Performance Analysis and Optimization of Parallel Scientific Applications on CMP Cluster Systems

Xingfu Wu, Valerie Taylor, Charles Lively, and Sameh Sharkawi
Department of Computer Science, Texas A&M University, College Station, TX 77843
Email: {wuxf, taylor}@cs.tamu.edu

Abstract

Chip multiprocessors (CMP) are widely used for high performance computing. Further, these CMPs are being configured in a hierarchical manner to compose a node in a cluster system. A major challenge to be addressed is efficient use of such cluster systems for large-scale scientific applications. In this paper, we quantify the performance gap resulting from using different number of processors per node; this information is used to provide a baseline for the amount of optimization needed when using all processors per node on CMP clusters. We conduct detailed performance analysis to identify how applications can be modified to efficiently utilize all processors per node on CMP clusters, especially focusing on two scientific applications: a 3D particle-in-cell, magnetic fusion application Gyrokinetic Toroidal Code (GTC) and a Lattice Boltzmann Method for simulating fluid dynamics (LBM). In terms of refinements, we use conventional techniques such as cache blocking, loop unrolling and loop fusion, and develop hybrid methods for optimizing MPI_Allreduce and MPI_Reduce. Using these optimizations, the application performance for utilizing all processors per node was improved by up to 18.97% for GTC and 15.77% for LBM on up to 2048 total processors on the CMP clusters.

1. Introduction

The current trend in parallel computing systems has been shifting towards cluster systems with CMPs (chip multiprocessors). Further, the CMPs are usually configured hierarchically (e.g., multiple CMPs compose a multi-chip module and multiple multi-chip modules compose a node) to compose a node of a CMP cluster. For example, with the DataStar P690 at the San Diego Supercomputing Center (SDSC) each node consists of 4 multi-chip modules (MCMs); each MCM consists of 4 CMPs; each CMP consists of 2 processors for a total of 32 processors per node [10]. When using these clusters to execute a given application, one issue to be addressed is how to efficiently utilize all processors per node given the significant sharing of node resources (e.g., caches, memory, networks) among the processors within the node. In this paper, we quantify the performance gap resulting from using different number of processors per node for two scientific applications: a 3D particle-in-cell application Gyrokinetic Toroidal Code (**GTC**) [2], which is a flagship DOE SciDAC fusion microturbulence code, and a Lattice-Boltzmann Method widely used for simulating fluid dynamics (**LBM**) [13]. The descriptions of the two applications are shown in Table 1. The problem size for LBM is a 3D mesh computational domain. The test case studied for GTC is 100 particles per cell and 100 time steps. Further, we use the detailed performance results to identify performance optimizations that can be made for efficient execution using all processors per node on the CMP clusters.

Table 1. Descriptions of GTC and LBM

Application	Discipline	Problem Sizes	Languages
GTC	Magnetic Fusion	100particles/cell 100 time steps	Fortran90, MPI
LBM	Fluid Dynamics	128x128x128 512x512x512	C, MPI

Other work in this area has focused on using all processors per node with systems that consists of up to 4 processors per node. Phillips et al. [8] presented the performance results for 4 processors per node and 3 processors per node on Lemieux Alpha cluster at Pittsburgh Supercomputing Center, and noted that leaving idle one processor per node reduces performance variability. Petrini et al. [7] found that application execution times may vary significantly between 3 processor per node and 4 processors per node on a large scale supercomputer, ASCI Q, and concluded that system noise within the nodes was the source of the performance variability. In our previous work [14], we proposed processor partitioning, and investigated processor partitioning impacted the performance of NAS parallel benchmarks on SMP clusters. In this paper, however, we use systems with up to 32 processors per node. Further, we identify optimizations that result in efficient use of all processors per node on the CMP clusters.

The experiments conducted in this work utilize systems with different number of processors per node. At SDSC, DataStar P655 has 8 processors per node and DataStar P690 has 32 processors per node. At the DOE National Energy Research Scientific Computing Center (NERSC) Bassi has 8 processors per node and Seaborg has 16 processors per node [6]. At the Renaissance

Computing Institute BlueGene/L has 2 processors per node [9]. Further, each system has a different node memory hierarchy. The experimental results indicate that large performance gaps exist. For example, for LBM using a total of 32 processors on the SDSC DataStar P690 there is an increase in execution time of 23.53% when using 1 node with 32 processors versus 4 nodes with 8 processors per node (corresponding to less sharing of node resources) for a problem size of 128x128x128.

Much of the computation involved in parallel scientific applications occurs within nested loops. Therefore, loop optimization is fundamentally important for these applications. In this paper, we use cache blocking, loop unrolling and loop fusion to optimize the scientific applications on CMP clusters in order to take advantage of advanced memory hierarchy. We also develop hybrid methods to optimize MPI_Allreduce and MPI_Reduce, which are common collective communication subroutines. For a given optimization of GTC, the application performance was improved by up to 18.97% on up to 2048 processors. For a given optimization of LBM, the application performance was improved by up to 15.77% when utilizing all processors per node. This is important because access to a large number of processors only occurs when using all processors per node for CMP cluster systems.

The remainder of this paper is organized as follows. Section 2 describes the architecture and memory hierarchy of the five supercomputers used in our experiments, and presents their MPI performance when utilizing different configurations of the node hierarchy. Section 3 discusses application optimization methods used to refine the applications. Section 4 investigates performance characteristics of GTC, and presents our optimization results. Section 5 discusses performance characteristics and optimization results of LBM. Section 6 concludes this paper. In this paper, we describe the system configuration as **MxN whereby M denotes the number of nodes with N processors per node (PPN)**. The job scheduler for each supercomputer always dispatches one process to one processor. All experiments were executed multiple times to insure consistency of the performance data. Prophesy system [12] is used to collect application performance data.

Table 2. Specifications of five supercomputer architectures

Configurations	P655	BlueGene/L	P690	Seaborg	Bassi
Total CPUs	2176	2048	224	6080	888
Total Nodes	272	1024	7	380	111
MCMs/Node	1	NA	4	NA	NA
chips/MCM	4	NA	4	NA	NA
Cores/chip	2	2	2	1	1
CPUs / Node	8	2	32	16	8
CPU type	1.5, 1.7GHz POWER4	700 MHz PowerPC	1.7GHz POWER4	375MHz POWER3	1.9GHz POWER5
Memory/Node	16, 32GB	1GB	128GB	16-64GB	32GB
L1 Cache/CPU	64/32 KB	32KB	64/32 KB	32/64 KB	64/32 KB
L2 Cache/chip	1.41MB	16 128-byte lines	1.41MB	8MB	1.92MB
L3 Cache/chip	32MB	4MB	32MB	NA	36MB
Network	Federation	3D Torus	Federation	Colony	Federation

2. Execution Platforms

Details about the five supercomputers used for our experiments are given in Table 2. These systems differ in the following main features: number of processors per node, configurations of node memory hierarchy, CPU speed, multi-core chips, and communication networks.

DataStar P655 has 176 (8-way) compute nodes with 1.5GHz P4 and 16GB memory, and 96 (8-way) compute nodes with 1.7GHz P4 and 32GB memory [10]. Each node of the P655 has 1 MCM with 4 chips per MCM. DataStar P690 has 7 (32-way) compute nodes. Each node of P690 has 4 MCMs with 4 chips per MCM. The use of 8-way nodes for P655 is exclusive. The use of 32-way nodes on P690 is shared among users. Further, access to the P690 is limited to five nodes.

RENCI BlueGene/L [9] is a system-on-chip supercomputer. The BlueGene/L has 1024 dual 700 MHz PowerPC 440 nodes with 1GB memory per node. Note that the L2 cache is a prefetch buffer which holds 16 128-byte lines.

NERSC Seabog [6] is a distributed memory computer with 6,080 processors, 16 PPN and a shared memory pool of size 16-64GB (312 nodes have 16GB; 64 nodes have 32GB; 4 nodes have 64GB). The use of 16-way nodes is exclusive. Bassi [6] is a distributed shared memory computer with 888 processors, 8 PPN and a shared memory pool of 32GB per node for exclusive access. Only Bassi is configured to use 20GB large page size on each node.

2.1 Multi-Chip Module and Processor Affinity

In this section, we address processor affinity (i.e., how processes are dispatched to processors) and its policy in the CMP clusters: P655, P690, and BlueGene/L. Although Bassi is POWER5 system, it is configured

with only one active core per POWER5 chip. Seaborg is a POWER3 system with one processor per chip.

Figure 1 shows the logical view of a MCM. Each MCM has four POWER4 chips; the two processors on the same chip share L2 and L3 caches. The logical interconnection of four POWER4 chips is point-to-point with uni-directional buses connecting each pair of chips to form an 8-way SMP with an all-to-all interconnection topology. The MCMs are used as eight-way basic building blocks to form larger SMPs, such as P690 with four MCMs, by extending each bus from each module to its neighboring module in one direction.

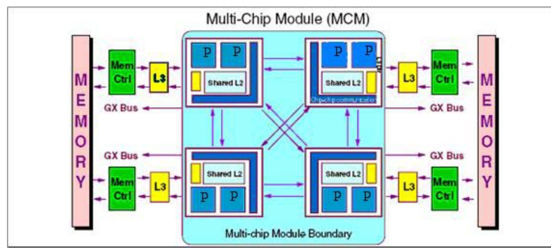

Figure 1. A logical view of a MCM [1]

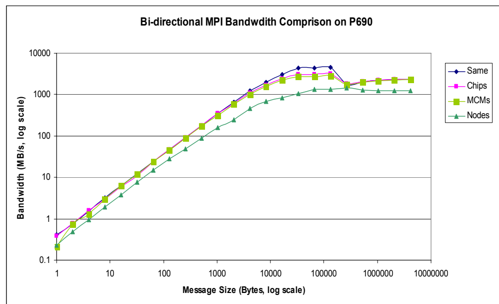

Figure 2. Bi-directional bandwidth comparison on P690 using processor binding

The processor affinity policy for both POWER4 and POWER5 is discussed in [3]. When a virtual processor is dispatched, it is first dispatched onto the same physical processor that it last ran on. Otherwise, it will be dispatched onto the first available processor in the following order: on the same chip, then to another chip on the same multi-chip module (MCM), then to a chip on the same node.

Figure 2 presents the bi-directional MPI bandwidth comparison on P690 using processor binding (e.g., we use IBM AIX command *bindprocessor* to implement processor binding and use Intel's IMB benchmark Sendrecv [5] to measure bi-directional bandwidth). We use the term **PPM** to denote processors per MCM and **PPC** to denote processors per chip. We measure the MPI bandwidth at the following levels: within a chip (**Same**), across two chips on one MCM (**Chips**), across two MCMs (**MCMs**) or across two nodes (**Nodes**) on P690. The results indicate that using two processors within a chip results in much better bandwidth than using one processor per chip (on the same MCM) or using one processor per MCM or per node for small or medium message sizes in the range of 1 byte to 256KB messages. It is interesting to note that the bandwidth for the message size of 256KB or larger on P690 is very similar for using two processors in the same chip or using two processors across different chips (but on the same MCM) or using two processors on the different MCMs of the same node; using two nodes still resulted in poor bandwidth for large message sizes.

3. Optimization Methods

Much of the computation involved in parallel scientific applications such as LBM and GTC occurs within nested loops. Therefore, loop optimization is fundamentally important for such applications. In this section, we discuss the details of well-known techniques of cache blocking, loop unrolling and loop fusion to optimize the scientific applications. We also present our hybrid methods to optimize MPI_Allreduce and MPI_Reduce.

Cache blocking is a well-known loop optimization technique to aid in taking advantage of memory hierarchy; its main purpose is to eliminate as many cache misses as possible. This technique transforms the memory domain of an application into smaller chunks, such that computations are executed on the chucks that easily fit into cache to maximize data reuse. The optimal loop block size varies with different applications on different systems.

Loop unrolling is a well-known code transformation technique that replicates the original loop body multiple times, adjusts the loop termination code and eliminates redundant branch instructions. Outer loop unrolling can increase computational intensity and minimize load/stores, while inner loop unrolling can reduce data dependency and eliminate intermediate loads and stores. We combine inner and outer loop unrolling to optimize the scientific applications.

The performance for the hand-tuned scientific codes using cache blocking and unrolling can be further improved by using compiler directives, which are hardware-specific. The IBM XL Fortran compiler provides some hardware-specific directives for performance optimization, such as UNROLL_AND_FUSE [3]. The directive instructs the compiler to allow loop unrolling and fusion where applicable. Loop fusion is also a code transformation technique that minimizes the required number of loop iterations, and improves data locality by increasing data reuse in registers and cache.

In the scientific applications such as GTC, MPI_Allreduce dominates the most communication time. In order to optimize MPI_Allreduce, we implement the following hybrid method:

- Measure the performance of the original MPI_ Allreduce with various message sizes on a cluster,
- Measure the performance of Rabenseifner's algorithm for allreduce in [11] on the cluster,
- Compare both performance to decide message size ranges for the best performance,
- Implement a hybrid method for MPI_Allreduce based on the message size ranges (basically using the original algorithm at very small message sizes and the Rabenseifner's algorithm otherwise).

Rabenseifner's algorithm performs a reduce-scatter followed by an allgather for MPI_Allreduce, and it also performs a reduce-scatter followed by a gather for MPI_Reduce. The hybrid method takes different systems and message sizes into account in order to optimize MPI_Allreduce. We also use the similar hybrid method to optimize MPI_Reduce. A MPI program is required to be recompiled with our own MPI library for the two subroutines Allreduce and Reduce.

4. Gyrokinetic Toroidal Code (GTC)

The Gyrokinetic Toroidal code (GTC) [2] is a 3D particle-in-cell application developed at the Princeton Plasma Physics Laboratory to study turbulent transport in magnetic fusion. GTC is currently the flagship DOE SciDAC fusion microturbulence code. The problem sizes for the GTC code used in our experiments all are 100 particles per cell and 100 time steps. Table 3 shows the detailed dataset configurations, where micell is the number of ions per grid cell, mecell is the number of electrons per grid cell, mzetamax is the total number of toroidal grid points, and npartdom is the number of particle domain partitions per toroidal domain.

Table 3. Datasets with scaling the number of processors for GTC

Processors	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
micell	100	100	100	100	100	100	200	400	800	1600	3200
mecell	100	100	100	100	100	100	200	400	800	1600	3200
mzetamax	2	4	8	16	32	64	64	64	64	64	64
npartdom	1	1	1	1	1	1	2	4	8	16	32

Table 4. Performance comparison of GTC on 2 processors on P655

Metrics	**Across 2 nodes**	**Across 2 chips**	**Within a chip**
	2x1/1PPC	**1x2/1PPC**	**1x2/2PPC**
Runtime (% difference)	984.26 (baseline)	990.02 (0.59%)	1114.32 (13.21%)
L1 hit rate	92.356%	92.375%	92.386%
TLB miss rate	0.005%	0.005%	0.005%
L2 bandwidth/processor	4618.906MB/s	4582.185MB/s	4071.585MB/s
% accesses from L2	2.228%	2.214%	2.235%

Table 5. Performance comparison of GTC on 2 processors on P690

Metrics	**Across 2 nodes**	**Across 2 MCMs**	**Across 2 chips**	**Within a chip**
	2x1/1PPM-1PPC	**1x2/1PPM-1PPC**	**1x2/2PPM-1PPC**	**1x2/2PPM-2PPC**
Runtime (% difference)	980.23 (baseline)	994.58 (1.46%)	1001.49 (2.17%)	1009.66 (3.00%)
L1 hit rate	92.515%	92.48%	92.457%	92.475%
TLB miss rate	0.005%	0.005%	0.005%	0.005%
L2 bandwidth/processor	4578.505 MB/s	4499.834 MB/s	4491.888 MB/s	4457.119 MB/s
% accesses from L2	2.211%	2.181%	2.177%	2.16%

Table 6. Execution times (seconds) of GTC for different PPN on 64 processors

System	**64x1**	**32x2**	**16x4**	**8x8**	**4x16**	**2x32**
Seaborg	3545.08	3549.41	3574.92	3617.47	3773.43	NA
P655	1203.32	1253.83	1266.49	1305.74	NA	NA
BlueGene/L	3937.46	--	NA	NA	NA	NA
Bassi	--	875.11	886.54	894.64	NA	NA
P690	NA	NA	NA	NA	1210.26	1297.12

4.1 Application Performance Analysis

Table 4 provides the performance results from P655 for dispatching two processes to two processors on the same chip, on two different chips on the same MCM, or across two nodes respectively. The percentage given next to runtime (**unit: seconds**) corresponds to the increase in comparison to the baseline configuration corresponding to using one processor per node (or the least amount of sharing of node resources). There is 13.21% increase in execution time when using two processors within the same chip (1x2/2PPC) due to contention resulting from the sharing of resources, such as L2 and L3 caches. For the case of using one processor per chip with two chips on the same MCM (1x2/1PPC), there is a very small increase in execution time (only 0.59%). Recall that each node on P655 has

one MCM, which consists of four dual-core POWER4 chips shown in Table 2. Table 4 also presents the hardware counters' performance using *hpmcount* [4]. The hardware counters indicate large differences in L2 bandwidth per processor for the two cases of using one processor per node versus using two processors within the same chip. Hence, there is significant contention for L2 when using two processors within the same chip.

Table 5 shows the performance results for dispatching two processes to two processors on the same chip, on two different chips on the same MCM, or on two different chips on two different MCMs on P690, respectively. Each node on P690 has four MCMs with four dual-core POWER4 chips per MCM shown in Table 2. There is only a 3.00% increase in execution time when using two processors within the same chip (1x2/2PPC) due to contention resulting from the sharing of resources, such as L2 and L3 caches. Comparing Table 5 with Table 4, it is clear that L2 bandwidth per processor accounts for the performance difference when using two processors within one chip (1x2/2PPC) versus using two processors across nodes (2x1/1PPM-1PPC).

Table 6 indicates the performance gaps resulting from using different PPN on a total of 64 processors on the five supercomputers. The highlighted time corresponds to the smallest execution time among the different configurations. As expected, the smallest time corresponds to the minimum amount of sharing of node resources. With increasing number of PPN, the application execution time increases. The performance data in Table 6 utilized default processor affinity described in Section 2.1. For example, with the 16x4 configuration on P655 (16 nodes with 4 processors per node), the 4 processors per node were dispatched such that the only two chips were used, whereby both cores per chip were used. The time difference between the schemes 64x1 and 8x8 is 8.51% on P655. The time difference between the schemes 64x1 and 4x16 is 6.44% on Seaborg. The time difference between the schemes 32x2 and 8x8 is 2.23% on Bassi. The time difference between the schemes 4x16 and 2x32 is 7.18% on P690. Because the GTC code could not be executed on UNC RENCI BlueGene/L with 2 processors per node (in VN mode) except 2048 processors, we could not collect any data for using 2 PPN on the machine. We find that the subroutines *pushi* and *chargei* in GTC take more than 90% of the application execution time on 64 processors, and are sensitive to different memory access patterns and communication patterns for different PPN.

Table 7 illustrates that impact of using processor binding to reduce contention of chip resources. Using one processor per chip (1PPC) results in approximately 3% decrease in execution time versus using two processors per chip (2PPC) for both 32x2 and 16x4 configurations. While the difference is small, it is the case that processor binding can aid task schedulers in maximizing the application performance in dedicated usage of CMP nodes.

Table 7. Performance comparison of GTC on P655 using processor binding

#MxN	2PPC	1PPC	% difference
32x2	1253.83	1218.31	2.92
16x4	1266.49	1230.47	2.93

4.2 Application Performance Optimization

Much of the computation involved in GTC, especially the subroutines *pushi* and *chargei* which dominate the most of the total execution time of GTC on up to 2048 processors in the datasets we used, occurs within nested loops. Therefore, we use cache blocking, loop unrolling and loop fusion to optimize the application. Because MPI_Allreduce dominates the most communication time of the code on large number of processors. Therefore, we use the hybrid method described in Section 3 to optimize the MPI_Allreduce. The range of message sizes are defined on different systems as follows: for P655, the hybrid method uses the original algorithm for the message sizes of less than 1KB and the Rabenseifner's algorithm otherwise. For BlueGene/L, because GTC could only be executed on the number of processors with 1PPN, the hybrid method uses the Rabenseifner's algorithm for the message sizes of between 256 bytes and 2KB and the original algorithm otherwise.

Table 8. Execution time (seconds) of GTC between the original and the optimized on BlueGene/L

#Procs (MxN)	original	optimized	% improvement
8 (8x1)	3804.03	3082.54	18.97%
16 (16x1)	3834.98	3124.80	18.52%
32 (32x1)	3869.60	3166.56	18.17%
64 (64x1)	3937.46	3221.31	18.19%
128 (128x1)	3919.06	3202.81	18.28%
256 (256x1)	3913.07	3208.95	17.99%
512 (512x1)	3820.28	3120.65	18.31%
1024 (1024x1)	3788.49	3096.36	18.27%
2048 (1024x2)	3808.03	3108.40	18.37%

Table 9. Execution times (seconds) of GTC between the original and the optimized on P655

#Procs (MxN)	original	optimized	% improvement
8 (1x8)	1207.07	1144.73	5.16%
16 (2x8)	1242.56	1174.80	5.45%
32 (4x8)	1273.75	1203.32	5.53%
64 (8x8)	1305.73	1240.52	4.99%
128 (16x8)	1263.71	1201.99	4.88%
256 (32x8)	1237.22	1177.27	4.85%
512 (64x8)	1228.28	1172.25	4.56%

Our experimental results indicate that the optimal loop block size for GTC is 2x2. This block size is used to optimize outer loops of the triple nested loops in GTC code. We also unroll the inner loops four times for four

major double nested loops in the subroutines *pushi* and *chargei* such that the double nested loops are restructured into single loops. Further, we used the compiler options consistent with that given in [2].

The results of applying the aforementioned optimizations are given in Tables 8 and 9 for BlueGene/L and P655, respectively. We performed the optimization in series so that we could quantify the results of each optimization. These results are not given in this paper due to space. The results, however, indicated that majority of the optimization for GTC resulted from hand-tuned loop unrolling and blocking. Overall, we achieve up to 18.97% performance improvement on BlueGene/L. It should be noted that the performance gap on P655 using 64x1 versus 8x8 was 8.51% in Table 6. Using our optimizations we were able to reduce the execution time for using all processors per node by 4.99%. Note that the same optimization resulted in much smaller performance improvement for 64x1 than that for 8x8 on P655.

5. LBM Application

The Lattice Boltzmann method (LBM) is widely used for simulating fluid dynamics; this method has the ability to deal efficiently with complex geometries and topologies. The LBM application is computation intensive. In our simulations, we use the D3Q19 lattice model (19 velocities in 3D) with the collision and streaming operations [13].

5.1 Application Performance Analysis

Table 10 provides the execution times of the LBM application with problem sizes of 128x128x128 and 512x512x512 for different PPN on a total of 32 processors across the five supercomputers. The performance data was collected using the default processor affinity described in Section 2.1. The highlighted time corresponds to the smallest execution time among the different PPN. With increasing number of PPN, the application execution time increases significantly on Seaborg, P655, and P690; there is very little difference in execution times on Bassi. For example, the time difference between the 1 PPN and 16 PPN on Seaborg is 21.3% for the problem size of 512x512x512, and 15.83% for the problem size of 128x128x128. The time difference between the 1 PPN and 8 PPN on P655 is 20.73% for the problem size of 512x512x512, and 10.03% for the problem size of 128x128x128. The time difference between the 8 PPN and 32 PPN on P690 is 17.79% for the problem size of 512x512x512, and 23.53% for the problem size of 128x128x128.

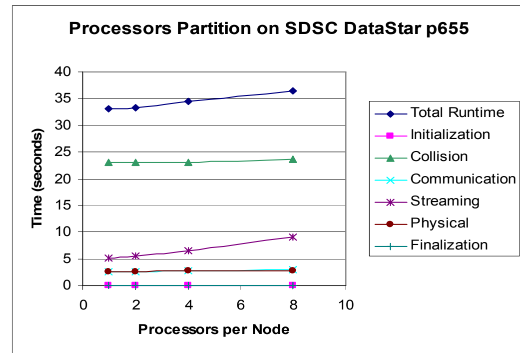

Figure 3. Execution times of LBM with the problem size of 128x128x128 for different PPN on 32 processors on P655

Table 10. Execution times (seconds) of LBM for different PPN on 32 processors

System Name	Problem size	32x1	16x2	8x4	4x8	2x16	1x32
Seaborg	128x128x128	94.21	94.42	95.29	98.72	109.12	NA
	512x512x512	6247.87	6307.41	6418.77	6722.31	7578.84	NA
P655	128x128x128	33.09	33.38	34.57	36.41	NA	NA
	512x512x512	2154.94	2224.76	2361.67	2601.66	NA	NA
P690	128x128x128	NA	NA	NA	29.15	30.35	36.01
	512x512x512	NA	NA	NA	2198.90	2248.62	2590.18
Bassi	128x128x128	21.43	21.92	21.14	21.78	NA	NA
	512x512x512	1379.63	1382.30	1380.32	1384.84	NA	NA
BlueGene/L	128x128x128	33.40	33.42	NA	NA	NA	NA
	512x512x512	2252.99	2253.27	NA	NA	NA	NA

The LBM code is divided into six kernels as given below: *Initialization, Collision, Communication, Streaming, Physical,* and *Finalization.* The kernels *Collision, Communication, Streaming*, and *Physical* are executed in a loop with 200 iterations. Kernels *Initialization* and *Finalization* are executed once. Figure 3 provides the execution times for the problem size of 128x128x128 for different PPN on 32 processors on P655; similar performance trend occurs on P690 and Seaborg. Figures 3 indicates that the kernel *collision* dominates most of the application execution time, but the performance trend for the kernel *Streaming* is similar to that for the total runtime on P655. The similar

performance trend also occurs for the problem size of 512x512x512.

Table 11 indicates the results from using processor binding such that only one core per chip is used thereby minimizing resource sharing. The results indicate a 4.71% decrease in execution time for the 16x2 configuration and a 6.75% reduction in execution time for the 8x4 configuration. This indicates that processor binding can aid task schedulers in maximizing the application performance in dedicated usage of CMP nodes.

Table 11. Performance comparisons of LBM with 512x512x512 on P655 using processor binding

#MxN	2PPC	1PPC	% difference
16x2	2224.76	2124.66	4.71
8x4	2361.67	2212.28	6.75

Table 12 provides the following details about the performance characteristics for LBM with problem size of 512x512x512: the L1 hit rate, L2 bandwidth (per processor), and percentage accesses from L2. Because BlueGene/L does not support *hpmcount*, we did not collect its hardware level performance. With increasing PPN, the L1 hit rate varies very little across all systems because of equal workload per processor. On P655, the L2 bandwidth and percentage accesses from L2 decrease with an increase of PPN; this is especially the case for the L2 bandwidth. For example, the L2 bandwidth for the 32x1 scheme is more than 11% larger than that for the 4x8 scheme. On Bassi, there is very small difference in percentage accesses from L2 across different PPN due to 20GB large page size, which uses hardware prefetch mechanisms to eliminate costly TLB misses. Further, Bassi has the highest memory bandwidth and MPI bandwidth, which result in very little change in communication rate for the different configurations. Hence, the execution time is flat for the different PPN on Bassi. It is interesting that Bassi has the lowest L1 hit rate and correspondingly, the highest percentage accesses from L2.

Table 12. Memory performance for LBM with the size of 512x512x512

System	Metrics	32x1	16x2	8x4	4x8	2x16
P655	**L1 hit rate**	98.13%	98.15%	97.99%	97.69%	NA
	L2 bandwidth (MB/s)	1127.01	1098.07	1082.61	1015.14	
	% accesses from L2	0.618%	0.602%	0.593%	0.541%	
Bassi	**L1 hit rate**	94.93%	92.82%	95.33%	95.85%	NA
	L2 bandwidth (MB/s)	3.035	4.788	2.523	2.145	
	% accesses from L2	5.069%	7.177%	4.669%	4.147%	
Seaborg	**L1 hit rate**	99.14%	99.00%	98.93%	98.88%	99.14%
	L2 bandwidth (MB/s)	0.069	0.066	0.064	0.060	0.050
	% accesses from L2	0.332%	0.380%	0.409%	0.420%	0.340%

Table 13. Execution times (seconds) of LBM between the original and the optimized on P655

Processors (MxN)	original	optimized	% improvement	Optimal block size
32 (4x8)	2601.66	2191.28	15.77%	4x4
64 (8x8)	1306.55	1110.71	14.99%	4x4
128 (16x8)	653.31	561.56	14.04%	4x4
256 (32x8)	319.71	287.71	10.01%	4x4
512 (64x8)	163.44	157.53	3.62%	16x16
1024 (128x8)	83.9	79.37	5.40%	16x16
2048 (256x8)	48.56	45.3	6.71%	16x16

Table 14. Execution times (seconds) of LBM between the original and the optimized on BlueGene/L

Processors (MxN)	original	optimized	% improvement	Optimal block size
32 (16x2)	2253.27	1954.95	13.24%	16x16
64 (32x2)	1280.72	1092.11	14.73%	16x16
128 (64x2)	699.71	606.83	13.27%	16x16
256 (128x2)	356.09	311.35	12.56%	16x16
512 (256x2)	153.95	150.73	2.09%	2x2

5.2 Application Performance Optimization

In this section, we use the LBM application with the problem size of 512x512x512 to utilize the information learned from the previous section to determine how to optimize the application. It is noted that optimizing the dominated kernel *collision* does not eliminate the performance impact using different number of processors per node because of its flat performance trend across different PPN. Hence, we focus on optimizing the kernel *Streaming*, which entails moving particles in motion to new locations along their respective 19 velocities [13]. This kernel requires a significant number of memory copy operations, which causes memory congestion when large numbers of

processors per node are used. In particular, the kernel *Streaming* consists of five triple-nested loops. We focus on using cache blocking to optimize the outer two loops of each triple-nested loop. For the sake of simplicity, we present results for the problem size of 512x512x512.

Table 13 provides the performance comparison between the original code and the optimized for the problem size of 512x512x512 on P655 using all processors per node. The optimized code with the block size of 4x4 achieved the best performance on 32, 64, 128 and 256 processors, but the optimized code with the block size of 16x16 obtained the best performance on 512, 1024, and 2048 processors. For only optimizing the kernel *Streaming* in the original application code, the percentage of performance improvement is up to 15.77%. Compare to the results for different PPN on P655 in Table 10, where there is 20.73% time difference between the scheme 32x1 and the 4x8, our optimization reduces execution time by 15.77%. This is a big improvement. Our results also show that the same optimization resulted in much smaller performance improvement for 32x1 than that for 4x8 on P655. Note that the performance improvement percentage trend decreases with increasing the number of total processors for the optimal block size of 4x4 because of the decrease of workload per processor and the increase of the communication percentage with increasing the number of processors. For the problem size of 512x512x512, the LBM application can only be executed on 32 processors or more because of the large sufficient memory requirement.

Table 14 provides the performance comparison between the original code and the optimized for the problem size of 512x512x512 on RENCI BlueGene/L using all processors per node. The optimized code with the block size of 16x16 achieved the best performance on 32, 64, 128 and 256 processors, but the optimized code with the block size of 2x2 obtained the best performance on 512 processors. For only optimizing the kernel *Streaming* in the original application code, the percentage of performance improvement is up to 14.73%. Compare to the results for different PPN on BlueGene/L in Table 10, this is a big improvement. Notice that, for the same problem size and same total number of processors, the cache blocking optimization resulted in much larger performance improvement of the LBM application when using all the processors per node versus one processor per node.

6. Conclusions

This paper used two scientific applications, GTC and LBM, to analyze the performance impact of the sharing of node resources on five supercomputers, P655, P690, BlueGene/L, Seaborg and Bassi, especially focusing on CMP clusters, P655, P690 and BlueGene/L. The experimental results indicated that there can be a significant difference in execution time when using different numbers of processors per node and using different numbers of cores per chip, chips per MCM, and different numbers of MCMs on these CMP clusters. Memory bandwidth contention, especially L2 cache, is the primary source of performance degradation. Using conventional loop optimization techniques which aids in taking advantage of advanced memory hierarchy and a hybrid communication optimization method, the application performance can be improved by up to 18.97% for GTC and up to 15.77% for LBM on up to 2048 processors when utilizing all processors per node. Hence, understanding the application and system characteristics can lead to optimization that aids in the efficient use of all processors per node on CMP clusters.

Acknowledgements

The authors would like to thank Stephane Ethier and Shirley Moore for providing the GTC code and datasets, and Dazhi Yu and Jacques Richard for providing the LBM code.

References

[1] S. Behling, R. Bell, et al., *The POWER4 Processor Introduction and Tuning Guide*, IBM Redbooks, 2001.

[2] S. Ethier, First Experience on BlueGene/L, *BlueGene Applications Workshop*, April, 2005.

[3] B. Gibbs, B. Atyam, et al., *Advanced POWER Virtualization on IBM@server p5 Servers: Architecture and Performance Considerations*, IBM Redbooks, 2005.

[4] hpmcount, http://www.nersc.gov/nusers/resources/software/ibm /hpmcount/.

[5] Intel MPI Benchmarks (Version 2.3), http://www.intel.com/cd/software/products/asmona/eng/cluster/mpi/.

[6] NERSC Seaborg and Bassi, http://www.nersc.gov/.

[7] F. Petrini, D. Kerbyson, and S. Pakin, The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q, *SC03*.

[8]J. Phillips, G. Zheng, S. Kumar, and L. Kale, NAMD: Biomolecular Simulation on Thousands of Processors, *SC02*.

[9] RENCI BlueGene/L, http://www.renci.org/.

[10] DataStar, http://www.sdsc.edu/user_services/datastar/.

[11] R. Thakur, R. Rabenseifner, and W. Gropp, Optimization of Collective Communication Operations in MPICH, *International Journal of High Performance Computing Applications*, Vol. 19, No. 1, 2005.

[12] V. Taylor, X. Wu, and R. Stevens, Prophesy: An Infrastructure for Performance Analysis and Modeling System of Parallel and Grid Applications, *ACM SIGMETRICS Performance Evaluation Review*, V. 30, 2003.

[13] X. Wu, V. Taylor, S. Garrick, D. Yu, and J. Richard, Performance Analysis, Modeling and Prediction of a Parallel Multiblock Lattice Boltzmann Application Using Prophesy System, *IEEE Conference on Cluster Computing*, 2006.

[14] X. Wu and V. Taylor, Processor Partitioning: An Experimental Performance Analysis of Parallel Applications on SMP Cluster Systems, *the 19th International Conference on Parallel and Distributed Computing and Systems*, 2007.