Scientific Programming 18 (2010) 153–167
DOI 10.3233/SPR-2010-0306
IOS Press

A programming model performance study using the NAS parallel benchmarks

Hongzhang Shan [a,*], Filip Blagojević [a], Seung-Jai Min [a], Paul Hargrove [a], Haoqiang Jin [b], Karl Fuerlinger [c], Alice Koniges [d] and Nicholas J. Wright [d]

[a] *Future Technology Group, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA, USA*
[b] *NAS Division, NASA Ames Research Center, Moffett Field, CA, USA*
[c] *University of California at Berkeley, EECS Department, Computer Science Division, Berkeley, CA, USA*
[d] *NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA, USA*

Abstract. Harnessing the power of multicore platforms is challenging due to the additional levels of parallelism present. In this paper we use the NAS Parallel Benchmarks to study three programming models, MPI, OpenMP and PGAS to understand their performance and memory usage characteristics on current multicore architectures. To understand these characteristics we use the Integrated Performance Monitoring tool and other ways to measure communication versus computation time, as well as the fraction of the run time spent in OpenMP. The benchmarks are run on two different Cray XT5 systems and an Infiniband cluster. Our results show that in general the three programming models exhibit very similar performance characteristics. In a few cases, OpenMP is significantly faster because it explicitly avoids communication. For these particular cases, we were able to re-write the UPC versions and achieve equal performance to OpenMP. Using OpenMP was also the most advantageous in terms of memory usage. Also we compare performance differences between the two Cray systems, which have quad-core and hex-core processors. We show that at scale the performance is almost always slower on the hex-core system because of increased contention for network resources.

Keywords: Programming model, performance study, UPC, OpenMP, MPI, memory usage

1. Introduction

A new revolution in computer architecture is upon us, as the traditional increases in clock speed seen over the past 10 or more years disappear due to energy and other constraints on processor designs. In its place, we see a growing number of cores on a chip and thus effectively taking into account this enhanced parallelism is required to achieve good performance. Additionally, as the growth in memory capacity is not keeping track with the growth in the number of cores, memory considerations are becoming much more important. To address these issues researchers are considering new programming models, as well as exploring other traditional but less commonly used ones, to determine the best way to program these new architectures. In this study, we consider several programming models and analyze their performance using the NAS parallel benchmarks.

Traditionally, High Performance Computing (HPC) applications used an MPI-everywhere model 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 by sending and receiving messages via 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-

*Corresponding author: Hongzhang Shan, Future Technology Group, Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA. E-mail: hshan@lbl.gov.

1058-9244/10/$27.50 © 2010 – IOS Press and the authors. All rights reserved

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 larger problems to be tackled. 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 an Infiniband cluster 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 2 describes the two Cray XT5 machines and the Linux cluster we use in more detail, Section 3 describes our data collection techniques, Section 4 describes the results of our memory usage measurements and Section 5 contains the performance results comparing the different programming models. Section 6 contains the performance comparison between Jaguar (hex-core) and Hopper (quad-core) XT5 machines. Finally, Section 7 contains the related work and Section 8 summarizes.

2. 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 $1.6\times$ the computational power available per Jaguar node compared to a Hopper node. However, the memory and network subsystems on each node are same.

We also report performance results from Ranger, which is an AMD Opteron (Barcelona) system at TACC with 3936 nodes connected with two Sun Constellation InfiniBand switches, each of which has 3456 ports and a full Clos fat-tree topology. Each node contains four quad-core processors running at 2.3 GHz and a single SDR InfiniBand adapter. The Barcelona processors in Ranger are almost equivalent to those in Hopper. The principle difference between this and the Cray machines is the amount of network bandwidth available, one SDR IB link is approximately 0.8 GB/s whereas the XT5 architecture has 6 links per node each capable of 1.6 GB/s (9.6 GB/s/node).

We note that the NERSC XT5 machine used here is actually phase 1 of Hopper; a much larger machine, phase 2, will be installed in late 2010 and will replace the machine used here.

3. 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 [5], and the UPC codes come from a distribution developed by George Washington University (GWU), the Berkeley UPC group and NAS [1,3].

As well as recording the runtimes of our performance experiments we also instrument them using the Integrated Performance Monitoring (IPM) framework [6,8]. 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 trace points 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 execution times 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 and Ranger. For UPC, the Berkeley UPC compiler and runtime system are used [1], with PGI as a back-end compiler.

4. Memory usage of different programming models

Figure 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.

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.

Figure 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 $512 \times 512 \times 512 \times sizeof(\text{double complex}) = 2$ GB.

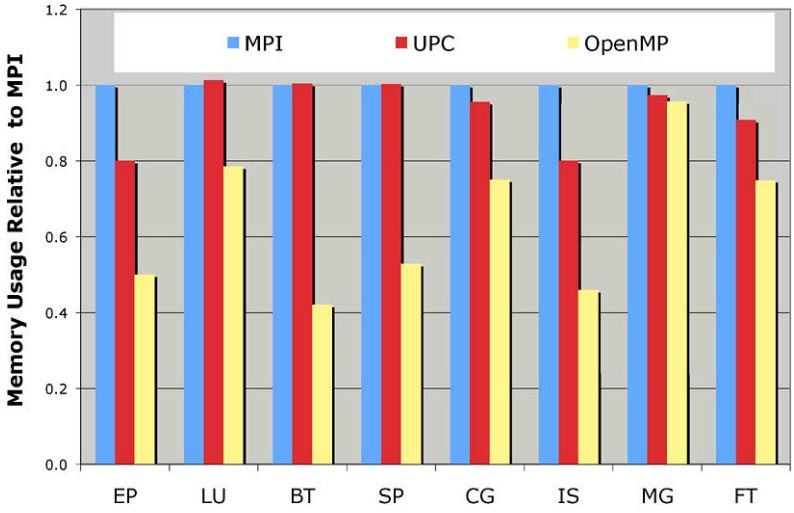


Fig. 1. Memory usage of NPB3.3 applications for MPI, OpenMP and UPC for four-core runs relative to MPI. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

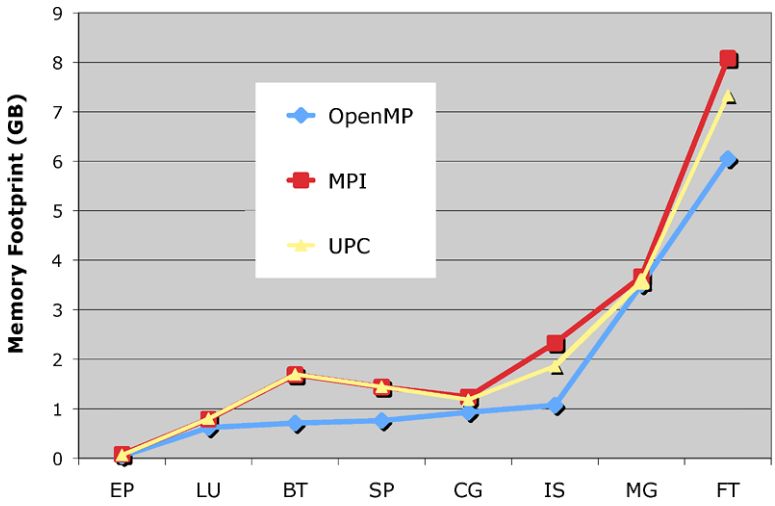


Fig. 2. The memory footprints of NPB3.3 applications for MPI, OpenMP and UPC for four-core runs. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

The main memory usage difference is caused by this extra data array of size 2 GB.

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 is due to some combination of the MPI runtime system requirements, communication buffers in the applications and 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.

4.1. 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.

Figure 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.

The base case is using 256 MPI processes and 1 OpenMP thread for each MPI process, i.e., setting the environment variable 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 used in hybrid programming mode, showing a great promise for its future.

5. Performance effects of different programming models

In this section, we investigate the performance differences between programming models (OpenMP, MPI and UPC) on two different platforms: Hopper and Ranger. Furthermore, we examine if the observed performance differences are due to the semantics of the programming models or the implementation and coding styles. Single-node performance for MPI, OpenMP and UPC are examined first. Then, the performance in cluster environments are compared to include the effects of the inter-node communication.

5.1. Single-node performance

Figure 4 represents the execution time ratio of UPC and MPI to OpenMP for single-node runs on Hopper and Ranger. Because BT and SP require a square number of tasks, we do not observe the performance of

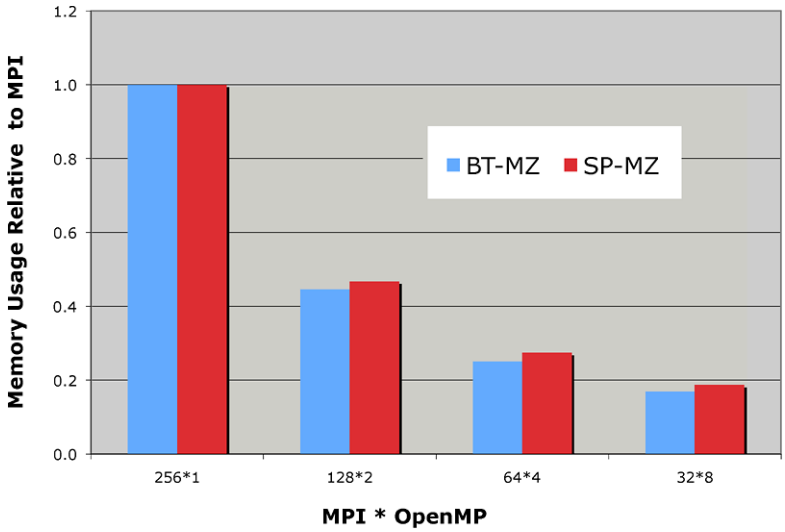


Fig. 3. The relative memory usage of BT-MZ and SP-MZ for MPI + OpenMP hybrid programming models for different combinations of MPI tasks and OpenMP threads. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

these benchmarks on Hopper (8 cores per node). For EP, CG, MG and LU, the three observed programming models deliver similar performance. However, for LU and SP on Ranger, OpenMP underperforms compared to MPI and UPC. The observed performance difference is attributed to the small granularity of the parallel loops.

For FT and IS OpenMP delivers the best performance. OpenMP is more than 40% faster than MPI and UPC for FT on Hopper, 50% faster than MPI on Ranger and 68% faster than UPC on Ranger. In case of IS, OpenMP outperforms MPI and UPC by up to 7% on Hopper. On Ranger OpenMP outperforms MPI by 63% and UPC by 45%.

Detailed examination of FT and IS shows that the explicit transpose and communication operations in the MPI versions cause performance degradation compared to the OpenMP ones. The UPC version of FT follows the MPI execution pattern, i.e., it also includes explicit transpose and communication phases. Because UPC-FT is derived from the Fortran version of MPI-FT, the data layout is specifically optimized for the column-wise accesses. UPC is a C language derivative and therefore the additional overhead of column-wise data accesses is visible during the computational phase. While it is possible to rearrange the data structures and further optimize the UPC-FT benchmark, these type of optimizations would only bring the UPC performance closer to MPI, still significantly below OpenMP.

The results presented in Fig. 4 indicate that the intra-node performance differences between programming models are mainly caused by the overhead of explicit communication. In MPI the explicit communication is unavoidable. If translated directly from the MPI code, the UPC code also inherits the explicit communication overhead. However, UPC also allows for a shared-memory programming style and therefore can be used in OpenMP fashion within a single node, thus avoiding the communication overhead. We provide more details about the UPC shared-memory programming style in the next section.

5.2. *UPC shared-memory execution*

The UPC applications used in this study were derived from the MPI-based implementations, and therefore follow identical execution patterns. Consequently, the UPC-NAS benchmarks do not fully utilize the global address space abstraction, an important feature of PGAS languages. In this section we focus on our redesign of the UPC-IS and FT benchmarks to exploit shared-memory communication and avoid explicit communication.

The new UPC versions follow a parallelization scheme similar to a hybrid OpenMP-MPI code. Only the master-thread performs communication and multiple UPC threads perform computation in parallel between communication events. Figure 5 represents the performance of IS and FT benchmarks using this UPC shared-memory programming model. We observe significant benefits by avoiding explicit communication

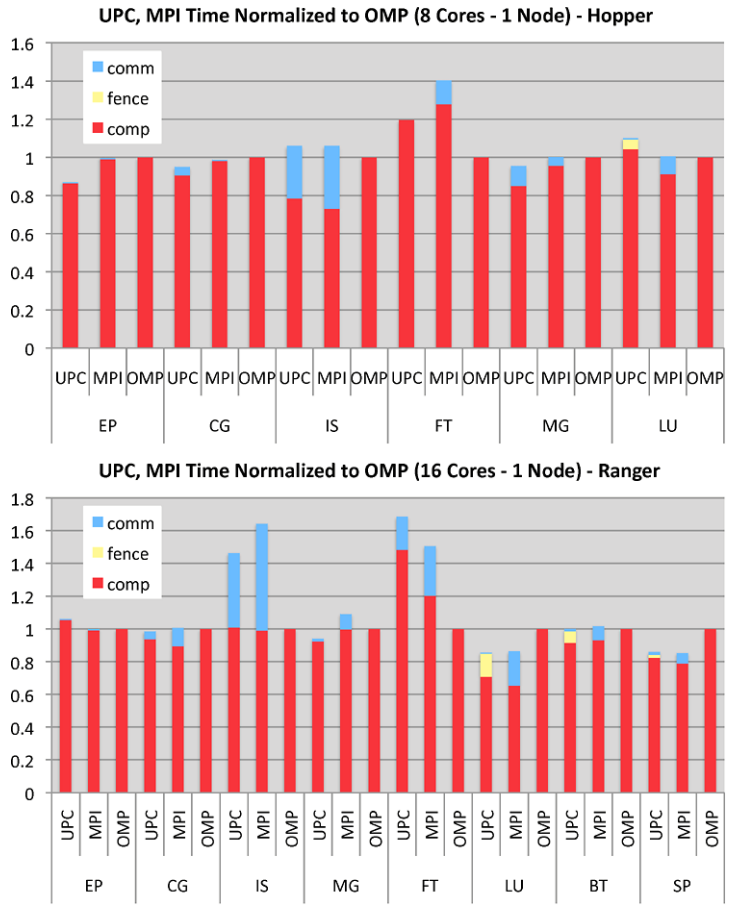


Fig. 4. Execution time of UPC and MPI relative to OpenMP on Hopper and Ranger. On both platforms we use only a single node, 8 cores on Hopper and 16 cores on Ranger. *comm* – communication time, *comp* – computation time, *fence* – time spent in the `upc_fence` operation. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

within a single node. Note that for the new UPC versions of IS and FT we were able to entirely eliminate the communication overhead. Single-node performance for UPC shared-memory IS and FT is comparable to OpenMP and significantly outperforms MPI.

While capable of achieving equivalent performance to OpenMP, the UPC shared-memory method currently uses more memory than the OpenMP version of the same code. This is because the current UPC runtime requires all the UPC threads to equally participate in the memory allocation of the shared heap. Obviously, for the shared-memory UPC programming model this is not optimal, only the part of the heap that belongs to the master thread is actually used. We are working on enabling a UPC runtime feature that will enable uneven heap distribution across UPC threads, which should bring the memory usage in line with that of the OpenMP version.

5.3. *Distributed environment performance*

In a distributed environment, we compare the performance of MPI and UPC. Figure 6 represents the execution time of UPC normalized to the execution time of MPI, on 64 cores of Hopper and Ranger. With the exception of LU and SP on Hopper and MG on Ranger the performance of UPC and MPI are very similar.

The UPC versions of LU, BT and SP implement pipelined algorithms with point-to-point communication, that are highly optimized for communication/computation overlap. To better understand the observable performance degradation in UPC-LU and SP, we divide the total execution time into three parts: computation, communication and the time spent in `upc_fence` operation. The absolute performance numbers for these three benchmarks are presented in Table 1.

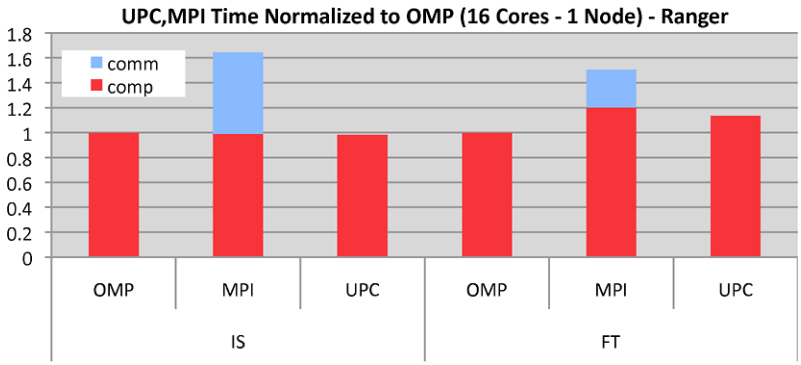


Fig. 5. Execution time of UPC-IS, FT and MPI-IS, FT, relative to OpenMP-IS and FT on Ranger. The UPC-IS and FT benchmarks are programmed in the *shared-memory* style, avoiding any explicit communication. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/ SPR-2010-0306.)

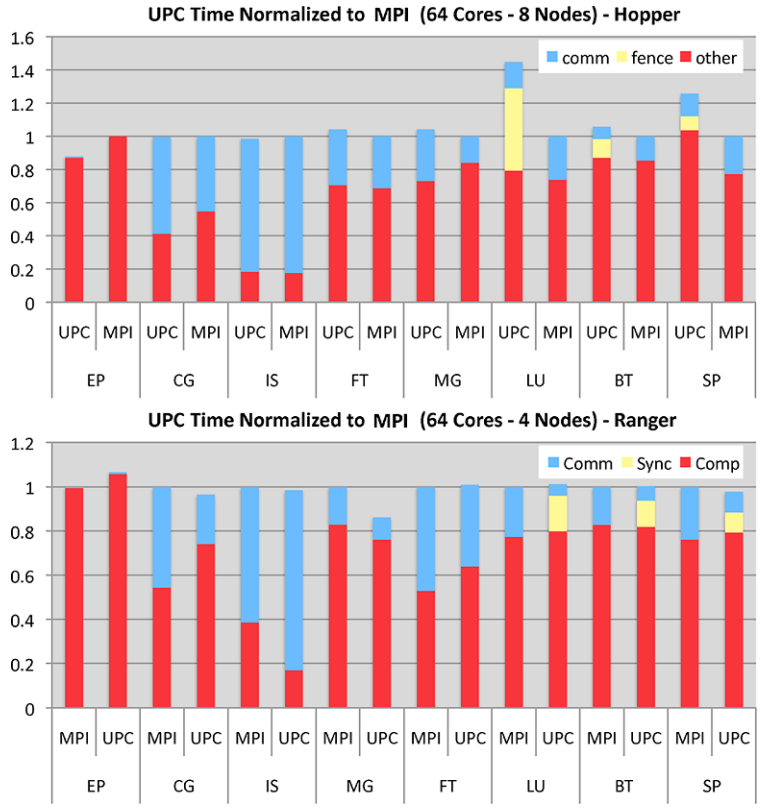


Fig. 6. Execution time of UPC relative to MPI on Hopper and Ranger for 64 cores. *comm* – communication time, *comp* – computation time, *fence* – time spent in the `upc_fence` operation. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

The point-to-point synchronization in the UPC benchmarks is implemented through a shared variable: a UPC task is polling over a shared variable waiting for another task to change the value of the same variable. The polling is always performed on a shared variable with affinity to the local node, to avoid excessive inter-node communication. To ensure the progress of each UPC thread, the polling loop contains the `upc_fence` instruction, which further queries the network for the reception of new active messages. On Hopper (SeaStar2 interconnect) we detected significant overhead while multiple UPC tasks simultaneously perform network polling, indicating contention within the network driver. The overhead of network

polling is also visible on Ranger, but the performance impact is significantly lower than on Hopper.

It is interesting to note that UPC-MG on Ranger outperforms the MPI version by up to 20%. On Hopper however this difference is not present. The presented MG performance on Ranger and Hopper are obtained with two different variants of the UPC benchmarks: (i) on Hopper, following the MPI-MG implementation, the fine-grain communication messages are aggregated into a single larger message; (ii) on Ranger, we found the more beneficial approach to be if the message aggregation overhead is avoided and instead a large number of fine-grain messages is used for communication.

While the overall performance of the UPC and MPI benchmarks are similar, the presented results reveal certain tradeoffs between the two observed programming models. Frequent network polling can lower the UPC performance on certain platforms. The observed behavior can be improved by fine-tuning the UPC network polling code for the problematic architecture. In terms of productivity, one-sided communication available in UPC enables effortless usage of fine-grained communication, which further avoids message aggregation overhead. The reader should note that the same effect can be achieved with MPI, with somewhat increased programming complexity, due to the lack of one-sided communication.

Table 1

The communication time and computation time of UPC and MPI for 64-core runs on Hopper

	LU		BT		SP	
	UPC	MPI	UPC	MPI	UPC	MPI
Fence	14.5	0	4	0	3.34	0
Comm	4.69	7.71	2.68	5.27	5.26	8.89
Comp	23.44	21.77	31.27	30.64	40.65	30.28

5.4. 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, the time breakdown is shown in Fig. 7 and Table 2. 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.

The SP-MZ performance scales very well with the number 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

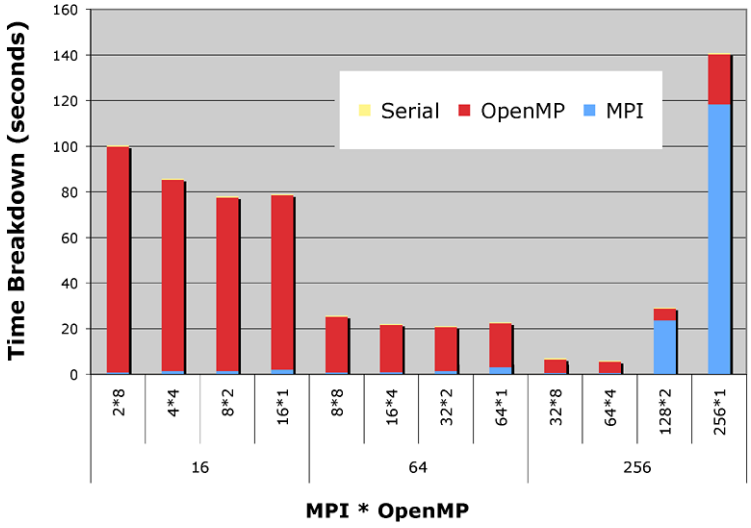


Fig. 7. The BT-MZ time breakdown on Hopper. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

number of cores used (see Table 3). 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. This is not caused by the OpenMP overhead to activate and idle the OpenMP threads (which is further explored in the next section). Instead, this is related with two factors. One is load imbalance. When four or eight OpenMP threads are used, the work (the length of the for loop) can not be evenly divided among the threads. The other is memory contention and cache coherence protocols.

Table 2
The MPI, OpenMP and serial times (s) 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

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 4). This is due to the following reasons. First, each MPI process is assigned several zones to work on and the OpenMP threads spawned by the same MPI process will then work on these zones together, one at a time. Using more OpenMP threads will increase the overall cache size for the same amount of data and improve the performance (assuming one OpenMP thread assigned to one core). The effect of using more caches is more important for LU-MZ than SP-MZ and BT-MZ. In LU-MZ, there are total of 16 zones while in BT-MZ and SP-MZ, there are total of 256 zones. Therefore zone size in LU-MZ is much larger than the zone size in BT-MZ and SP-MZ. Secondly, the data access exhibits much better spatial locality.

5.5. *OpenMP overhead analysis*

For hybrid MPI + OpenMP applications, using more OpenMP threads will potentially improve the cache usage and communication patterns while it may also increase spawning and activation/deactivation overhead, causing more memory contention and cache coherence protocol activities. In this section, we will use STREAM [7] to help us to understand the performance effect of some of these factors as the high spatial local-

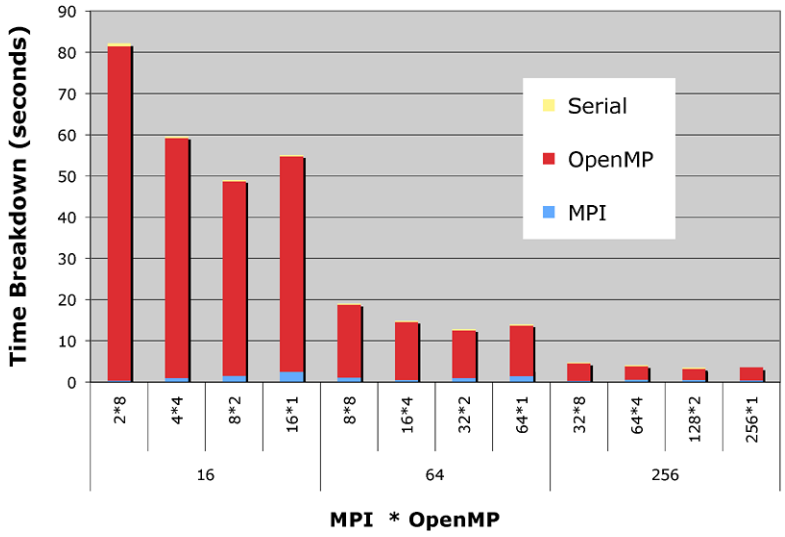


Fig. 8. The SP-MZ time breakdown on Hopper. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

Table 3
The MPI, OpenMP and serial times (s) 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 4
The MPI, OpenMP and serial times (s) 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

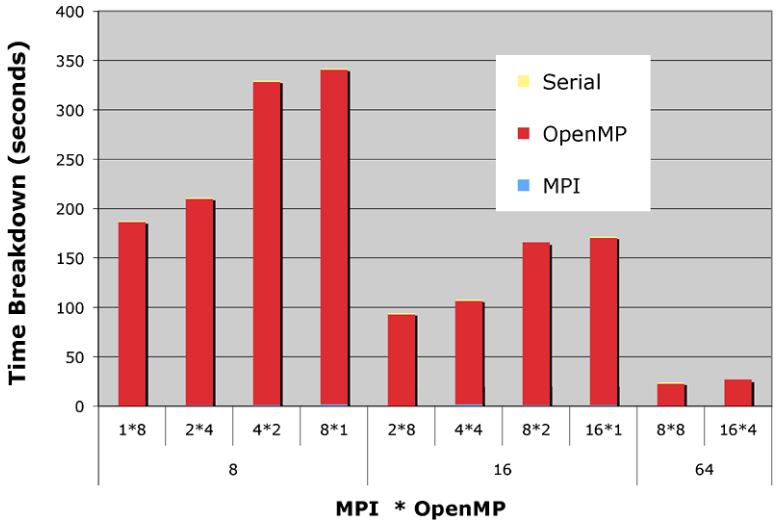


Fig. 9. The LU-MZ time breakdown on Hopper. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

ity it exhibits make it straightforward for OpenMP to work effectively.

Figure 10 shows the OpenMP runtime of STREAM for different array sizes. When the array size is equal to or smaller than 1024, using more OpenMP threads, instead of reducing the total running time, actually increases it. For these smaller array sizes, the overhead to activate and idle the OpenMP threads overwhelms the advantage of partitioning the work. This situation lasts until the array size becomes 4096. At this point, the overhead for using OpenMP "parallel for" directives starts to be amortized and the advantage of using more threads becomes apparent.

The actual overheads for OpenMP threads to enter and exit the loop are shown in Table 5 for PGI compiler. It only takes about 2 μs for eight OpenMP threads. The spawning overhead is, however, relatively larger. It takes about 60 μs to spawn one OpenMP thread and 275 μs for eight OpenMP threads. However, since these threads are only spawned once during the entire program execution, the spawning overhead is not a significant performance issue.

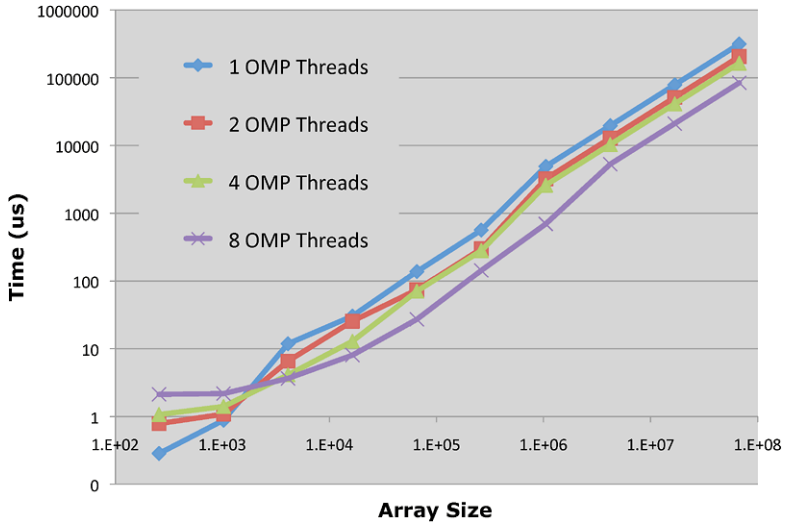


Fig. 10. The OpenMP performance of STREAM on Hopper. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

Table 5

The parallel for and spawning overhead for OpenMP threads on Hopper (μs)

	1	2	4	8
Parallel for	0.3	0.7	1.2	2.0
Spawning	60	102	167	275

The performance of OpenMP will also be highly related with the peak memory bandwidth, initial data placement, false sharing, cache size and other factors. However, quantifying their effects will be difficult as they are highly related with specific applications.

6. 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.

6.1. Single-node performance

On Jaguar, each node has 12 cores, and the MPI and UPC version of NPB3.3 cannot 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 1.08 (2.6/2.4) due to the different CPU frequencies or 1.0 due to same memory bandwidth. This is exactly the case as shown in Fig. 11. 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 ($6 \times 2.6/4 \times 2.4$). 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.

6.2. 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 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. 12 and the corresponding normalized runtime breakdown

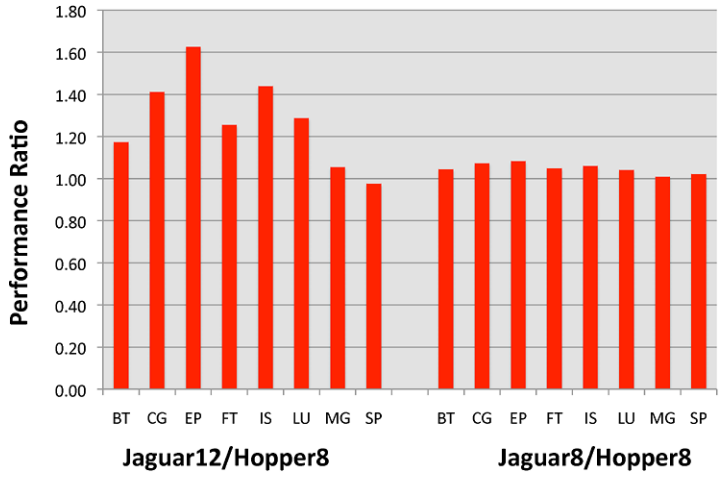


Fig. 11. The performance ratio of Jaguar vs. Hopper for OpenMP model on a node. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

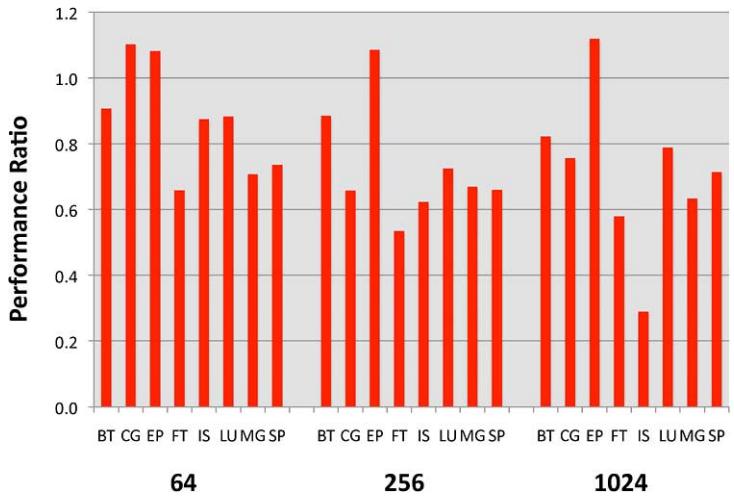


Fig. 12. The performance ratio of Jaguar vs. Hopper for MPI programs. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

is shown in Fig. 13 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. 13. 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 multi-zone benchmarks. For BT-MZ, the performance ratio is very close to the expected value, one, as shown in Fig. 14. 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.

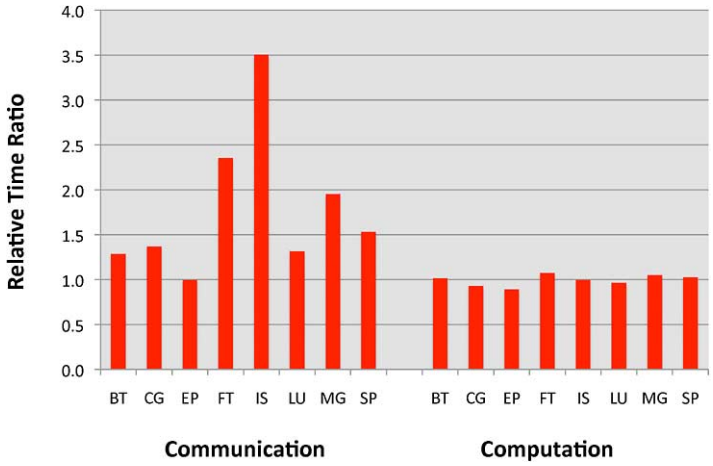


Fig. 13. The normalized time breakdown of Jaguar vs. Hopper for MPI programs using 1024 cores. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

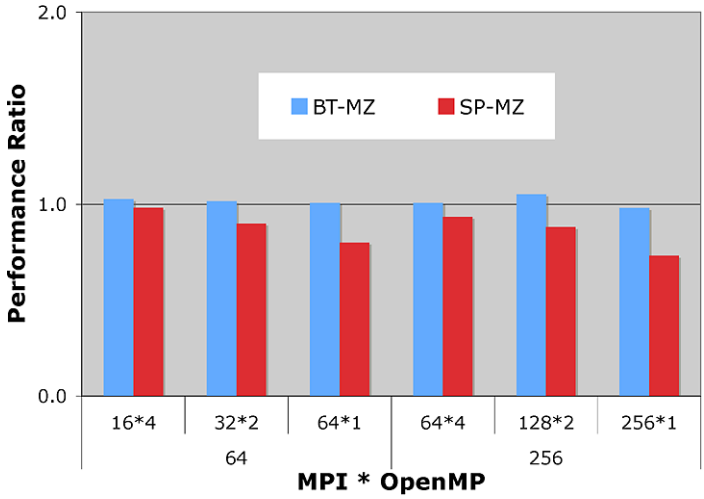


Fig. 14. The performance ratio of Jaguar vs. Hopper for NPB3.3-MZ. (Colors are visible in the online version of the article; http://dx.doi.org/10.3233/SPR-2010-0306.)

7. Related work

A lot of literature has been published comparing different programming models. Among them, the most related to our work is [4], 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 [2]. 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 provide detailed time breakdowns to compare the MPI performance and UPC performance. This differs from previous research which only compare the absolute performance of MPI and UPC [3,4].

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 quantitatively analyzed and compared.

8. Conclusions

In this paper we have examined the performance of three different programming models OpenMP, MPI and UPC on the Cray XT5 machine, Hopper at NERSC, and the Linux InfiniBand cluster Ranger at TACC. As well as simply measuring the runtime by using IPM and by inserting explicit timers into the code we were able to 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 single-node performance of each of the different programming methods is very close for the NAS Parallel Benchmarks. In the cases that show the most performance difference it is always OpenMP that is the fastest. Our performance analysis shows that this is always due to the reduced communication costs in the shared-memory model. Our initial results showed that for the FT and IS benchmarks this effect was especially pronounced. For the UPC case, this was mostly because the UPC version was a translation of the MPI one and used the same communication algorithm. When we produced shared-memory style versions using UPC that took full advantage of the features of the language the performance of IS and FT was almost identical to that of OpenMP. This nicely illustrates the need to take into account the features of a programming model that allow closest match to the attributes of the hardware. We also compared the performance of MPI and UPC on 64 cores of Hopper and Ranger. The results showed that the performance of the two methods is almost identical in this case.

We 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 overheads present when using OpenMP and compared their cost to the cost of a simple loop of the stream program. We found that the overhead with eight threads was approximately 2 μs. This is useful to bear in mind for application developers looking to add OpenMP directives to their codes – any loop that takes less than this amount of time is not going to gain any performance benefit.

Finally, we 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× 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.

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 No. 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.

References

[1] Berkeley UPC – Unified Parallel C, available at: http://upc.lbl.gov.

[2] G. Hager, G. Jost and R. Rabenseifner, Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: *The Cray User Group Conference*, May 2009.

[3] H. Jin, R. Hood and P. Mehrota, A practical study of UPC with the NAS parallel benchmarks, in: *Partitioned Global Address Space Languages*, October 2009.

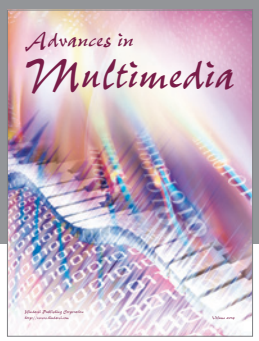
[4] D.A. Mallon, G.L. Taboada, C. Teijeiro, J. Tourino, B.B. Fraguela, A. Gomez, R. Doallo and J.C. Mourino, Performance evaluation of MPI, UPC, and OpenMP on multicore architectures, in: *Euro PVM/MPI 2009*, September 7–10, 2009.

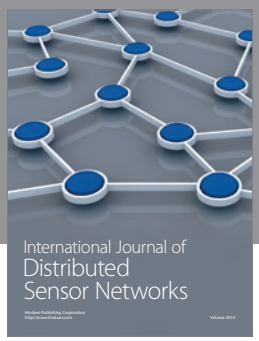
[5] NAS Parallel Benchmarks, available at: http://www.nas.nasa.gov/Resources/Software/npb.html.

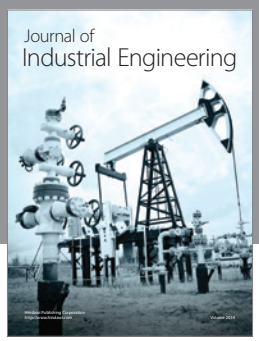
[6] D. Skinner, Integrated Performance Monitoring: A portable profiling infrastructure for parallel applications, in: *Proc. ISC2005: International Supercomputing Conference*, Heidelberg, Germany, 2005.

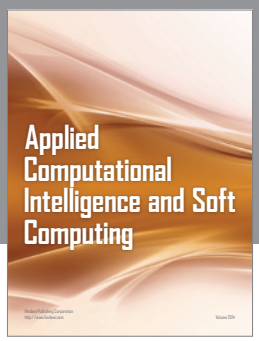
[7] STREAM, Sustainable memory bandwidth in high performance computers, available at: http://www.cs.virginia.edu/stream.

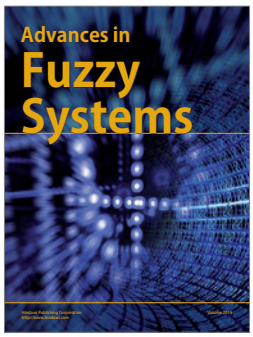
[8] N.J. Wright, W. Pfeiffer and A. Snavely, Characterizing parallel scaling of scientific applications using IPM, in: *The 10th LCI International Conference on High-Performance Clustered Computing*, March 10–12, 2009.

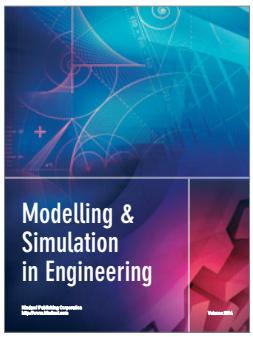


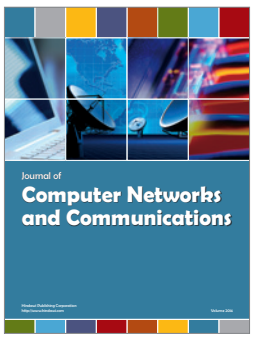












Hindawi
Submit your manuscripts at
http://www.hindawi.com