Analyzing the Effect of Different Programming Models Upon Performance and Memory Usage on Cray XT5 Platforms

Hongzhang Shan
Future Technology Group, Computational Research Division,
Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Haoqiang Jin
NAS Division, NASA Arms Research Center, Moffett Field, CA 94035-1000

Karl Fuerlinger
University of California at Berkeley, EECS Department, Computer Science Division Berkeley, CA 94720

Alice Koniges, Nicholas J. Wright
NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Abstract—**Harnessing the power of multicore platforms is challenging due to the additional levels of parallelism present. In this paper, we examine the effect of the choice of programming model upon performance and overall memory usage on the Cray XT5. We use detailed time breakdowns to measure the contributions to the total runtime from computation, communication, and OpenMP regions of the applications, gaining insights into the reasons behind any performance differences observed. We also examine the performance differences between two different Cray XT5 machines, which have quad-core and hex-core processors.**

I. Introduction

Moore's Law continues unabated; the number of transistors on a chip still doubles every eighteen months. However, because of power constraints, clock speeds remain approximately constant and the extra transistors are being used to add cores. This will lead to future architectures with multicore/manycore designs with a greater number of simpler, smaller cores as compared to today. Also, as the growth in memory capacity is not keeping track with the growth in the number of cores, memory per core will also decrease on future architectures. Thus a critical factor affecting parallel programming models in the future is the amount of memory that they use.

Currently, the most popular programming model is to use MPI with as many MPI tasks as there are cores. In MPI programs, each MPI process has its private address space and processes move data from one address space to another via sending and receiving messages. The communication between these processes is carried out through explicit message passing. Therefore, extra data copies and/or duplication are usually needed. This currently popular programming model of MPI everywhere is not likely to be a viable model on these newer architectures simply because of the reduced amount of memory per core that will be available. It is therefore important to investigate other available programming models to understand whether they can replace or be combined with MPI in order to avoid these issues.

Another parallel programming model commonly used is shared memory, using threads. OpenMP is the most commonly used programming model for shared memory parallelism in the High Performance Computing (HPC) community. Generally, OpenMP provides convenient features for loop-level parallelism as well as some advanced dynamic approaches to parallelism such as tasking. In OpenMP programs, all data is shared by all OpenMP threads and can be directly accessed. Unlike MPI programs, no extra copying is needed for data communication and exchange. In general, shared memory paradigms such as OpenMP could potentially save a large amount of memory and enable relatively larger data sets to be run. This is particularly true if one considers a hybrid programming model, that is one that uses OpenMP within a node and MPI between nodes.

Another approach to parallelism is the PGAS (Partitioned Global Address Space) languages. These attempt to combine the convenience of the global view of data with an awareness of data locality. One of these PGAS languages, UPC (Unified Parallel C) is an extension to C with both shared and local addresses.

In this paper we compare the performance of the NAS parallel benchmarks, written in MPI, OpenMP and UPC on the Cray XT5 platform and examine the memory usage of the different programming models. We also examine the performance differences between two different Cray XT5 machines, one with quad-core processors and one with hex-core.

This paper is structured as follows: Section II describes the two Cray XT5 machines we use in more detail, Section III describes our data collection techniques, Section IV describes the results of our memory usage measurements and Section V contains the performance results comparing the different programming models. Section VI contains the performance comparison between Jaguar, the hex-core, and Hopper, the quad-core, XT5 machines. Finally, Section VII contains the related work and Section VIII summarizes.

II. Platforms for Experiments

In this work we compare the performance of two Cray XT5 machines, Jaguar and Hopper. Hopper is located at NERSC and Jaguar is located at Oak Ridge National Laboratory. The machines are very similar, both are made up of dual socket nodes connected with Seastar 2+ Cray interconnect and with 16 GB DDR2 800 MHz memory per node. The principal difference is the processors. Hopper contains 2.4 GHz quad-core AMD 'Shanghai' Opteron processors whereas Jaguar contains hex-core 2.6 GHz 'Istanbul' processors. In principle therefore there is 1.6x the computational power available per Jaguar node compared to a Hopper node. However the memory and network subsystems on each node are the same, this is illustrated by the stream [?] number which is 1.5X slower per core on Jaguar. Therefore the relative performance of Jaguar to Hopper of a scientific application will depend upon the demands it places upon the hardware; a purely memory bound code will be slower on Jaguar on a per core basis, whereas a compute bound code will be faster.

We note here that the NERSC XT5 machine used here is actually phase 1 of Hopper; a much larger machine, phase 2, will be installed later this year.

III. Data Collection Techniques

To gain an idea of how the different programming languages compare, we use versions of the NAS Parallel Benchmarks (NPB) as a benchmark suite. In the examples used in this paper, the MPI and OpenMP versions come from the standard NAS distribution [?], and the UPC codes come from a distribution developed by George Washington University (GWU), the Berkeley UPC group, and NAS [?], [?].

As well as recording the runtimes of our performance experiments we also instrument them using the Integrated Performance Monitoring (IPM) framework [?], [?]. IPM provides a low overhead mechanism for obtaining information about the MPI performance characteristics of an application. It uses the Profiling interface of MPI (PMPI) to obtain information about the time taken and type of MPI calls, the size of the messages sent and the message destination. Recently IPM was augmented to obtain OpenMP profiling information also. By using compiler instrumentation to insert tracepoints at the beginning and end of every OpenMP region within the code we are able to measure the time taken in OpenMP by the application. This is a useful indicator to understand the scaling behavior of OpenMP based codes. Those that spend a significant amount of their runtimes in serial regions, i.e. those involving neither OpenMP nor MPI are not going to be able to scale efficiently to large numbers of threads. We also use IPM to record the memory high-water mark for each of the applications. This is simply the number reported by the Linux kernel.

The compiler used for MPI, OpenMP, and hybrid MPI+OpenMP is the default PGI compiler installed on Cray XT5. For UPC, the berkeley UPC compiler and runtime system are used [?].

IV. Memory Usage of Different Programming Models

Fig. 1 shows the difference in the amount of memory usage of NPB3.3 programs when MPI, OpenMP, and UPC versions of the code are used. The data are collected for class C data sets when four cores are used. All the values are relative to the amount of memory used by the MPI version. Clearly the most memory efficient version is the OpenMP one. The memory savings are around 50% for BT, EP, IS, and SP, 20% for CG, FT, and LU. Only for MG, the amount of memory used by the OpenMP version is close to MPI, but still less by 4%. The smaller difference between OpenMP and MPI for MG is mainly because the communication buffer needed for MPI in this code is quite small compared with the grid data. For smaller data sets, the percentage difference will become larger. The EP benchmark is an Embarrassingly Parallel program which requires almost no effort to communicate data between different tasks and therefore no extra data copy or duplication are needed for MPI. Thus the additional memory usage in EP's MPI version can be attributed to the MPI runtime consumption.

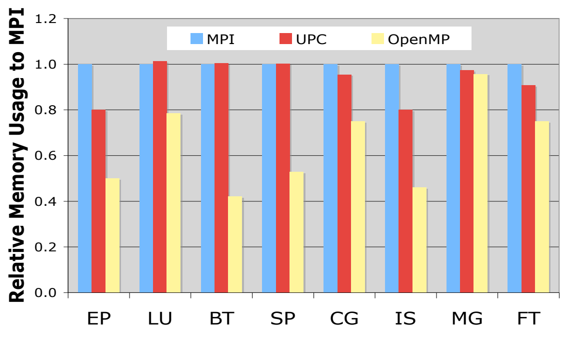

Fig. 1. The relative memory usage of NPB3.3 applications for MPI, OpenMP, and UPC for four-core runs. The ratios are relatively to MPI usage.

Fig. 2 shows that the actual amount of memory usage for each of the benchmarks. For EP it is quite small. For FT, BT, SP, and IS, MPI uses substantially more memory than OpenMP.

Specifically, for FT, OpenMP consumes around 25% less memory than MPI (as shown in Fig. 2). FT performs a Fourier Transform which contains a transpose operation. This large memory difference is mainly caused by the differences between the transpose implementation between MPI and OpenMP. For the transpose, which is implemented by all-to-all communication in MPI, an extra array is needed for MPI to hold the communication data while in OpenMP, the data can be directly accessed by all threads and thus extra array is not necessary. For CLASS C, the array size is 512*512*512*sizeof(double complex)=2GB. The main memory usage difference is caused by this extra data array of size 2GB.

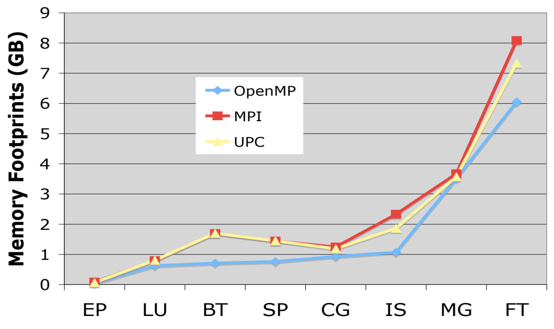

Fig. 2. The memory footprints of NPB3.3 applications for MPI, OpenMP, and UPC for four-core runs.

Another difference between MPI and OpenMP is that usually the amount of memory usage for OpenMP is constant regardless of the number of cores used while for MPI, more memory is needed as the number of tasks increases. This may be due to MPI runtime system requirements, communication buffers in the applications, or the replication of stored data to avoid communication. This may not be true for all OpenMP applications however. In some applications, the OpenMP threads may need to dynamically allocate more space, to store private variables or to store data on a per thread basis for later aggregation to avoid locks.

The amounts of memory used by the UPC versions is very close to the MPI versions, only slightly less. Like OpenMP programs, UPC could provide a shared address space for the shared data. Therefore, potentially it could save the same amount of memory as OpenMP. However, in this study, the UPC programs used are converted from the corresponding MPI programs. For performance reasons, explicit data partition and one-sided communication via upc_memput/upc_memget are used, leading much higher memory usage than OpenMP.

A. *MPI+OpenMP Hybrid Model*

We now consider hybrid programming models that use OpenMP within shared memory nodes and MPI between nodes. One question is whether or not the memory usage advantage of OpenMP is retained in hybrid programming models. The NPBs include "multi-zone" (MZ) versions that use a standard hybrid OpenMP and MPI programming model for the NPB3.3-MZ release.

Fig. 3 displays the relative memory footprints of SP-MZ and BT-MZ. The results are collected for 256 cores on Hopper for different combinations of MPI tasks and OpenMP threads.

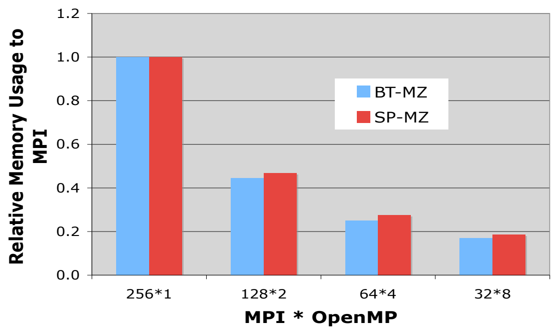

Fig. 3. The relative memory usage of BT-MZ and SP-MZ for MPI+OpenMP hybrid programming models for different combinations of MPI tasks and OpenMP threads.

The base case is using 256 MPI processes and 1 OpenMP thread for each MPI process, i.e., setting the OMP_NUM_THREADS to 1. Then, we increase the OMP_NUM_THREADS to 2, 4, and 8 and reduce the number of MPI processes correspondingly. All the memory usage measurements are relative to the base case. The results in Fig. 3 indicate that using more OpenMP threads could significantly reduce the amount of memory needed. When the number of OpenMP threads reaches 8, the amount of memory needed drops to 20% of the base case. Thus using OpenMP saves a significant amount of memory when hybrid programming models, showing a great promise for its future.

V. Performance Effects of Different Programming Models

In this section, we investigate the performance differences between programming models on the same platform, Hopper. Furthermore, we explain potential reasons for these performance differences, such as whether they are due to the semantics of the programming models or due to the implementation or coding styles. The performance on one node for MPI, OpenMP, and UPC are examined first. Then, the inter-node performance is compared to include the effects of the interconnect.

A. *Performance Using One Node*

Fig. 4 shows the performance ratio of OpenMP and UPC to MPI for 8-core runs on Hopper which has eight core nodes. (Due to the algorithmic limitations within BT and SP, they can

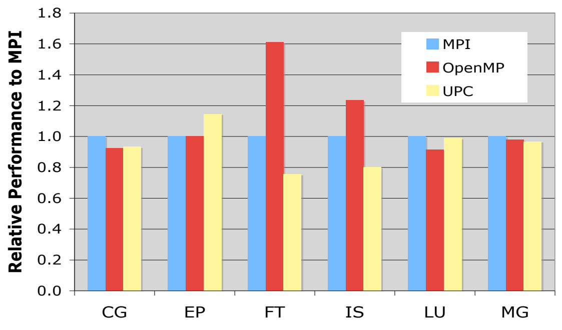

Fig. 4. The Relative Performance to MPI on a 8-core node.

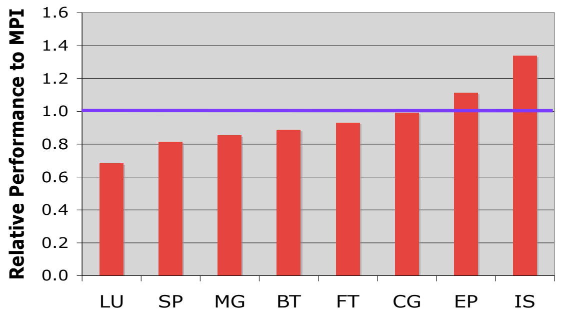

Fig. 5. The Relative Performance of UPC to MPI on Hopper for 64 cores.

only be run with square number of tasks and therefore cannot be run with 8 tasks.)

For CG, LU, EP, and MG, the three programming models deliver very similar performance. However, FT and IS show substantial performance difference across these three models. OpenMP delivers the best results, almost 60% faster than MPI, UPC delivers the lowest performance, 20% slower than MPI. Further examination shows that for FT, the explicit transpose in the MPI program causes extra data copy and movement, which is not necessary in the OpenMP program, slowing down its execution. The UPC program follows similar programming pattern of MPI. It also includes an explicit transpose phase and this phase takes more time than MPI, leading its performance to be the worst of these three. Theoretically, due to the shared memory concept supported by UPC, the UPC version of FT could be programmed in a way similar to OpenMP instead of MPI. However, it is not currently implemented this way.

The IS benchmark suffers a problem similar to that of FT. In its MPI implementation, there exists an explicit data exchange phase, which does not exist in the OpenMP version. This extra data copy and movement phase causes the MPI and UPC performance to degrade.

The results indicate that the performance differences inside a node between different programming models are mainly caused by the programming differences needed for different programming models. For most of the NPB these effects are not significant the exceptions being FT and IS. In these cases the shared memory model of OpenMP allows for less overall communication and therefore higher performance.

B. Performance Using Eight Nodes

Currently, OpenMP programs cannot not be run using more than one node on Cray platforms. Therefore, in this subsection, we will compare only the performance of MPI and UPC.

Fig. 5 shows the relative performance of UPC to MPI for 64 cores, which is eight nodes of Hopper. UPC performs worse than MPI for LU, SP, MG, BT, and FT, almost the same for CG, and better for EP and IS. To understand the reasons why we analyze the performance of LU, CG and IS in more detail. To do this we divide the total runtime into two parts: the communication time and local computation time. For the MPI versions these measurements are made with IPM, for the the UPC versions they are made by inserting timers into the code. The ratios of these components of UPC runtime to MPI are shown in Fig. 6 and the actual times are displayed in Table I. For LU, the time the UPC version spends on local computation is slighter higher than the corresponding MPI time. However, the UPC communication time is much higher than the MPI communication time. Further examination shows that in MPI, the communication is carried out by send/recv pairs while in UPC, it is implemented by one-sided put/get method. However, due to the lack of a point-to-point synchronization primitive in the current UPC language, the synchronization for pipelining operations in LU is done by testing the value of a shared variable. It may be not as efficient as the implicit MPI synchronization through send/recv pairs. For CG, both the communication time and local computation times are similar for these two programming models. For IS, the communication time dominates the performance. Though the UPC computation time is 70% higher, the saving in the communication time outweighs the increase of the computation time, leading to a better overall runtime for UPC. The significantly better communication time in the UPC version is due to the one-sided put/get messaging it uses which is more efficient than the MPI_Alltoallv function in the MPI program.

TABLE I
THE COMMUNICATION TIME AND COMPUTATION TIME OF UPC AND MPI FOR 64-CORE RUNS ON HOPPER

	LU		CG		IS	
	MPI	UPC	MPI	UPC	MPI	UPC
Comm	5.06	14.25	6.08	5.99	1.51	0.83
Comp	24.56	29.05	8.98	9.22	0.31	0.53

Overall the MPI and UPC results show that the ease-programming one-sided messages in UPC usually performs more efficiently than the send/recv pairs in MPI. However, it does not guarantee a better overall performance.

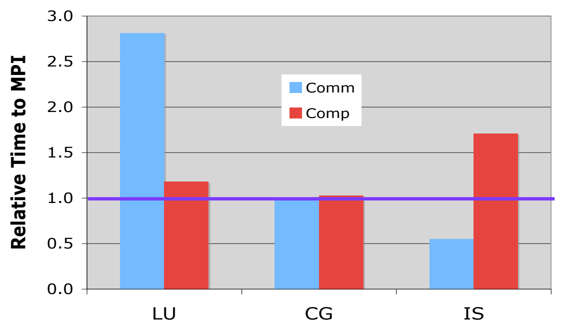

Fig. 6. The Relative Component Time Breakdown of UPC to MPI on Hopper (the ratios are used).

C. *Hybrid MPI+OpenMP*

We now turn to the performance of the Multi-Zone hybrid MPI/OpenMP NPB's. Using IPM we are able to partition the runtime into time spent in MPI or OpenMP or neither MPI or OpenMP (called Serial).

For BT-MZ, its time breakdown is shown in Fig. 7 and Table II. When 16 cores are used almost all of the runtime is in OpenMP regions. With the increase of the number of cores, OpenMP time reduces very fast while the MPI time increases. For 256 cores, when 256 MPI tasks are used, the MPI time increases sharply. This is due to load imbalance. For the CLASS C data set, there are total 256 zones, one per MPI tasks, with substantially different sizes. When fewer MPI processes are used, the load imbalance is improved due to the bin-packing load assignment algorithm, i.e., using fewer MPI processes and more OpenMP threads. This is a nice example of one of the oft-stated benefits of hybrid programming: the ability to mitigate load-balance issues by requiring less overall domain decomposition.

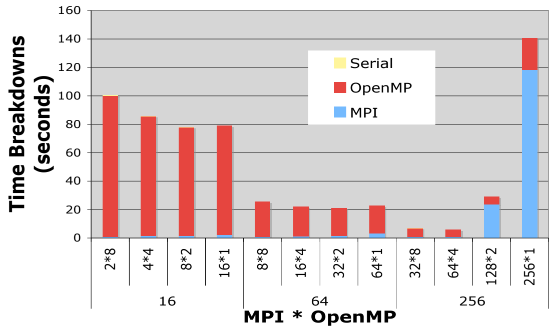

Fig. 7. The BT-MZ Time Breakdown on Hopper.

The SP-MZ performance scales very well with the number

TABLE II
THE MPI, OPENMP,AND SERIAL TIMES (SECONDS) FOR BT-MZ

16 Cores (MPI * OpenMP)				
	2*8	4*4	8*2	16*1
MPI	0.76	1.45	1.46	2.08
OpenMP	99.23	84.12	76.49	76.78
Serial	0.28	0.12	0.06	0.03

64 Cores (MPI * OpenMP)				
	8*8	16*4	32*2	64*1
MPI	0.83	0.95	1.53	3.23
OpenMP	24.76	21.09	19.53	19.52
Serial	0.07	0.03	0.02	0.01

256 Cores (MPI * OpenMP)				
	32*8	64*4	128*2	256*1
MPI	0.65	0.69	23.72	118.39
OpenMP	6.30	5.21	5.45	22.32
Serial	0.03	0.01	0.01	0.01

of cores as shown in Fig. 8. The time breakdown shows that most of the time is spent on the OpenMP regions. The percentage of time spent in MPI functions is quite small though it increases with the number of cores used (see Table III). Thus the performance is dominated by the OpenMP performance. There are also performance differences using different numbers of OpenMP threads. The best performance is obtained when the number of OpenMP threads per MPI process is 2.

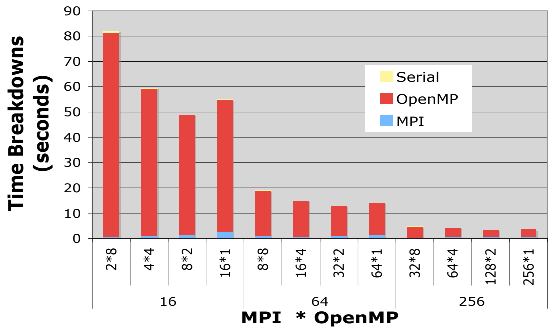

Fig. 8. The SP-MZ Time Breakdown on Hopper.

For LU-MZ, the best performance is obtained when maximum number of OpenMP threads are used, as shown in Fig. 9. This shows that most of the runtime is spent in OpenMP and the best performance is obtained when the number of OpenMP threads reaches 8 (see Table IV). The performance of LU-MZ favors using more OpenMP threads. This is probably due to the following reasons. First, in each OpenMP region, the workload for each OpenMP thread is high, enough to amortize the cost to activate and deactivate the OpenMP thread. Secondly, using more OpenMP threads also reduces the corresponding number of MPI processes. A side effect is that the total number of messages is reduced and the message sizes are increased, leading to better communication performance and improving

TABLE III
THE MPI, OPENMP,AND SERIAL TIMES (SECONDS) FOR SP-MZ

16 Cores (MPI * OpenMP)				
	2*8	4*4	8*2	16*1
MPI	0.39	0.97	1.52	2.55
OpenMP	81.10	58.28	47.30	52.36
Serial	0.64	0.25	0.14	0.08
64 Cores (MPI * OpenMP)				
	8*8	16*4	32*2	64*1
MPI	1.12	0.51	0.95	1.42
OpenMP	17.71	14.23	11.8	12.51
Serial	0.16	0.07	0.04	0.03
256 Cores (MPI * OpenMP)				
	32*8	64*4	128*2	256*1
MPI	0.29	0.59	0.49	0.44
OpenMP	4.38	3.46	2.92	3.10
Serial	0.05	0.02	0.02	0.00

TABLE IV
THE MPI, OPENMP,AND SERIAL TIMES (SECONDS) FOR LU-MZ

8 Cores (MPI * OpenMP)				
	1*8	2*4	4*2	8*1
MPI	0.00	0.60	1.23	2.08
OpenMP	187.01	210.15	328.23	339.60
Serial	0.06	0.03	0.02	0.01
16 Cores (MPI * OpenMP)				
	2*8	4*4	8*2	16*1
MPI	0.33	1.98	1.01	1.46
OpenMP	93.64	105.62	164.72	169.9
Serial	0.04	0.01	0.00	0.01
64 Cores (MPI * OpenMP)				
	8*8	16*4	32*2	64*1
MPI	0.31	0.37	X	X
OpenMP	23.4	26.12	X	X
Serial	0.01	0.01	X	X

the performance further. Finally, each MPI process is assigned several zones. The OpenMP threads spawned by the same MPI process will work on these zones together, one at a time. Using more OpenMP threads will increase the cache size for the same amount of data and improve the performance (assuming one OpenMP thread assigned to one core).

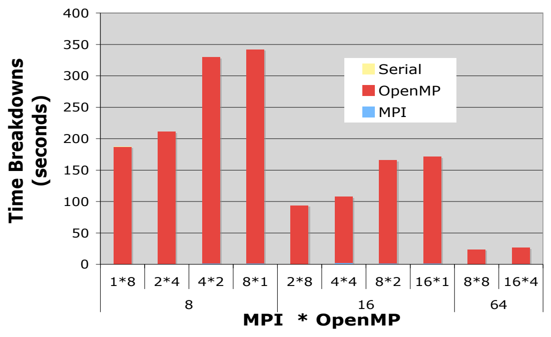

Fig. 9. The LU-MZ Time Breakdown on Hopper.

For hybrid MPI+OpenMP applications, using more OpenMP threads will potentially improve the cache usage and communication patterns while it may also increase the synchronization cost among the OpenMP threads and the activation/deactivation overhead. The tradeoff will be closely related with the workload size in the OpenMP regions. In LU-MZ, there are total 16 zones while in BT-MZ and SP-MZ, there are total 256 zones. The zone size in LU-MZ is much larger than the zone size in BT-MZ and SP-MZ. Therefore, LU-MZ requires relatively larger OpenMP threads to deliver the best performance. However, the exact number of OpenMP threads to deliver the best performance may vary on different architectures and platforms.

VI. Performance Comparison of Hopper and Jaguar

On Hopper, each node has two quad-core processors while on Jaguar each has two hex-core processors. In this section, we will examine the performance effects of this difference.

A. Single Node Performance

On Jaguar, each node has 12 cores, and the MPI and UPC version of NPB3.3 cannot not be run with 12 tasks, which require the number of tasks to be power of 2 or a square number. Thus we focus our attention on the OpenMP programs only.

If eight cores are used on Jaguar, the expected performance difference between Jaguar and Hopper will at best be (2.6/2.4=) 1.08 due to the different CPU frequencies or 1.0 due to same memory bandwidth. This is exactly the case as shown in Fig. 10. Some of the applications are very slightly faster on Jaguar, EP, CG, IS and LU, due to they being basically cache resident and therefore sensitive to the clock speed difference. When all 12 cores on a Jaguar node are used, the expected performance ratio is, at most, 1.625 (6*2.6/4*2.4). Only EP reaches this value. For CG and IS, the values are around 1.4. This is simply a reflection of the cache speed dependency of these applications as noted before. The worst ratio is for MG and SP which show approximately equal performance on 12 cores of Jaguar and 8 core of Hopper. This indicates that they are memory bound, and that for these kinds of applications the extra two cores on Jaguar are not providing any performance benefit.

B. Performance Using 64, 256 and 1024 Cores

On both machines, the codes were run on the same number of cores, 64, 256 and 1024. On Jaguar, each node has 12 cores while on Hopper 8. For these core counts that are not exactly divisible by 12 some of the cores on one node of Jaguar were left idle.

In this case it is hard to determine the expected performance ratios. As well as the factors due to clock speed and memory

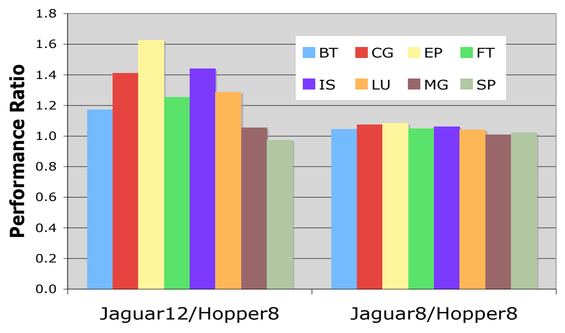

Fig. 10. The Performance Ratio of Jaguar vs. Hopper For OpenMP Model on a Node.

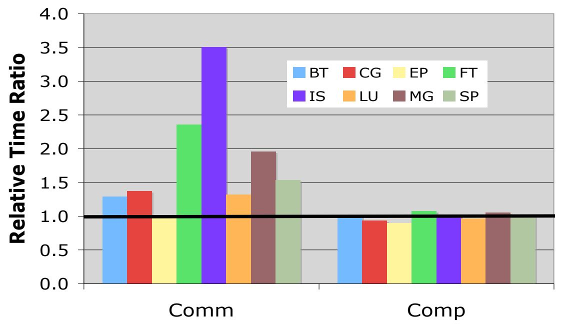

Fig. 12. The Normalized Time Breakdown of Jaguar vs. Hopper for MPI Programs using 1024 cores.

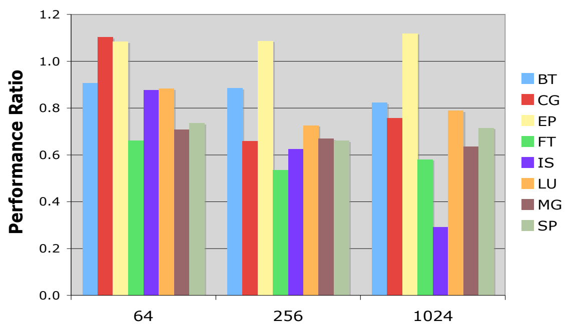

Fig. 11. The Performance Ratio of Jaguar vs. Hopper for MPI Programs.

contention outlined in the previous section there is also contention for network resources because on Jaguar there are 1.5× as many cores sharing the same network resource.

The normalized performance of Jaguar vs. Hopper at each of the three core counts is shown in Fig. 11 and the corresponding normalized runtime breakdown is shown in Fig. 12 as measured using IPM. The EP benchmark consistently performs better on Jaguar due to its higher processor frequency as discussed before. This can also be deduced from its lower computation time in Fig. 12. For all the other applications, the computation time on these two platforms are very close, which is to be expected from the single node experiments. The performance differences are mainly caused by different communication performance. For IS, the communication time on Jaguar is 3.5 times higher than Hopper, leading to the worst normalized performance on Jaguar. Using more cores per node without increasing the network bandwidth causes more network contention, degrading the overall performance substantially.

We now consider the multizone benchmarks. For BT-MZ, the performance ratio is very close to the expected value, one. For SP-MZ, Jaguar always performs worse than Hopper. (For LU-MZ, since there are total 16 zones for Class C data set so we can not run cases which uses more than 16 MPI processes.) The difference in performance between Jaguar and Hopper decreases with increasing number of OpenMP threads which indicates that the problem of network and memory contention is increased in the hex-core configuration.

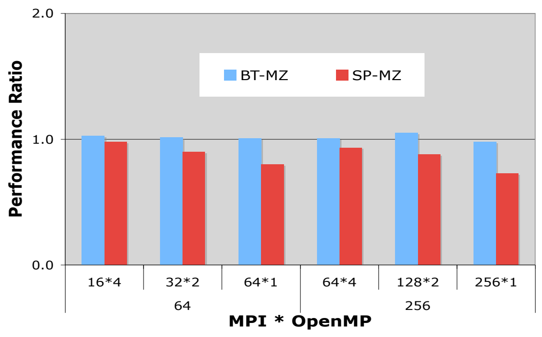

Fig. 13. The Performance Ratio of Jaguar vs. Hopper For NPB3.3-MZ.

VII. Related Work

A lot of literature has been published comparing different programming models. Among them, the most related to our work is [?], where the performance of MPI, OpenMP, and UPC were evaluated on a machine with 142 HP Integrity rx7640 nodes interconnected via InfiniBand. The authors claim that MPI is the best choice to use on multicore platforms, as it takes the highest advantage of data locality . Our work differs from several perspectives. First, we use more applications in our evaluation and the UPC codes we used are better written and tested. Secondly, we find that the best performance is not always produced by the MPI version of the code, both OpenMP (inside a node) and UPC can outperform MPI for some applications. Thirdly, we quantitatively studied the performance effects of increasing from 8 cores to 12 cores in a node.

We also examined the performance of NPB3.3-MZ which is developed in MPI+OpenMP hybrid programming models. We found that using more OpenMP threads always delivers better performance than using one OpenMP thread per MPI process. Similar work to evaluate the performance effects of hybrid MPI+OpenMP models on Cray XT5 can be found in [**?**]. However, in this paper, we provide detailed time breakdowns to help to understand how the performance changes with varying number of MPI and OpenMP tasks instead of only the absolute performance. We also provides detailed time breakdowns to compare the MPI performance and UPC performance. This differs us from several other researches which only compare the absolute performance of MPI and UPC [**?**], [**?**].

Furthermore, we quantitatively measured the amount of memory needed by different programming models, including MPI, OpenMP, UPC, and hybrid MPI+OpenMP. To the best of our knowledge this is the first time that memory usage of these different programming models has been quantitively analyzed and compared.

VIII. Conclusions

In this paper we have examined the performance of different programming models OpenMP, MPI and UPC on the Cray XT5 machine, Hopper at NERSC. As well as simply measuring the runtime by using IPM and inserting explicit timers into the code we were able measure the contributions to the runtime from computation, communication and OpenMP regions of the applications. Therefore we were able to gain insight into the reasons behind any performance differences observed. Our results show that in most cases the performance of each of the different programming methods are very close for the NAS Parallel Benchmarks. In the cases that show the most performance difference it is always the OpenMP case that is the fastest. Our performance analysis shows that this is always due to the reduced communication costs in the shared memory model.

We also compared the performance of MPI and UPC on 64 cores of Hopper. The results showed that the performance of the two methods is, on average, quite similar, with UPC being slightly slower overall. This is mainly because the UPC compiler and runtime systems we used are still under improvement and have not been fully tuned to work on the NUMA architectures, like Cray-XT5.

We also examined the memory usage of each of the different programming models. In general OpenMP has much reduced memory requirements. This is especially true for the FT and IS benchmarks, because these applications need extra arrays to hold communication data. This advantage is also been reflected in MPI+OpenMP hybrid results. Furthermore, the hybrid results indicate that using more than one OpenMP thread always produces better results than using only one OpenMP thread per MPI process, showing great promise for the future of the hybrid programming models.

We also looked at the performance differences caused by the different node size of Hopper and Jaguar (quad and hex core respectively). The results tell us that putting more cores on a node will potentially cause more memory contention. This may degrade the performance of applications which are memory bandwidth bound, often obviating the potential advantage of the additional two cores per socket. Another disadvantage of the hex-core configuration is the increased contention for interconnect resources. This is especially apparent from the results for the IS benchmark which runs $3\times$ slower on Jaguar than Hopper on 1024 cores. In this case it maybe better to run using less MPI tasks than cores and use hybrid programming models as this will lead to less contention for shared resources, as in the case of the SP-MZ results.

In future work we plan to look at another hybrid programming model, UPC + MPI, as well as extend our analysis to full scale scientific applications to understand in greater depth the advantages and disadvantages of each programming model.

IX. Acknowledgements

All authors from Lawrence Berkeley National Laboratory were supported by the Office of Advanced Scientific Computing Research in the Department of Energy Office of Science under contract number DE-AC02-05CH11231. This work used resources of the National Energy Research Scientific Computing Center, under Contract No. DE-AC02-05CH11231, and resources of the National Center for Computational Sciences, under contract No. DE-AC05-00OR22725. The authors would like to thank Seung-Jai Min, Filip Blagojevic, and Paul Hargrove for their help collecting the UPC results.