This increases the available memory per task, for example.

We compare both approaches to utilizing the two processors in a node with the Linpack benchmark [7]. Figure 3 shows the performance of Linpack as a function of the number of nodes when running in three different modes: default (single processor), virtual node, and computation offload to the coprocessor with `co_start()/co_join()`. Performance is shown as percentage of peak for each machine size, and the results are for weak scaling (constant work per task). We change the problem size with the number of nodes to keep memory utilization in each node close to 70%.

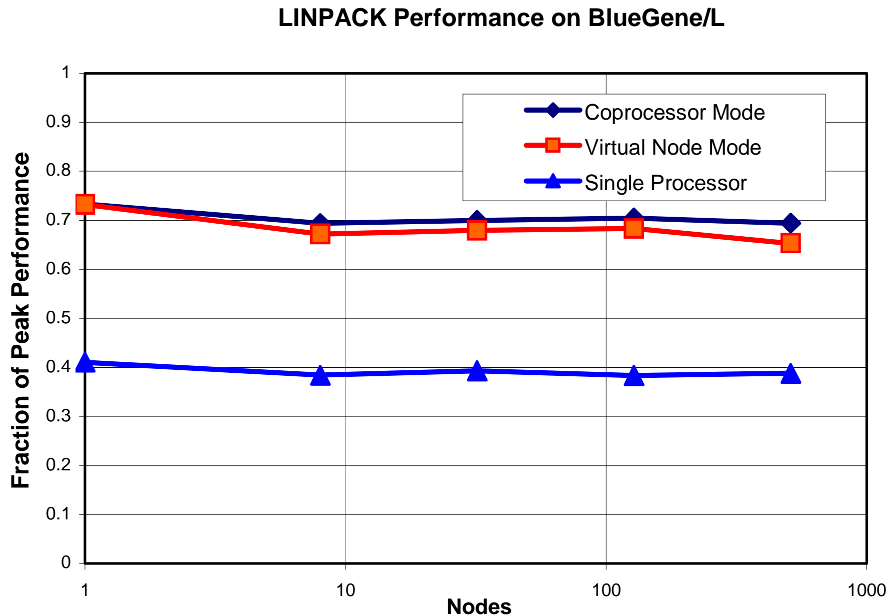

Figure 3. Linpack performance in BG/L is shown as a function of the number of compute nodes. Performance is indicated as a fraction of the theoretical peak. Results for three different strategies are included: using a single processor on each node, offloading computation to the coprocessor with the `co_start()/co_join()` model, and using virtual node mode to run with two tasks per node.

Using a single processor on each node immediately limits the maximum possible performance to 50% of peak. In this mode we consistently achieve 80% of the maximum possible performance, which corresponds to 40% of the peak performance for the node. In virtual node mode and computation offload mode both processors in each node are busy with computations. Both approaches display essentially equivalent performance when running on a single node, achieving 74% of peak. As we increase the machine size, the benefits of computation offload mode start to show up. At the full machine size of 512 nodes, computation offload mode achieves 70% of peak, while virtual node mode achieves 65% of peak.

The NAS parallel benchmarks provide some interesting test cases to study the effects of task placement on the torus network. The NAS BT (block tri-diagonal) benchmark solves Navier-Stokes equations in three dimensions using an alternate-direction implicit finite-difference method. The parallelization strategy distributes blocks to a two-dimensional square process mesh, and the communication pattern is primarily point-to-point communication between neighbors on the process mesh. On BG/L this two-dimensional process mesh is mapped to the three-dimensional mesh of compute nodes. The default mapping is to lay out MPI tasks in XYZ order. Alternatively, one can use a mapping file to explicitly specify the torus coordinates for each MPI task. The optimized mapping lays out contiguous 8x8 XY planes in such a manner that most of the edges of the planes are physically connected with direct links. As the figure shows, optimized mapping has a significant effect on overall performance because of the better physical adjacency of communicating nodes.

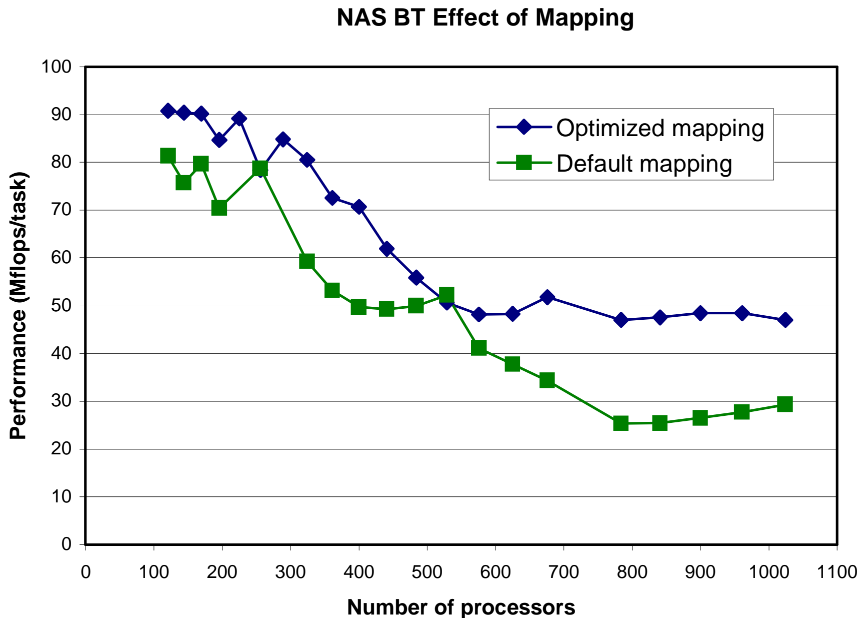

Figure 4. Comparison of the default mapping and optimized mapping for NAS BT on up to 1024 processors in virtual node mode. Mapping provides a significant performance boost at large task counts.

4.2 Applications

We have examined the performance of a number of parallel applications on BG/L. In this section we describe the applications and provide performance comparisons with other parallel platforms when possible.

4.2.1 sPPM

The first example is sPPM, which solves a gas dynamics problem on a three-dimensional rectangular grid using a simplified piece-wise parabolic method. There is a standard version of sPPM which is an ASCI Purple benchmark [8], and there is an optimized version of sPPM, which has been significantly re-written and tuned for IBM Power3/Power4 systems. In the measurements reported here, we use the optimized version of sPPM. The optimized version makes extensive use of routines to evaluate arrays of reciprocals and square roots. On IBM pSeries systems, these routines are supplied in the vector MASS library [9]. On BG/L, we make use of special SIMD instructions to obtain very efficient versions of these routines that exploit the double floating-point unit. The sPPM benchmark is set-up for weak scaling studies (constant work per task). The communication pattern is mainly boundary exchange with nearest neighbors, for all six faces of the local rectangular domain. This problem maps perfectly onto the BG/L hardware, because each node has six neighbors in the 3-d torus network.

The sPPM benchmark is highly scalable because there are extensive computations within each local domain, with a relatively small volume of boundary data for nearest-neighbor communication. The performance is compute-bound for problems that use a significant fraction of the memory available on each node. The measured performance for sPPM is shown in Figure 5, using a 128x128x128 local domain and double-precision variables (this requires about 150 MB of memory). Performance is computed in terms of grid-points processed per second per time-step per processor, or per node for BG/L in virtual node mode. We plot the performance relative to BG/L in coprocessor mode (COP in the plot). For virtual node mode (VNM in the plot), we use a local domain that is a factor of 2 smaller in one dimension and twice as many tasks, thus solving the same problem on a per node basis.

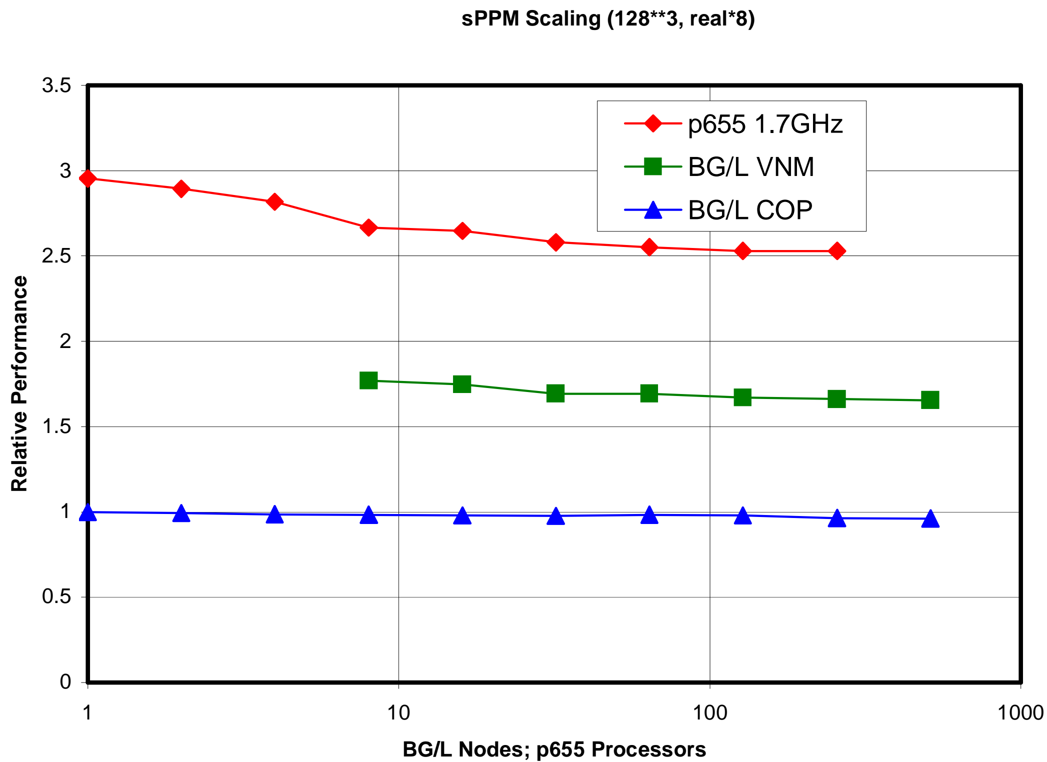

Figure 5. The computational performance for sPPM is shown for systems including IBM p655 at 1.7 GHz (top curve), and BG/L at 700 MHz using virtual node mode (VNM – middle curve) or coprocessor mode (COP – bottom curve) with a single computational task per node (lower curve). The x-axis indicates the number of BG/L nodes or the number of p655 processors.

The scaling curves are relatively flat, indicating very good scalability as expected for this application. The interconnect fabric for the IBM p655 system was the "Federation" switch, with two links per 8-processor node. The p655 system has a good set of hardware counters that provide insight into the computational requirements of sPPM. About 99% of the loads hit in the 32 KB L1 data cache on Power4, and the instruction mix is dominated by floating-point operations. This application makes very good use of the L1 data cache and has a small communication requirement. Less than 2% of the elapsed time is spent in communication routines. As a result, virtual node mode is very effective for sPPM on BG/L. We measure speed-ups of 1.7 – 1.8 depending on the number of nodes. The double floating-point unit on BG/L contributes about a 30% boost to the computational performance through the use of special routines to evaluate arrays of reciprocals and square roots. In other computationally intensive regions of the code, automatic generation of SIMD instructions by the compiler has to date been inhibited by alignment issues or array access patterns. On BG/L, the measured scaling curves remain basically flat up to the largest machine that we have tested so far: 2,048 nodes using 4,096 processors in virtual node mode. On this system, the sustained application performance was approximately 2.1 TFlops for sPPM, which corresponds to about 18% of the theoretical peak of 11.5 TFlops (700MHz * 4 ops/cycle * 4,096 processors).

4.2.2 UMT2K

UMT2K is an ASCI Purple benchmark, which solves a photon transport problem on an unstructured mesh [8]. This application is written in Fortran-90 using MPI and optionally OpenMP. We use an MPI-only implementation, because there is no support for OpenMP on BG/L. (And no plans to support it, given the

complexity created by the noncoherent L1 caches.) The unstructured mesh is statically partitioned using the Metis library [10]. In practice there can be a significant spread in the amount of computational work per task, and this load imbalance affects the scalability of the application. By using profiling tools on IBM pSeries systems, we found that the elapsed time for UMT2K was dominated by a single computational routine, `snswp3d`. The main performance issue in this routine is a sequence of dependent division operations. By splitting loops into independent vectorizable units, the IBM XL compiler was able to generate efficient double-FPU code for reciprocals, resulting in ~40-50% overall performance boost from the double-FPU for this application.

The performance of UMT2K is shown in Figure 6 for BG/L and an IBM p655 cluster (1.7 GHz Power4 processors, Federation switch).

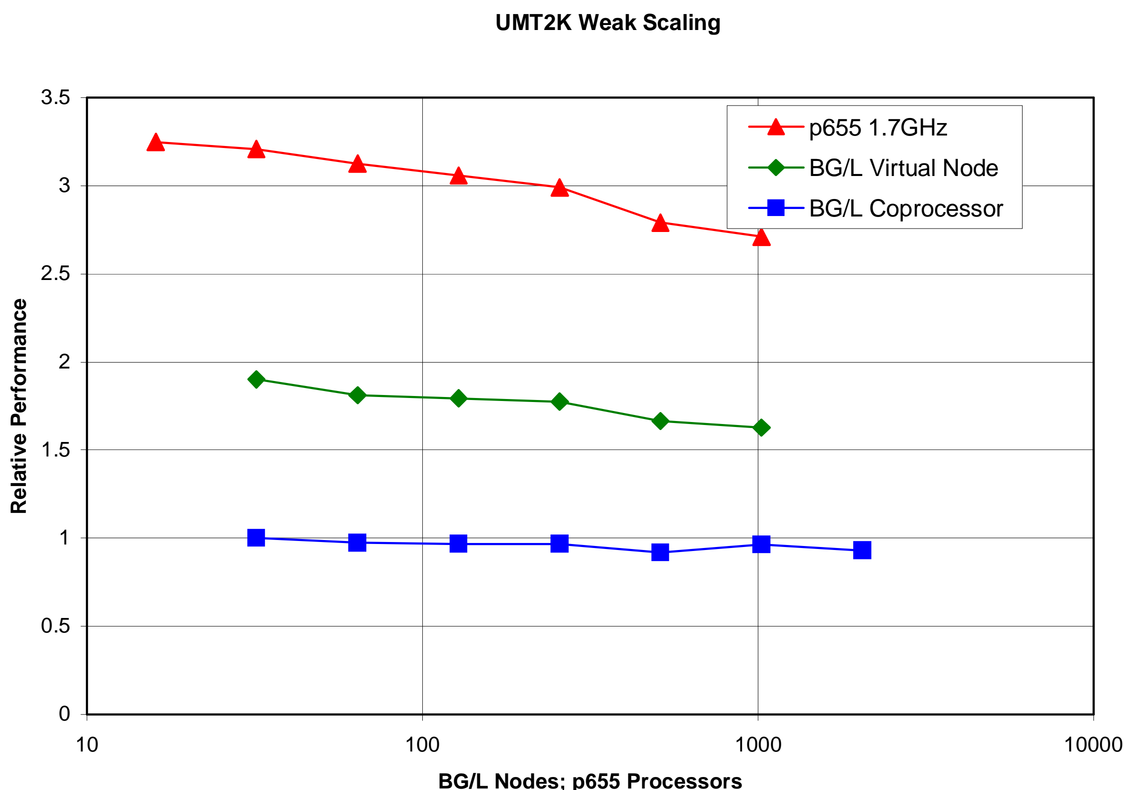

Figure 6. Weak scaling results for UMT2K on BG/L and an IBM p655 cluster. The x-axis indicates the number of BG/L nodes or the number of p655 processors, and the y-axis indicates overall performance relative to 32 nodes of BG/L in coprocessor mode.

The UMT2K test case shown in Figure 6 was modified from the "RFP2" benchmark problem, following the benchmark instructions to keep the amount of work per task approximately constant. UMT2K uses an unstructured mesh, which is partitioned using the Metis library [10]. At the present time, this partitioning method limits the scalability of UMT2K because it uses a table dimensioned by the number of partitions squared. This table grows too large to fit on a BG/L node when the number of partitions exceeds about 4000. This limitation can probably be overcome by using a parallel implementation of Metis. Our measurements show that virtual node mode provides a good performance boost, but the efficiency decreases for large process counts. It should be possible to optimize the mapping of MPI tasks to improve locality for point-to-point communication. Work is in progress to experiment with this approach.

4.2.3 Car-Parrinello Molecular Dynamics (CPMD)

The Car-Parrinello Molecular Dynamics code (CPMD) originated in IBM's Zurich Research Laboratory in the early 1990s [11]. It is an implementation of density functional theory using plane waves and pseudopotentials, and is particularly designed for ab-initio molecular dynamics. The CPMD code is widely used for research in computational chemistry, materials science, and biology. It has been licensed to several thousand research groups in more than 50 countries. The application is mainly written in Fortran, parallelized for distributed-memory with MPI, with an option to add an additional level of parallelism using OpenMP for multi-processor nodes. CPMD makes extensive use of three-dimensional FFTs, which require efficient all-to-all communication. A test case was prepared using a silicon-carbide super-cell with 216 atoms (a system that could also run on a simple workstation). The scaling behavior for this problem was studied on BG/L and on IBM Power4 p690 systems (logical partitions with 8 processors at 1.3 GHz and dual-plane Colony switch). The elapsed times per time-step in the simulation are listed in Table 1.

BG/L nodes p690 processors	p690 sec/step	BG/L sec/step coprocessor	BG/L sec/step virtual node mode
8	40.2	58.4	29.2
16	21.1	28.7	14.8
32	11.5	14.5	8.4
64	n.a.	8.2	4.6
128	n.a.	4.0	2.7
256	n.a.	2.4	1.5
512	n.a.	1.4	n.a.
1024	3.8	n.a.	n.a.

Table 1. The performance for CPMD using a 216 atom SiC supercell is listed for IBM p690 (Power4 1.3 GHz, Colony switch) and BG/L (700 MHz) systems. The performance metric is the elapsed time per time-step in the simulation. Values marked n.a. were not available.

This example is sensitive to latency in the communication subsystem. Small messages become important because the message-size for all-to-all communication is proportional to one over the square of the number of MPI tasks. BG/L out-performs the p690 system when there are more than 32 MPI tasks, because BG/L is more efficient for small messages. The value reported for the p690 system at 1,024 processors is the best-case performance number, using 128 MPI tasks and 8 OpenMP threads per task to minimize the cost of all-to-all communication. Both low latency in the MPI layer and a total lack of system daemons interference contribute to very good scalability on BG/L. The results also show that virtual node mode provides a good performance boost, up to the largest task counts tested (512 MPI tasks).

4.2.4 Enzo

Enzo is a large-scale multi-physics code for astrophysical simulation. Its primary application is modeling cosmological processes, with particular emphasis on large-scale structure formation where self-gravity is the dominant force. Enzo combines hydrodynamics for baryonic matter (which is treated as a real gas) and particle tracking for dark matter (which is treated as collisionless particles). The global gravitational field is due to the real gas and the dark matter particles. Enzo can operate in full adaptive mesh refinement (AMR) mode or in a non-adaptive "unigrid" mode. Numerical methods [12] include the piecewise parabolic method

(PPM) or Zeus hydrodynamics solvers (using multigrid), Fourier transforms for the global gravity solution, and various Newton iteration schemes for the real-gas effects (ionization, heating and cooling, etc.).

Enzo consists of about 35,000 lines of Fortran 90 and 65,000 lines of C++. The C++ code is primarily for managing the AMR grids and I/O. Most of the computation is performed in Fortran. The problem is parallelized by domain decomposition into rectangular subgrids, including the top/root grid (which is the only level in a non-AMR run). Message passing is done with MPI, and I/O makes use of the HDF5 data format. Enzo runs on many currently available high-end computers and is a major consumer of compute cycles at the San Diego Supercomputer Center, where it runs primarily on DataStar, a cluster of IBM Power4 processors with a peak speed of over 10 TFlops.

The main challenge in porting Enzo to BG/L was to build the HDF5 I/O library, which required extra care due to the cross-compiling environment. The version of HDF5 that was built supported serial I/O and 32-bit file offsets. These limitations can be addressed by future improvements in BG/L system software. The initial build of Enzo had very poor performance on BG/L. The problem was identified using MPI profiling tools that are available on BG/L. The performance issue was related to the way that non-blocking communication routines are completed. Enzo used a method based on occasional calls to MPI_Test. This had been observed to perform poorly on other systems that use MPICH. It was found that one could ensure progress in the MPI layer by adding a call to MPI_Barrier. On BG/L, this was absolutely essential to obtain scalable parallel performance, and it provided a performance improvement on some other systems as well. After getting Enzo up and running on BG/L hardware, experiments were done to explore ways of increasing performance. A ~30% performance improvement from the double-FPU was obtained by adding calls to optimized routines for vectors of reciprocals and square roots. Virtual node mode was also found to be very effective for cases that could fit in the available memory (256 MB per virtual node, which is half of the physical node memory).

The relative performance of Enzo for a test case using a 256**3 unigrid is shown in Table 2 for 32 and 64 nodes of BG/L and the corresponding numbers of IBM p655 processors (1.5 GHz Power4, Federation switch).

BG/L nodes or p655 processors	Relative speed of BG/L in coprocessor mode	Relative speed of BG/L in virtual node mode	Relative speed of p655 1.5 GHz
32	1.00	1.73	3.16
64	1.83	2.85	6.27

Table 2. Performance of Enzo for 256**3 unigrid on BG/L and IBM p655 (1.5GHz Power4, Federation switch) relative to 32 BG/L nodes in coprocessor mode.

This test case was computation-bound at the task counts that were used in the table. In coprocessor mode, one BG/L processor (700 MHz) provided about 30% of the performance of one p655 processor (1.5 GHz). This is similar to what we have observed with other applications. Virtual node mode gave a significant speedup by a factor of 1.73 on 32 BG/L nodes.

Scaling studies for Enzo were attempted on both systems. It was found that for a fixed problem size, scalability of Enzo is limited by bookkeeping work (integer-intensive operations) in one routine, which increases rapidly as the number of MPI tasks increases. This limits strong scaling on any system. Weak scaling studies were also attempted using a larger grid (512**3). On BG/L, this failed because the input files were larger than 2 GBytes, and the available runtime did not fully support large files. This experience makes it clear that large file support and more robust I/O throughput are needed for wider success of BG/L.

4.2.5 Polycrystal

The polycrystal application simulates grain interactions in polycrystals including the accurate resolution of inhomogeneous anisotropic elastic and plastic fields at grain boundaries, and multi-scale mechanisms of grain boundary sliding. The overall approach is to exploit the power of massively parallel computing – such as offered by BG/L and DOE ASCI platforms – to explicitly account for micro-structural effects on material behavior. The computational approach is based on a Lagrangian large-deformation finite element formulation. Multi-scale, atomistically-informed crystal plasticity models and equations of state computed from ab-initio quantum mechanical calculations are combined with a shock-capturing method to analyze the response of tantalum under extreme loading conditions. The considerable computing effort is distributed among processors via a parallel implementation based on mesh partitioning and message passing. Each mesh partition represents a grain with a different orientation and is assigned to a different processor.

Porting polycrystal to BG/L was challenging. The build environment for polycrystal requires Python, and has a complex set of environment variables and configuration files, with built-in assumptions about support for shared libraries. It was necessary to change the build process to make use of static libraries only, and it was also necessary to skip certain utilities that are normally built with polycrystal (such as MPI-Python).

Polycrystal has a requirement that a global grid must fit within the memory of each MPI process. For interestingly large problems, this requires several hundred Mbytes of memory, which is more than the available memory in virtual node mode. As a consequence, it was necessary to use coprocessor mode on BG/L. The computational work in polycrystal does not benefit from library calls to routines that are optimized for the double-FPU, and the compiler was not effective at generating double-FPU code due to unknown alignment of the key data structures. The overall result was that polycrystal could use just one floating-point unit on one of the two processors in each BG/L node. Polycrystal was observed to scale moderately well. Using a fixed problem size, the performance improved by about a factor of 30 going from 16 to 1,024 processors. Measurements with hardware counters, combined with MPI instrumentation, showed that the scalability was limited by considerations of load balance, not message-passing or network performance. The performance of polycrystal on BG/L (700 MHz) was measured to be a factor of 4 to 5 slower per processor compared to an IBM p655 cluster (1.7 GHz Power4, Federation interconnect). The experience with polycrystal makes it clear that available memory is a major constraint, and that there are important applications that will not be able to make use of virtual node mode.

5. Conclusions

In this paper we have described various techniques for achieving high benchmark and application performance on the BG/L supercomputer.

The use of optimized math libraries and the availability of double floating-point optimizations in the XL compiler suite make it possible for applications to benefit from the dual floating point unit in each processor. The paper shows large gains on BLAS level 1 operations resulting from the effective use of the DFPU. Our experience to date has been that optimized math libraries often provide the most effective way to use the DFPU. Success with automatic DFPU code generation by the compiler in complex applications has been limited.

The computational power of the second processor can be harnessed using coprocessor computation offload and virtual node modes. The results shown in the paper demonstrate close to doubling of the performance achieved by the Linpack benchmark, by deploying either computation offload or virtual node mode. For benchmarks and applications where computation offload mode is not a viable option, such as the NAS parallel benchmarks, virtual node mode shows substantial gains. It often achieves between 40% to 80% speedups.

As the torus dimension increases, locality will be an important factor for communication performance. By judiciously mapping MPI tasks to locations in the physical network, we can achieve substantial performance improvements.

This paper provides insight into some of the techniques we have found useful, but this process is only at the beginning. As the size of the machine available to us increases, we will be concentrating on techniques to scale existing applications to tens of thousands of MPI tasks in the very near future. There are also efforts underway toward automating some of the performance enhancing techniques allowing for faster and more efficient application porting.

References

[1] N. R. Adiga et al. An overview of the BlueGene/L supercomputer. In *SC2002 – High Performance Networking and Computing*, Baltimore, MD, November 2002.
[2] ASCI Red Homepage. http://www.sandia.gov/ASCI/Red/.
[3] S. Larsen and S. Amarasinghe, Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 145-156, June 2000
[4] A. Eichenberger, P.Wu, and K. O'Brien. Vectorization for short SIMD architectures with alignment constraints. In *Proceedings of SIGPLAN 2004 Conference on Programming Language Design and Implementation*,Washington, D.C., June 2004.
[5] G. Almasi, L. Bachega, S. Chatterjee, D. Lieber, X. Martorell, and J. E. Moreira. Enabling dual-core mode in BlueGene/L: Challenges and solutions. In *Proceedings of 15th Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brazil, November 2003.
[6] NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB.
[7] The Linpack Benchmark. http:/www.netlib.org/benchmark/top500/lists/linpack.html.
[8] ASCI Purple Benchmark Page. http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html.
[9] IBM Mathematical Acceleration Subsystem. http://techsupport.services.ibm.com/server/mass.
[10] Metis home page. http://www-users.cs.umn.edu/~karypis/metis/index.html.
[11] CPMD home page. http://www.cpmd.org
[12] Enzo Home Page, http://cosmos.ucsd.edu/enzo .