Parallel Computing 37 (2011) 562–575

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

High performance computing using MPI and OpenMP on multi-core parallel systems

Haoqiang Jin [a,*], Dennis Jespersen [a], Piyush Mehrotra [a], Rupak Biswas [a], Lei Huang [b], Barbara Chapman [b]

[a] NAS Division, NASA Ames Research Center, Moffett Field, CA 94035, United States
[b] Department of Computer Sciences, University of Houston, Houston, TX 77004, United States

ARTICLE INFO

Article history:
Available online 2 March 2011

Keywords:
Hybrid MPI + OpenMP programming
Multi-core Systems
OpenMP Extensions
Data Locality

ABSTRACT

The rapidly increasing number of cores in modern microprocessors is pushing the current high performance computing (HPC) systems into the petascale and exascale era. The hybrid nature of these systems – distributed memory across nodes and shared memory with non-uniform memory access within each node – poses a challenge to application developers. In this paper, we study a hybrid approach to programming such systems – a combination of two traditional programming models, MPI and OpenMP. We present the performance of standard benchmarks from the multi-zone NAS Parallel Benchmarks and two full applications using this approach on several multi-core based systems including an SGI Altix 4700, an IBM p575+ and an SGI Altix ICE 8200EX. We also present new data locality extensions to OpenMP to better match the hierarchical memory structure of multi-core architectures.

Published by Elsevier B.V.

1. Introduction

Multiple computing cores are becoming ubiquitous in commodity microprocessor chips, allowing the chip manufacturers to achieve high aggregate performance while avoiding the power demands of increased clock speeds. We expect the core counts per chip to rise as transistor counts increase according to Moore's law. As evidenced by the systems on the TOP500 list [30], the high-end computing system manufacturers of today take advantage of economies of scale by using such commodity parts to build large HPC systems. The basic building block of such systems consists of multiple multi-core chips sharing memory in a single node. The dominant HPC architectures, currently and for the foreseeable future, comprise clusters of such nodes interconnected via a high-speed network supporting a heterogeneous memory model—shared memory with non-uniform memory access (NUMA) within a single node and distributed memory across the nodes.

From the user's perspective, the most convenient approach to programming these systems is to ignore the hybrid memory system and use a pure message passing programming model. This is a reasonable strategy on most systems, since most Message Passing Interface (MPI) library developers take advantage of the shared memory within a node and optimize the intra-node communication. Thus, the users get good performance for their applications while using a single standardized programming model. However there are other approaches to programming such systems. These include new languages such as those developed for DARPA's High Productivity Computing Systems (HPCS) program, Fortress [1], Chapel [6] and X10 [7], which are attempting to provide a very high level approach to programming petascale and exascale systems. Since these languages have just been proposed in the last few years and their implementations are in the nascent stages, it is difficult to predict their ultimate utility and performance.

* Corresponding author.
E-mail address: haoqiang.jin@nasa.gov (H. Jin).

0167-8191/$ - see front matter Published by Elsevier B.V.
doi:10.1016/j.parco.2011.02.002

Another approach is embodied by the partitioned global address space (PGAS) languages that try to combine the best features of the shared memory programming model and the message passing model. Languages such as Co-Array Fortran [21], UPC [31] and Titanium [33] that utilize this model let users control data locality by specifying the distribution of data while enhancing programmability by allowing them to use a global name space with explicit specification of the remote data accesses. While these languages provide an incremental approach to programing HPC systems, studies have shown that the user has to carefully utilize the language features to obtain the performance of equivalent MPI programs [16].

In this paper, we focus on a hybrid approach to programming multi-core based HPC systems, combining standardized programming models – MPI for distributed memory systems and OpenMP for shared memory systems. Although this approach has been utilized by many users to implement their applications, we study the performance characteristics of this approach using some standard benchmarks, the multi-zone NAS Parallel Benchmarks (NPB-MZ), and two full applications used by NASA engineers, OVERFLOW and AKIE. We measure the performance of various hybrid configurations of these codes on several platforms including an SGI Altix 4700 (Intel Itanium2 chip), an IBM p575+ (IBM Power5+ chip) and an SGI Altix ICE 8200EX (Intel Xeon Nehalem-EP chip).

The paper also describes a new approach in which we extend the OpenMP model with new data locality extensions to better match the more complex memory subsystems available on modern HPC systems. The idea is to allow the user to use the simple shared memory model of OpenMP while providing "hints" in the form of data locality pragma to the compiler and the runtime system for them to efficiently target the hierarchical memory subsystems of the underlying architectures.

The paper is organized as follows. In the next section, we briefly describe the MPI and OpenMP models and the hybrid approach that we have studied in this paper. Section 3 presents the descriptions of the multi-zone NAS Parallel Benchmarks and the two applications OVERFLOW and AKIE. Section 4 starts with a description of the parallel systems used in the study, discusses the performance results that we obtained and concludes with a brief summary of our results. In Section 5, we present the proposed extensions to OpenMP describing the syntax and semantics of the various directives. Sections 6 and 7 describe some related work and the conclusions of our paper, respectively.

2. MPI and OpenMP programming models

MPI and OpenMP are the two primary programming models that have been widely used for many years for high performance computing. There have been many efforts studying various aspects of the two models from programmability to performance. Our focus here is the hybrid approach based on the two models, which becomes more important for modern multi-core parallel systems. We will first give a brief summary of each model to motivate our study of this hybrid approach.

2.1. Why still MPI?

MPI is a *de facto* standard for parallel programming on distributed memory systems. The most important advantages of this model are twofold: achievable performance and portability. Performance is a direct result of available optimized MPI libraries and full user control in the program development cycle. Portability arises from the standard API and the existence of MPI libraries on a wide range of machines. In general, an MPI program can run on both distributed and shared memory machines.

The primary difficulty with the MPI model is its discrete memory view in programming, which makes it hard to write and often involves a very long development cycle [13,14]. A good MPI program requires carefully thought out strategies for partitioning the data and managing the resultant communication. Secondly, due to the distributed nature of the model, global data may be duplicated for performance reasons, resulting in an increase in the overall memory requirement. Thirdly, it is very challenging to get very large MPI jobs running on large parallel systems due to machine instabilities and the lack of fault tolerance in the model (although this is true in general for other programming models). Also, large MPI jobs often demand substantial resources such as memory and network.

Lastly, most MPI libraries have a large set of runtime parameters that need to be "tuned" to achieve optimal performance. Although a vendor usually provides a default set of "optimized" parameters, it is not easy for a typical user to grasp the semantics and the optimal usage of these parameters on a specific target system. There is a hidden cost due to the underlying library implementation, which often gets overlooked. As an example, Table 1 lists timings measured for the CART3D

Table 1
CART3D timing (seconds) on a Xeon cluster.

Number of processes	8 GB node	16 GB node
32	*failed*	297.32
64	298.35	162.88
128	124.16	83.97
256	*failed*	42.62
`MPI_BUFS_PER_HOST`	48	256
`MPI_BUFS_PER_PROC`	32	128

application [4] on a cluster of Intel Xeon E5472 (Harpertown) nodes with two sizes of node memory [9]. The two parameters, `MPI_BUFS_PER_HOST` and `MPI_BUFS_PER_PROC`, control how much internal buffer space is allocated for communication in the MPI library. The results show a large impact on performance (more than 50%) from different MPI buffer sizes. The two failed cases on the smaller memory (8 GB) nodes are due to running out of memory in these nodes, but from different causes: at 32 processes, the application itself needs more memory; at 256 processes, memory usage of internal MPI buffers is too large (which may increase as a quadratic function of the number of processes, as we find out in Section 4).

2.2. Why OpenMP?

OpenMP, based on compiler directives and a set of supporting library calls, is a portable approach for parallel programming on shared memory systems. The OpenMP 2.5 specification (and its earlier versions) focuses on loop-level parallelism. With the introduction of *tasks* in the OpenMP 3.0 specification [22], the language has become more dynamic. OpenMP is supported by almost all major compiler vendors. The well-known advantage of OpenMP is its global view of application memory address space that allows relatively fast development of parallel applications. The directive based approach makes it possible to write sequentially consistent codes for easier maintenance. OpenMP offers an incremental approach towards parallelization, which is advantageous over MPI in the parallel code development cycle.

The difficulty with OpenMP, on the other hand, is that it is often hard to get decent performance, especially at large scale. This is due to the fine-grained memory access governed by the memory model that is unaware of the non-uniform memory access characteristics of the underlying shared address space system. It is possible to write SPMD-style codes by using thread ids and managing data explicitly as is done in an MPI code and researchers have demonstrated success using this approach to achieve good performance with OpenMP in real-world large-scale applications [4,19]. However, this defeats some of the advantages in using OpenMP.

2.3. The MPI + OpenMP approach

As clusters of shared memory nodes become the dominant parallel architecture, it is natural to consider the hybrid MPI + OpenMP approach. The hybrid model fits well with the hierarchical machine model, in which MPI is used for communication across distributed memory nodes and OpenMP is used for fine-grained parallelization within a node. Taking advantages of both worlds, the hybrid model in principle can maintain cross-node performance with MPI and reduce the number MPI processes needed within a node and thus, the associated overhead (e.g., message buffers). As we discuss later in Section 4.2, the hybrid approach can help reduce the demand for resources (such as memory and network), which can be very important for running very large jobs. For certain class of applications with easily exploitable multi-level parallelism, the hybrid model can potentially reduce application development effort also. In the following sections, we will examine the aforementioned advantages and potential disadvantages of the hybrid approach in a number of case studies.

3. Hybrid programming case studies

This section briefly describes the characteristics of the applications that were implemented using the hybrid MPI + OpenMP model, two pseudo-application benchmarks and two real-world applications. The next section will focus on the performance of these applications on several parallel systems.

3.1. Multi-zone NAS Parallel Benchmarks

The multi-zone versions of NAS Parallel Benchmarks (NPB-MZ) are derived from the pseudo-application benchmarks included in the regular NPBs [2,32]. They mimic many real-world multi-block codes that contain exploitable parallelism at multiple levels. Due to the structure of the physical problem, it is natural to use MPI for zonal coarse grain parallelism and OpenMP for fine grain parallelism within a zone. There are three benchmark problems defined in NPB-MZ, as summarized in Table 2, with problem size ranging from Class S (the smallest) to Class F (the largest). Both SP-MZ and LU-MZ have fixed-size zones for a given problem class and load balancing in this case is simple and straightforward. On the other hand, the zone sizes for a given class in BT-MZ can vary by as much as a factor of 20, raising load balancing issues. Due to the fixed number of zones in LU-MZ, the exploitable zonal parallelism for this benchmark is limited.

Table 2
Three benchmark problems in NPB-MZ.

Benchmark	Number of zones	Zonal size
BT-MZ	Increases with problem size	Varies within a class
SP-MZ	Increases with problem size	Fixed within a class
LU-MZ	Fixed (=16)	Fixed within a class

The hybrid MPI + OpenMP implementation of NPB-MZ uses MPI for inter-zone parallelization and a bin-packing algorithm for balancing loads among different zones. There is no further domain decomposition for each zone. OpenMP parallelization is used for loops within a zone. For details of the implementations, please refer to [15]. Because of the limited number of zones (=16) in LU-MZ, which limits the number of MPI processes, we have not used this benchmark in this study.

3.2. Overflow

OVERFLOW is a general-purpose Navier–Stokes solver for Computational Fluid Dynamics (CFD) problems [20]. The code is focused on high-fidelity viscous simulations around realistic aerospace configurations. The geometric region of interest is decomposed into one or more curvilinear but logically Cartesian meshes. Arbitrary overlapping of meshes is allowed. The code uses finite differences in space, implicit time stepping, and explicit exchange of information at overlap boundaries. For parallelization, meshes are grouped and assigned to MPI processes. For load balancing purposes, each mesh may be further decomposed into smaller domains. During the solution process these smaller domains act as logically independent meshes. The hybrid decomposition for OVERFLOW involves OpenMP parallelism underneath the MPI parallelism. All MPI ranks have the same number of OpenMP threads. The OpenMP shared-memory parallelism is at a fairly fine-grained level within a domain.

The numerical scheme in OVERFLOW involves a flow solver step on each mesh followed by an explicit exchange of boundary overlap information. For a given case, as the number of MPI processes increases, more mesh splitting generally occurs in order to get good load balancing. There are a few drawbacks to this strategy. First, more mesh splitting results in a numerical algorithm which has more of an explicit flavor (with unlimited mesh splitting the domain sizes could shrink to just a few mesh points each). This reduction of implicitness can result in slower convergence (or even non-convergence). Slower convergence implies that more iterations (and thus longer wallclock time) would be needed to reach the desired solution. This is illustrated in Fig. 1 for an airfoil test case, showing the effect on the convergence rate of splitting a single mesh into 120 zones. Without mesh splitting, the calculated L2 residual monotonically decreases as the iteration proceeds. For the split case, the residual appears to be non-convergent. This is admittedly a highly unrealistic example, but for realistic cases a possible change in convergence behavior due to mesh splitting can add an undesired degree of uncertainty to what may already be a difficult solution process.

Secondly, more mesh splitting causes an increase in the number of mesh points, since "ghost" or "halo" points are used for affecting data interpolation at split boundaries. Table 3 shows the total number of mesh points for different numbers of domain groups produced from a dataset with 23 zones and 36 million mesh points. For 256 groups, the total mesh size is increased by 30%. Since the flow code solution time is essentially proportional to the total number of mesh points, the increase in mesh points partially counteracts the efficiency of increased parallelism.

The hybrid approach results in less mesh splitting, which provides twofold benefit: there are fewer total mesh points, and convergence behavior is less likely to be adversely affected. This is due to the fact that both these properties depend on the total number of MPI ranks and not on the total number of processor cores. The impact of hybrid versus MPI parallelization on the convergence rate of implicit methods has also been pointed out by Kaushik and his colleagues [17].

3.3. Akie

AKIE is a turbine machinery application that is used to study 3D flow instability around rotor blades [11]. The multi-block configuration consists of solving partial differential equations for flow around each blade and interpolating solutions between blades. Parallelism can be exploited at both inter-blade and intra-blade levels. For performance reason, the code uses a 32-bit numerical algorithm. We started with a sequential version of AKIE and developed the hybrid MPI + OpenMP version

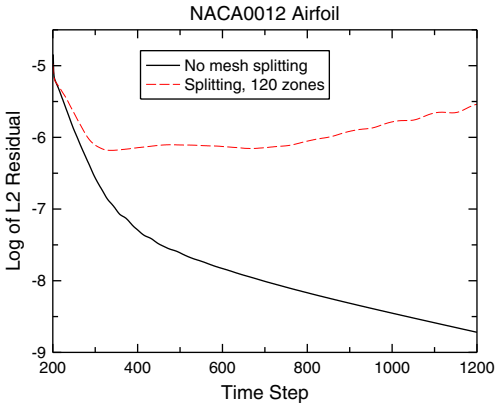

Fig. 1. Numerical convergence rate as a result of mesh splitting.

Table 3
OVERFLOW mesh size for different numbers of domain groups.

Number of groups	Total number of mesh points (M)
16	37
32	38
64	41
128	43
256	47

(*p1h*). Since this version did not achieve the desired scalability (notably on the Altix 4700 as discussed in the following section), we developed a pure MPI version (*p2d*) as well. Both versions are based on a hierarchical two-level parallelization approach with the difference being only in how the second-level parallelization is done, as illustrated in Fig. 2. The first-level parallelization applies domain decomposition to the blade dimension. Each MPI process (*P0*···*Pn*) works on one group of blades and exchanges boundary data at the end of each iteration. Parallelism at the first level is limited to the number of blades available in a given problem. In order to exploit additional parallelism, the second-level parallelization within a blade would be needed.

For the second-level parallelization, the hybrid version (*p1h*) uses OpenMP parallel regions for loops (mostly applied to the *j* dimension of the 3D space) within a blade. Thread synchronization is used to avoid race conditions in different regions. In contrast, the pure MPI version (*p2d*) applies further domain decomposition (to the *j* dimension) within a blade with an MPI process assigned to each sub-domain. Thus, each first-level MPI group consists of sub-level MPI processes working on sub-domains and performing additional communication within the group. There is no change in algorithm compared to the OpenMP approach; that is, both versions maintain the same numerical accuracy. For these reasons, AKIE is a good case for comparing the MPI and hybrid approaches. We should point out that both the hybrid and MPI versions were implemented from the same code base. In the actual implementation of the pure MPI version (*p2d*), a two-dimensional layout of the MPI communication groups is used to match with that of the data decomposition. The MPI processes at the two levels are initiated statically and there is no use of MPI-2's dynamic process creation in the code.

4. Performance studies

Our studies of the performance of the hybrid MPI + OpenMP applications were conducted on three multi-core parallel systems located at the NASA Advanced Supercomputing (NAS) Division, NASA Ames Research Center. We first briefly describe the parallel systems, and then compare performance of the hybrid applications. Throughout the rest of the paper, we use the terms "core" and "CPU" interchangeably.

4.1. Parallel systems

Table 4 summarizes the main characteristics of the three parallel systems used for this study. The first system, an SGI Altix 4700, is part of the Columbia supercluster and consists of 512 compute blades. Each blade hosts two dual-core Intel Itanium2 processors that share the same local memory board through a bus. The system has a cache-coherent non-uniform

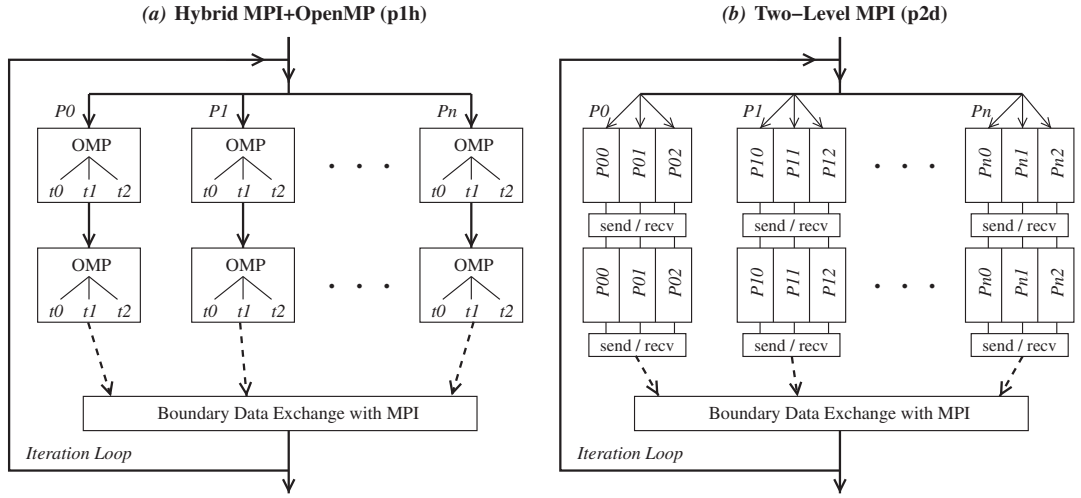

Fig. 2. Schematic comparison of the AKIE (a) hybrid and (b) MPI codes.

Table 4
Summary of the parallel systems.

System name	Columbia22	Schirra	Pleiades (Nehalem)
System type	SGI Altix 4700	IBM p575+	SGI Altix ICE 8200EX
Processor type	Intel Itanium2	IBM Power5+	Intel Xeon
Model	9040		X5570
Series (# Cores)	Montecito (2)	(2)	Nehalem-EP (4)
Processor speed	1.6 GHz	1.9 GHz	2.93 GHz
HT or SMT (Threads/Core)	–	2	2
L2 Cache (cores sharing)	256 KB (1)	1.92 MB (2)	256 KB (1)
L3 Cache (cores sharing)	9 MB (1)	36 MB (2)	8 MB (4)
Memory interface	FSB	On-Chip	On-Chip
FSB speed	667 MHz	–	–
No. of hosts	1	40	1280
Cores/host	2048	16	8
Total cores	2048	640	10240
Memory/host	4096 GB	32 GB	24 GB
Interconnect	NUMALink4	IBM HPS	InfiniBand
Topology	FatTree	FatTree	Hypercube
Operating system	SLES 10.2	AIX 5.3	SLES 10.2
Compiler	Intel-10.1	IBM XLF-10.1	Intel-11.1
MPI library	SGI MPT-1.9	IBM POE-4.3	SGI MPT-1.25

memory access (ccNUMA) architecture supported by the NUMAlink4 interconnect for communication between blades, which enables globally addressable space for 2048 cores in the system. The second system, an IBM p575+, is a cluster of IBM Power5+ nodes connected with the IBM high-performance switch (HPS). Each node contains 8 dual-core IBM Power5+ processors and offers relatively flat access to the node memory from each of the 16 cores in the node as a result of the round-robin memory page placement policy. Each Power5+ processor is capable of simultaneous multithreading (SMT). The third parallel system, an SGI Altix ICE 8200EX, is part of the newly expanded Pleiades cluster [24]. Pleiades comprises nodes based on three types of Intel Xeon processors: quad-core E5472 (Harpertown), quad-core X5570 (Nehalem-EP), and six-core X5670 (Westmere). All the nodes are interconnected with InfiniBand in a hypercube topology. In this study, we only used the Nehalem-based nodes. These nodes use two quad-core Intel X5570 (Nehalem-EP) processors. Each Nehalem processor contains four cores with hyperthreading (HT, similar to SMT) capability, which enables two threads per core. An on-chip memory controller connects directly to local DDR3 memory, which improves memory throughput substantially compared to the previous generations of Intel processors. However, accessing memory within a Nehalem-based node is non-uniform. Both SGI Altix systems have the first-touch memory placement policy as the default.

On the software side, both MPI and OpenMP are available on the SGI Altix 4700 through the SGI MPT library and the compiler support. For the two cluster systems, MPI, supported by the SGI MPT or the IBM POE library, is available for communication between nodes, and either MPI or OpenMP can be used within each node (see Table 4). For all of our experiments in the study, we fully populated the core resources on each node. For instance, to run an application on the SGI Altix ICE system with 32 MPI processes and 4 OpenMP threads per MPI process, we allocate 16 nodes for the job and assign 2 MPI processes on each node with a total of 8 threads per node. To ensure consistent performance results, we applied the process/thread binding tools available on each parallel system to bind processes and threads to processor cores.

4.2. Multi-zone NAS Parallel Benchmarks

As mentioned in Section 3.1, because of the limited number of zones in LU-MZ, which limits the number of MPI processes, we only discuss the performance of BT-MZ and SP-MZ in this section. Before diving into performance details, we first examine the memory usage of the benchmark applications for different process and thread combinations.

The measured memory usage for SP-MZ Class C on the Altix ICE is shown in Fig. 3: per process as a function of MPI processes on the left panel (a) and per core as a function of OpenMP threads on the right panel (b). The total number of processing cores used in each experiment equals to the number of MPI processes times the number of OpenMP threads per MPI process. The results were measured from the high-water-mark memory usage at the end of each run with a correction for the use of shared memory segment among processes on a node. For comparison, the actual memory used by the data arrays in the application is also included in the figure.

Within an Altix ICE node (up to 8 cores, Fig. 3(a)), the memory usage roughly reflects the need of the application data. As the number of MPI processes increases, the data size per process decreases, but the actual memory usage reduces only to a point (around 128 cores), then increases almost quadratically. The increased memory is mainly due to the increased MPI buffers allocated for collective communication between nodes in the underlying MPI library (the SGI MPT in this case) and is dependent on the number of nodes used. In the hybrid mode, for a given number of cores, we see substantial reduction in memory usage, especially in large core counts (see Fig. 3(b)). For instance for 256 cores, the 256×1 run (similar to pure

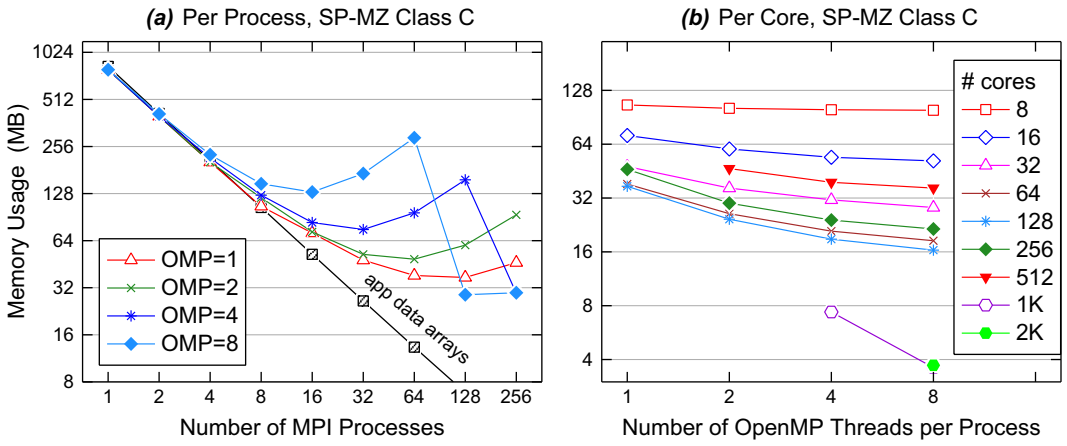

Fig. 3. SP-MZ Class C memory usage on the Altix ICE-Nehalem.

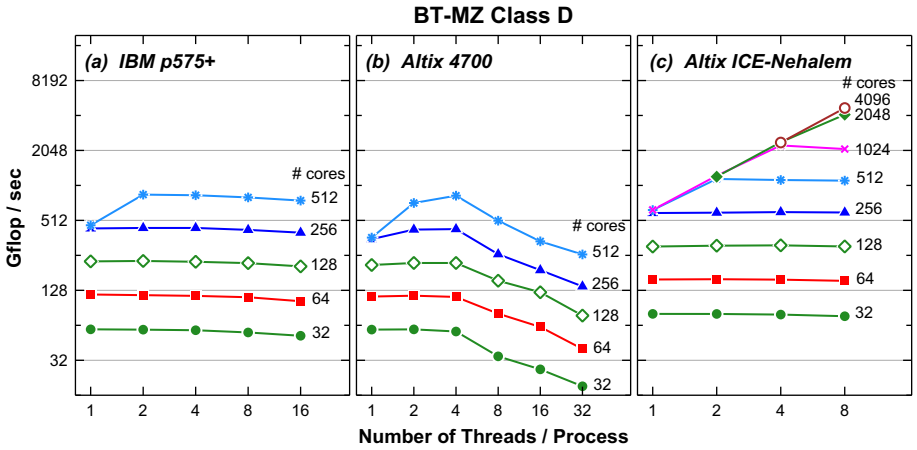

Fig. 4. BT-MZ performance (Gflop/s) on three parallel systems.

MPI) uses 11.9 GB (=256 ∗ 46.3 MB) of memory, while the 32 × 8 (hybrid) run uses only 5.5 GB (=32 ∗ 8 ∗ 21.5 MB) of memory. This large memory saving could be very important for future multi-core systems where less per-core memory is expected. The figure also shows a sudden drop in memory usage beyond 64 nodes (512 cores). This corresponds to the MPT, SGI's MPI library, switching to a less optimal communication scheme, which allocates less space for message buffers and could cause performance degradation for communication in certain applications (such as the CART3D case described in Section 2). Although using a different data array size, the memory profile of BT-MZ (not shown here) is very similar to that of SP-MZ. The trends in memory usage for Class D (also not shown here) are very similar to Class C.

Figs. 4 and 5 show the performance (Gflop/s, higher is better) of BT-MZ and SP-MZ Class D on three parallel systems. Each curve corresponds to a given number of cores used, while varying the number of OpenMP threads. For a given core count, going from left to right, the number of MPI processes decreases while the number of OpenMP threads per MPI process increases from 1 to 16 on the IBM p575+, 1 to 32 on the Altix 4700, and 1 to 8 on the Altix ICE-Nehalem. The Class D data set has a total of 1024 zones. Thus, MPI can only be used up to 1024 processes and beyond that, the hybrid approach is needed. For BT-MZ, MPI scales nicely up to 256 processes on all three systems, but beyond that there is no further improvement due to load imbalance as a result of uneven size zones in the benchmark (see Fig. 4). However, further performance gains can be obtained by using OpenMP threads to improve load balance at large core counts. For a given core count, BT-MZ shows relatively flat performance from different process-thread combinations as long as the load for MPI processes is balanced (up to 256 processes) on both IBM p575+ and Altix ICE-Nehalem. However, on the Altix 4700 the performance degrades quickly beyond 4 OpenMP threads when NUMA memory blades are involved. As discussed in Section 4.1 each of the Altix 4700 blade contains 4 Itanium2 cores sharing the same local memory board through a bus. Accessing memory in another blade for more than 4 cores has to go through the SGI NUMAlink, which has longer latency. The performance as shown in Fig. 4(b) is indicative of long memory latency associated with accessing remote memory on the system.

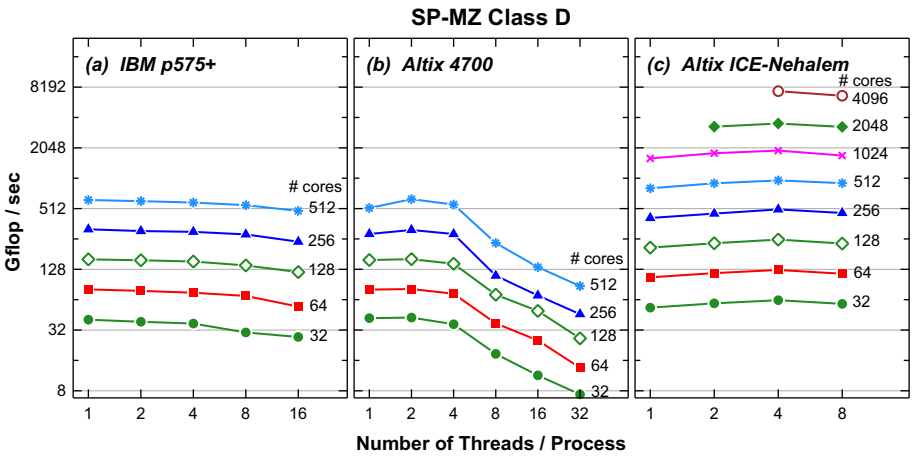

Fig. 5. SP-MZ performance (Gflop/s) on three parallel systems.

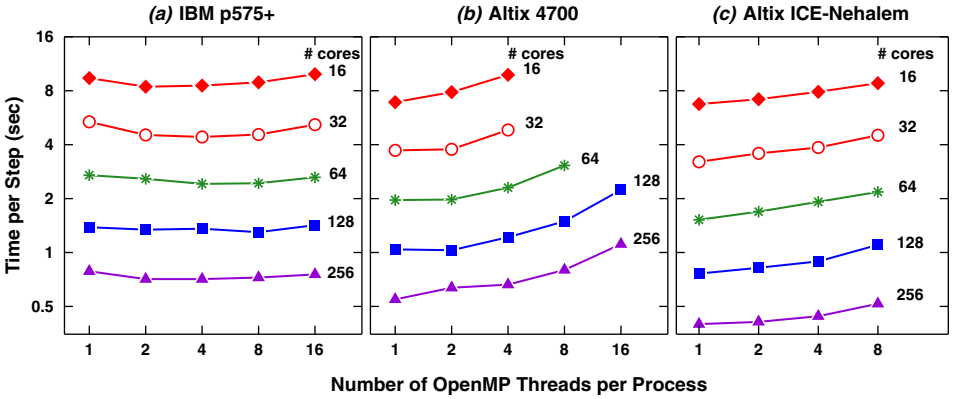

Fig. 6. OVERFLOW2 performance on three parallel systems. Numbers in the graphs indicate the number of cores used.

SP-MZ scales very well up to 1024 MPI processes on the Altix ICE-Nehalem (see Fig. 5(c)). However, there are performance differences from using different numbers of OpenMP threads for a given core count. The best performance is obtained when the number of OpenMP threads per process is 4. This phenomenon needs some explanation. First, using more OpenMP threads reduces the corresponding number of MPI processes. As a result, the total number of messages is reduced while the message sizes are increased, which leads to better communication performance and improves the overall performance. Secondly, the OpenMP threads spawned by the same MPI process work on zones in SP-MZ assigned to the MPI process together, one zone at a time. Using more OpenMP threads effectively increases the total cache size for the same amount of data and improves the performance (assuming one OpenMP thread assigned to one core). However, as the number of OpenMP threads increases, the overhead associated with OpenMP constructs and remote memory accesses across sockets in a node increases. From a combination of these factors, we see a relative performance increase up to 4 threads after which we see a performance drop for 8 threads.

On the IBM p575+ for a given core count, we see persistent performance drop for SP-MZ as the number of OpenMP threads increases. We attribute this result to the fact that on a p575+ node the memory page placement policy is round-robin (not first-touch) but the OpenMP parallelization for SP-MZ assumes a first-touch memory placement. Such a mismatch causes longer latency in data access from different OpenMP threads. As a result we do not see an effective increase in data cache as observed on the Altix ICE-Nehalem. In contrast, the performance of SP-MZ on the Altix 4700 is relatively flat up to 4 OpenMP threads and then drops quickly after that, which indicates the severe NUMA effect on the system.

4.3. Overflow

In our experiments, we used OVERFLOW 2.0aa on a realistic wing/body/nacelle/pylon configuration with 23 zones and 35.9 million mesh points. Fig. 6 shows the performance of the hybrid code in seconds per time step (lower is better) measured on the IBM p575+, the Altix 4700, and the Altix ICE-Nehalem using different process-thread combinations. The number

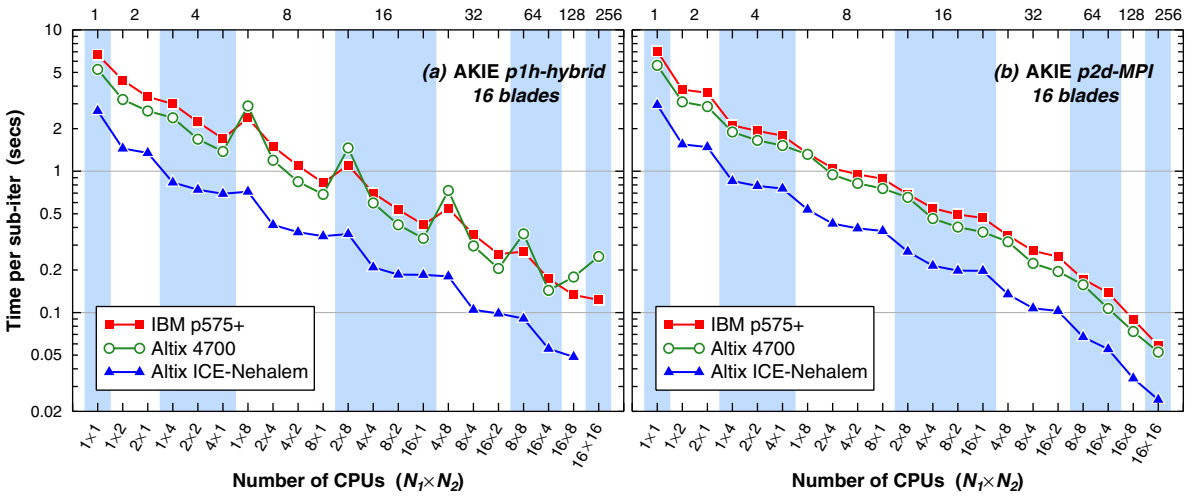

Fig. 7. Timings of the AKIE (a) hybrid and (b) MPI codes on three parallel systems.

of OpenMP threads per MPI process varies from 1 to 8 on the Altix ICE and up to 16 on the other two systems. The data points for the pure MPI version on all three systems are very close to those for the hybrid version with 1 OpenMP thread as plotted in the graph. On the IBM p575+, the hybrid version with multiple OpenMP threads outperformed the pure MPI version. The best results were observed with 4 OpenMP threads. Recall from Section 3.2 that as the number of MPI processes increases, OVERFLOW does more mesh splitting, resulting in more mesh points. As the number of OpenMP threads increases for a fixed total number of cores, fewer MPI processes are used and faster execution time might be expected since the total number of mesh points decreases. At the same time, overhead associated with OpenMP parallelization increases, which has a negative impact on performance. The balance point from the two effects depends on the application and on the system characteristics.

The benefit observed on the IBM system from the hybrid version is not seen on the two Altix systems. Instead, we observe a monotonic increase in time as the number of OpenMP threads increases. This increase in runtime is predominantly caused by the increase in remote memory accesses from OpenMP on the ccNUMA architectures, which overshadows any reduction in runtime from a smaller amount of work. The key to improving OpenMP performance on a NUMA-based system is to improve data locality and reduce remote memory access. In contrast, no obvious NUMA effect was observed on the IBM p575+, which can be attributed to the round-robin memory page placement policy on the system.

4.4. Akie

We used a 16-blade case to compare the performance of hybrid and MPI versions of the AKIE code on the three parallel systems. Per-iteration timings of runs from various process-thread combinations are shown in Fig. 7 (lower is better). In the $N_1 \times N_2$ notation, N_1 indicates the number of MPI processes at the first level, and N_2 indicates the number of OpenMP threads (the hybrid version) or MPI processes (the MPI version) at the second level for each of the first level MPI processes. The total number of cores used, $N_1 \times N_2$, increases from 1 to 256. The value of N_1 is limited by the number of blades, which is 16 in the current case. The value of N_2 is mostly constrained by the partitioned dimension of a given problem. For the hybrid case, the number of OpenMP threads N_2 is also limited by the size of an SMP node (16 on the IBM p575+, 8 on the Altix ICE, 2048 on the Altix 4700). For instance, we could not run the 16×16 hybrid version on the Altix ICE, but there is no such a limitation for the MPI version since N_2 MPI processes can be spread across multiple nodes if needed. Overall, we observe a similar performance for the IBM p575+ and the Altix 4700. The Nehalem-based SGI Altix ICE produces 2–3 times better performance than the other two systems. First, this is partly due to faster processor speed in the system. Second, the use of the 32-bit numerical algorithm in the application has larger performance boost (from the 64-bit numerical algorithm) on the Nehalem processor than on the Power5+ and Itanium2 processors used in the other two systems.

For a given number of core counts, the best performance is achieved with the smallest possible N_2 for both hybrid and MPI versions. As N_2 increases, the runtime also increases. The hybrid version shows a larger jump at $\geqslant 8$ OpenMP threads, especially on the SGI Altix 4700 system, while the MPI version shows a much smoother trend. To get a more quantitative comparison, Fig. 8 plots the ratio of timings of the MPI version with those of the hybrid version for various $N_1 \times N_2$ combinations. A value larger than 1 indicates better performance of the hybrid version. As shown in the figure, at $N_2 = 1$ the hybrid version performs about 10% better than the 2D MPI version, indicating less overhead in the hybrid version. The better hybrid performance persists on the Altix ICE-Nehalem system up to 4 OpenMP threads (number of cores in a socket), in contrast to other two systems. This is attributed to the improved memory latency in the Nehalem system. However, at 8 OpenMP threads the hybrid performance drops to about 80% of the MPI performance on the Altix ICE and about 42% on the Altix

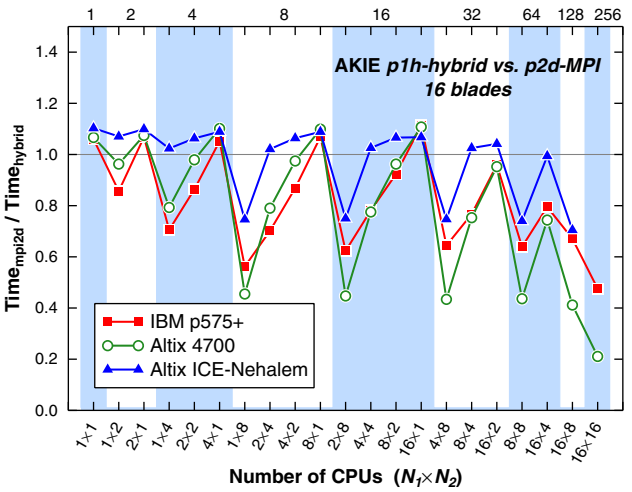

Fig. 8. Timing comparison of the AKIE hybrid and MPI codes.

4700, indicating large impact of NUMA memory on OpenMP performance on both of these systems. The IBM p575+, which has relatively flat memory nodes, is somewhere in between. At 16 OpenMP threads, the performance degradation is even larger. It points out the importance of improved memory latency for better OpenMP performance and, thus, better hybrid performance.

4.5. Summary of the hybrid approach

We observed a number of benefits from the hybrid MPI + OpenMP approach compared to the pure MPI approach for real-world applications. The hybrid approach can reduce the memory footprint and overhead associated with MPI calls and buffers, improve load balance, and maintain numerical convergence and stability. Another benefit that has not been discussed much is the potential to reduce the parallel code development cycle for those applications that demonstrate exploitable multi-level parallelism, as illustrated by the multi-zone applications. We also observed some limiting factors with the hybrid approach. These are mainly associated with limitations of the OpenMP model, including performance sensitivity to memory latency and no language mechanisms to enforce data affinity to mitigate the NUMA effect. The number of OpenMP threads is limited by the number of cores available on a hardware SMP node. In the next section, we will discuss extensions to the OpenMP programming model to improve data locality in the model.

Lastly, the interaction between MPI processes and OpenMP threads is not well defined in either model, which may have other consequences on performance. As pointed out by previous studies e.g., [26], properly binding processes and threads to CPUs (or *pinning*) can improve performance in some cases. Although not presented here, we have found that pinning has larger performance impact on the NUMA-based systems, such as the SGI Altix 4700, and in the presence of simultaneous multithreading (SMT).

5. Extensions to OpenMP for locality

As we observed in the previous sections, one of the keys to improving the hybrid performance is to improve the OpenMP performance. The fundamental difficulty is that the current OpenMP model is still based on a flat memory assumption that does not match with the modern architectures, which often exhibit more hierarchical and complex memory subsystems. Managing the data layout and maintaining the data and task affinity are essential factors to achieving scalable performance on such systems. In this section, we introduce the concept of locality into OpenMP as language extensions without sacrificing the simplicity of the OpenMP model. We describe the proposed syntax to specify location and data layout that allows the user to control task-data affinity. More detailed discussion of this work can be found in [12]. This section presumes that the reader is familiar with OpenMP; for further details please refer to [22].

5.1. Introduction of location

The key concept is the introduction of *location* into OpenMP as a virtual layer for expressing locality and task-data affinity in a program. The location concept is similar to the notion of *Place* in X10 [7] and *Locale* in Chapel [8]. We define location in OpenMP as follows.

Location: *a logical execution environment that contains data and OpenMP implicit and explicit tasks.*

A location includes a virtual task execution engine and its associated memory. At runtime, a location may be mapped onto hardware resources, such as a shared-memory node, a cache-coherent NUMA node, or a distributed shared-memory (DSM) node. Tasks assigned to a location may be executed concurrently following the usual OpenMP rules. We expect that tasks running on a location may have a faster access to local memory than to memory on other locations.

We introduce a parameter `NLOCS` as a pre-defined variable for the number of locations available in an OpenMP program. It can be defined at runtime via the environment variable `OMP_NUM_LOCS` or at the compile time through compiler flags. Each location is uniquely identified with a number `MYLOC` in the range of `[0:NLOCS-1]`.

5.1.1. The `location` clause

The "`location (m[:n])`" clause can be used with the `parallel` and `task` directives to assign implicit and explicit OpenMP tasks to a set of locations. *m* and *n* are integers defining the range of locations. A single value *m* indicates a single location. To support some level of fault tolerance for the case where the value of *m* or *n* is out of the available location boundary, the actually assigned location is determined by (*m* or *n* modulo `NLOCS`).

Mapping of threads (or implicit tasks) from a parallel region to locations is done in a (default) block distribution fashion. For instance, if we have 16 threads and 4 locations as shown in the following example,

```
#pragma omp parallel location (0:3) num_threads (16)
```

threads 0–3 will be assigned to location 0, threads 4–7 to location 1, etc. Other mapping policies are possible, but we only discuss the simplest one in this proposal.

5.1.2. Location inheritance rule

The location inheritance rule for those parallel regions and tasks without the "`location`" clause is hierarchical, that is, it is inherited from the parent. At the beginning of a program execution, the default location association is all locations. Thus, when there is no location associated with a top-level parallel region, implicit tasks from the parallel region will be mapped across all locations in a block distribution fashion. When encountering a nested parallel region without a specified location, the nested parallel region will be executed in the same location as its parent. This design choice allows natural task-location affinity and, thus, task-data affinity to achieve good data locality in the presence of nested parallelism.

If a task has been assigned to a location, all of its child tasks will be running on the same location if no other `location` clause is specified. On the contrary, if a `location` clause is specified for one of its children, the child task will be executed on the specified location.

5.1.3. Mapping locations to hardware

The mapping of logical locations to underneath hardware is not specified at the language level and, rather, is left to the runtime implementation or an external deployment tool. The runtime implementation may assign locations to computing nodes based on the query result of the machine topology and bind threads to locations if required. Alternatively, an external tool (perhaps named `omprun`) will allow users to describe the machine architecture information and configure how to map locations onto the underlying hardware. It will interact with OS/job schedulers to allocate the required resources for the OpenMP programs.

5.2. Data layout

The current OpenMP specification does not specify how and where data gets allocated on the target hardware. It usually relies on the default memory management policy of the underlying operating system. The most common one is the *first touch* policy, that is, whichever CPU touches a data page first will allocate the memory page on the node containing the CPU. With the concept of location, we introduce in OpenMP a way to express data layout. The goal is to allow users to control and manage shared data among locations. The data layout, currently, is static during the lifetime of the program. That is, we are not considering data redistribution or migration at this time but may extend the proposal to support dynamicity later on. For data without a user-specified data layout, the system would use the default memory management policy. The concept of data distribution for shared memory programming was first included in the SGI's MIPSpro compiler [29].

5.2.1. The `distribute` directive

We use a data distribution syntax to express data layout as a directive in OpenMP as follows.

```
#pragma omp distribute (dist-type-list: variable-list) [location (m:n)]
```

"`dist-type-list`" is a comma-separated list of distribution types for the corresponding array dimensions in the `variable-list`. Variables in the `variable-list` should have the same dimensions that match with the number of distribution types listed. Possible distribution types include "`BLOCK`" for a block distribution across a list of locations given

in the `location` clause, and "∗" for non-distributed dimension. If the `location` clause is not present, it means all locations are to be used for the distribution. Other types of data distribution are also possible, but we do not discuss them here for the sake of simplicity.

The distributed data still keeps its global address and is accessed in the same way as other shared data in the program. With distributed data, the user can control and manage shared data to improve data locality in OpenMP programs. The following example shows how the first dimension of array *A* gets distributed across all locations.

```
double A[N][N];
#pragma omp distribute (BLOCK,∗: A) location (0:NLOCS-1).
```

5.2.2. The `onloc` clause

To achieve greater control of task-data affinity, we can map implicit and explicit tasks in OpenMP to locations based on either an explicit location number or the placement of distributed data. For the latter case, we introduce the "`onloc (variable)`" clause to assign a task to a location based on where the data resides. Only distributed variables are meaningful in the `onloc` clause. The variable can be either an entire array (for the `parallel` construct) or an array element (for the `task` construct). In the following example,

```
#pragma omp task onloc (A[i])
  foo (A[i]);
```

the location for the task execution is determined by the location of "`A[i]`". When the `onloc` clause applies to a parallel region, it implies that the list of locations for the region is derived from the list associated with the distributed variable, as shown in the following example,

```
#pragma omp parallel onloc (A)
  {...}
```

Mapping of implicit tasks to locations for this case then follows the same rule as discussed for the `location` clause. Compared to `location`, the `onloc` approach can achieve better control of task-data affinity.

The following restriction applies to the `location` and `onloc` clauses: At most one `onloc` or `location` clause can appear on the `parallel` or `task` directive. If a non-distributed variable is used in the `onloc` clause, it will be treated as if no such a clause is specified and an implementation may print warning messages.

5.3. Implementation and experiments

We are currently developing the proposed location feature in the OpenUH compiler [23]. The OpenUH compiler is a branch of Open64 compiler and is used as a research infrastructure for OpenMP compiler and tools research. Besides the translation of OpenMP constructs into the appropriate runtime calls, the focus is on the development of OpenMP runtime

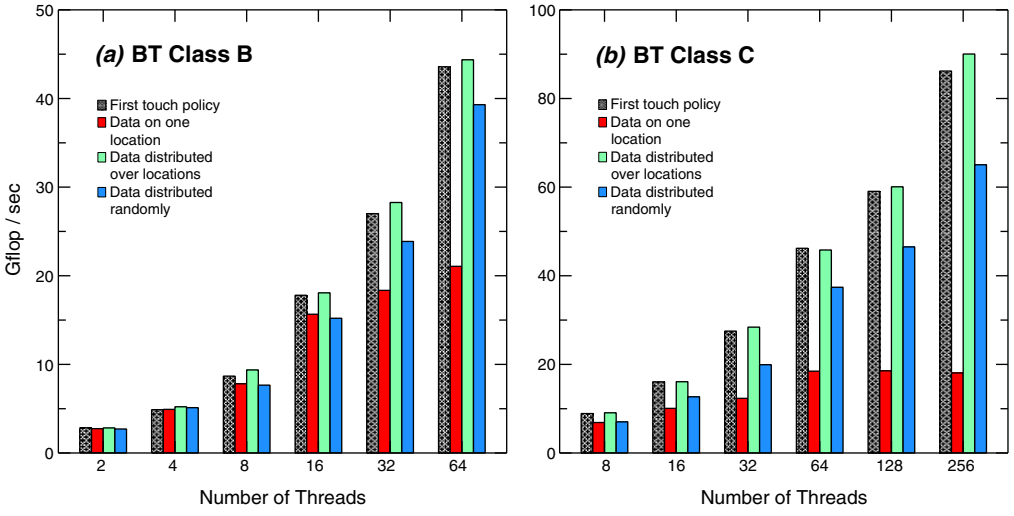

Fig. 9. BT performance (Gflop/s) from different data layouts.

infrastructure to support location and data layout management. Our current approach relies on the `libnuma` library [18] available under Linux for managing distributed resources. For detailed description, please refer to [12].

Since our implementation is not yet complete, we are not able to demonstrate the performance gains using the proposed features. However, as an experiment, we revised the BT benchmark from the NPB suite manually to illustrate the importance of different data layouts on performance. Results collected on the SGI Altix 4700 system are shown in Fig. 9 for BT Class B and Class C, higher Gflop/s indicating better performance. The four different columns for each thread count show the performance of four different data layout policies: first touch, all data on one location, layout based on data access pattern, and random placement. From this experiment, we observe significant performance impact from different data layouts on the NUMA system, especially for larger data sets. In general, the layout based on data access pattern (shown in the third column) produces the best performance, followed by first touch (first column). The single location and random placement performed worse than the other two. Although the case with data distributed over locations is similar to the case using the first touch policy, we believe that it can be improved further by optimizing data layout with the data access pattern, which would be possible after our compiler implementation is completed.

Although the goal of the work is to introduce a minimal set of features via compiler directives to achieve desired performance gain, it is anticipated that some amount of code transformations could be required (either by compiler or by hand) in order to achieve high performance. For instance, the static data layout in the current proposal would not work well for applications that alter data access patterns in different computing phases. One possible solution would be to introduce the concept of *data redistribution* for different phases. However, the benefit of data redistribution could be overshadowed by the associated cost. An alternative approach would involve the use of duplicated datasets with different data layouts suitable for each computing phase. The obvious drawback of this approach is the increased memory footprint. For examining the effectiveness of the proposed locality extensions, we would need more experiments with real applications.

6. Related work

MPI and OpenMP are the two mostly studied parallel programming models over the years. There are many publications on the two programming models and also on the hybrid approach. A good summary of the hybrid MPI + OpenMP parallel programming can be found in a tutorial given by Rabenseifner et al. [25]. The related publication [26] includes the performance study of the NPB-MZ hybrid codes on an AMD Opteron-based cluster and points out some of the keys for achieving good scalability. Shan et al. [28] described an analysis of performance effects from different programming choices for NPB and NPB-MZ on Cray XT5 platforms. They presented the benefit of the hybrid model on memory usage. Our current work extends the study to the Intel processor-based clusters in detail and provides better understanding of the effect from the underlying MPI library implementation.

Our previous work [27] compared the performance of the hybrid OVERFLOW code on three parallel systems. The current work gives a more comprehensive analysis of the hybrid approach based on numerical accuracy and convergence in addition to presenting the performance of the code on the current multi-core architectures. We also utilize the AKIE code to present a more *apples-to-apples* comparison of the hybrid and the pure MPI approaches.

SGI has included the concept of data distribution for shared memory programming in its MIPSpro compiler [29]. Two earlier proposals by Benkner [3] and Bircsak [5] made another attempt to extend OpenMP for ccNUMA machines. These efforts were based on the data distribution concepts from High Performance Fortran (HPF). Our extensions to OpenMP are built on the concept of location, which follows similar concepts in the modern HPCS languages and provides a virtual layer for expressing the NUMA effect. This design choice is intended to preserve the simplicity of the OpenMP programming.

7. Conclusions

In summary, we have presented case studies of the hybrid MPI + OpenMP programming approach applied to two pseudo-application benchmarks and two real-world applications, and demonstrated benefits of the hybrid approach for performance and resource usage on three multi-core based parallel systems. As the current high performance computing systems are pushing toward petascale and exascale, per-core resources, such as memory, are expected to become smaller. Thus, the reduction on the memory usage and overhead associated with MPI calls and buffers resulting from the hybrid approach will become more important. Our studies have also shown the usefulness of the hybrid approach for improving load balance and numerical convergence. Although modern parallel languages such as the HPCS languages are maturing, the more traditional MPI + OpenMP approach is still viable for programming on a cluster of SMPs.

There are two primary limitations that impede a wider adoption of hybrid programming: no well-defined interaction between MPI processes and OpenMP threads, and limited performance of OpenMP, especially on ccNUMA architectures. There is an on-going effort in the MPI community to improve the interface with threads in MPI 3 [10]. In this paper, we have described our approach to extending OpenMP with the concept of location to improve the performance of the resultant programs on ccNUMA systems. We also presented the syntax of language extensions to express task-data affinity. For future work, it would be useful to conduct more detailed studies on the performance sensitivity of applications to memory latency (such as via hardware performance counters) and MPI parameters. We would like to refine the location concept design and verify the effectiveness of the extensions by studying the performance of test cases and benchmarks, for example, to

investigate different thread-to-location mappings on performance. Another area of interest is to extend the location concept to platforms other than NUMA SMPs.

Acknowledgments

The authors thank Jahed Djomehri for providing the CART3D results presented here, Johnny Chang and Robert Hood for their valuable discussion and comments, the NAS HECC staff for their support in conducting these performance measurements on the NAS systems, and anonymous reviewers for their valuable comments on the manuscript. The work was partially supported by the National Science Foundation grant CCF-0702775 with the University of Houston.

References

[1] E. Allen, D. Chase, C. Flood, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele, Project Fortress: A Multicore Language for Multicore Processors. Linux Magazine, 2007.
[2] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, M. Yarrow, The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995.
[3] S. Benkner, T. Brandes, Exploiting data locality on scalable shared memory machines with data parallel programs, in: Proceedings of the Euro-Par 2000 Parallel Processing, Munich, Germany, 2000, pp. 647–657.
[4] M.J. Berger, M.J. Aftosmis, D.D. Marshall, S.M. Murman, Performance of a new CFD flow solver using a hybrid programming paradigm, Journal of Parallel and Distributed Computing 65 (4) (2005) 414–423.
[5] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, C.D. Offner, Extending OpenMP for NUMA machines, in: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, Dallas, TX, 2000.
[6] B.L. Chamberlain, D. Callahan, H.P. Zima, Parallel programmability and the chapel language, International Journal of High Performance Computing Applications 21 (3) (2007) 291–312.
[7] P. Charles, C. Donawa, K. Ebcioglu, C. Grotho, A. Kielstra, V. Saraswat, V. Sarkar, C.V. Praun, X10: an object-oriented approach to non-uniform cluster computing, in: Proceedings of the 20th ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, ACM SIGPLAN, 2005, pp. 519–538.
[8] R. Diaconescu, H. Zima, An approach to data distributions in chapel, Int. J. High Perform. Comput. Appl. 21 (3) (2007) 313–335.
[9] J. Djomehri, D. Jespersen, J. Taft, H. Jin, R. Hood, P. Mehrotra, Performance of CFD applications on NASA supercomputers, in Proceedings of the International Conference on Parallel Computational Fluid Dynamics, 2009, pp. 240–244.
[10] R.L. Graham, G. Bosilca, MPI forum: preview of the MPI 3 standard. SC09 Birds-of-Feather Session, 2009. Available from: <http://www.open-mpi.org/papers/sc-2009/MPI_Forum_SC09_BOF-2up.pdf>.
[11] C. Hah, A.J. Wennerstrom, Three-dimensional flow fields inside a transonic compressor with swept blades, ASME Journal of Turbomachinery 113 (1) (1991) 241–251.
[12] L. Huang, H. Jin, Liqi Yi, B. Chapman, Enabling locality-aware computations in OpenMP, Scientific Programming 18 (3-4) (2010) 169–181 (special issue).
[13] L. Hochstein, V.R. Basili, The ASC-alliance projects: a case study of large-scale parallel scientific code development, Computer 41 (3) (2008) 50–58.
[14] L. Hochstein, F. Shull, L.B. Reid, The role of MPI in development time: a case study, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, Austin, Texas, 2008.
[15] H. Jin, R.F. Van der Wijgnaart, Performance characteristics of the multi-zone NAS parallel benchmarks, Journal of Parallel and Distributed Computing 66 (2006) 674–685.
[16] H. Jin, R. Hood, P. Mehotra, A practical study of UPC with the NAS parallel benchmarks, in: Proceedings of the 3rd Conference on Partitioned Global Address Space (PGAS) Programming Models, Ashburn, VA, 2009.
[17] D. Kaushik, S. Balay, D. Keyes, B. Smith, Understanding the performance of hybrid MPI/ OpenMP programming model for implicit CFD codes, in: Proceedings of the 21st International Conference on Parallel Computational Fluid Dynamics, Moffett Field, CA, USA, May 18–22, 2009, pp. 174–177.
[18] A. Kleen, An NUMA API for Linux, SUSE Labs, 2004. Available from: <http://www.halobates.de/numaapi3.pdf>.
[19] G. Krawezik, F. Cappello, Performance comparison of MPI and three OpenMP programming styles on shared memory Multiprocessors, in: Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures, San Diego, CA, 2003, pp. 118–127.
[20] R.H. Nichols, R.W. Tramel, P.G. Buning, Solver and turbulence model upgrades to OVERFLOW 2 for unsteady and high-speed applications, in: Proceedings of the 24th Applied Aerodynamics Conference, volume AIAA-2006-2824, 2006.
[21] R. Numrich, J. Reid, Co-array fortran for parallel programming, In ACM Fortran Forum 17 (1998) 1–31.
[22] OpenMP Architecture Review Board, OpenMP Application Program Interface 3.0, 2008. Available from: <http://www.openmp.org/>.
[23] The OpenUH compiler project. Available from:<http://www.cs.uh.edu/openuh>.
[24] Pleiades Hardware. Available from: <http://www.nas.nasa.gov/Resources/Systems/pleiades.html>.
[25] R. Rabenseifner, G. Hager, G. Jost, Tutorial on hybrid MPI and OpenMP parallel programing, in: Supercomputing Conference 2009 (SC09), Portland, OR, 2009.
[26] R. Rabenseifner, G. Hager, G. Jost, Hybrid MPI/OpenMP parallel programing on clusters of multi-core SMP nodes, in: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP2009), Weimar, Germany, 2009, pp. 427–436.
[27] S. Saini, D. Talcott, D. Jespersen, J. Djomehri, H. Jin, R. Biswas, Scientific application-based performance comparison of SGI Altix 4700, IBM Power5+, and SGI ICE 8200 supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008.
[28] H. Shan, H. Jin, K. Fuerlinger, A. Koniges, N. Wright, Analyzing the Performance Effects of Programming Models and Memory Usage on Cray XT5 Platforms, Cray User Group (CUG) meeting, Edingburgh, United Kingdom, 2010.
[29] Silicon Graphics, Inc., MIPSpro (TM) Power Fortran 77 Programmer's Guide, Document 007-2361, SGI, 1999.
[30] The Top500 List of Supercomputer Sites. Available from: <http://www.top500.org/lists/>.
[31] The UPC Consortium, UPC Language Specification (V1.2), 2005. Available from: <http://www.upc.gwu.edu/documentation.html>.
[32] R.F. Van der Wijgnaart, H. Jin, The NAS Parallel Benchmarks, Multi-Zone Versions. Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.
[33] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: a high-performance java dialect, in: Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing, 1998.