1 Introduction

Since its introduction in 1997, OpenMP has become the de facto standard for shared memory parallel programming. The notable advantages of the model are its global view of memory space that simplifies programming development and its incremental approach toward parallelization. However, it is a big challenge to scale OpenMP codes to tens or hundreds of processors. One of the difficulties is a result of limited parallelism that can be exploited on a single level of loop nest. Although the current standard [1] allows one to use nested OpenMP parallel regions, the performance is not very satisfactory. One of the known issues with nested OpenMP is its lack of support for thread team reuse at the nesting level, which affects the overall application performance and will be more profound on multi-core, multi-chip architectures. There is no guarantee that the same OS threads will be used at each invocation of parallel regions although many OS and compilers have provided support for thread affinity at a single level. To remedy this deficiency, the NANOS compiler team [2] has introduced the `GROUPS` clause to the outer parallel region to specify a thread group composition prior to the start of nested parallel regions, and Zhang et al. [3] proposed extensions for thread mapping and grouping.

Chapman and collaborators [4] proposed the *Subteam* concept to improve work distribution by introducing subteams of threads within a single level of thread team, as an alternative for nested OpenMP. Conceptually, a subteam is similar to a process subgroup in the MPI context. The user has control over how threads are subdivided in order to suit application needs. The subteam proposal introduced an `subthreads` clause to a work-sharing directive so that the work, including the implicit barrier at the end of the construct, will be performed among the subset of threads within the team.

One of the prominent extensions to the current OpenMP is the *Workqueuing* (or Taskq) model first introduced by Shah et al. [5] and implemented in the Intel C++ compiler [6]. It was designed to work with recursive algorithms and cases where work units can only be determined dynamically. Because of its dynamic nature, Taskq can also be used effectively in an unbalanced workload environment. The Taskq model will be included in the coming OpenMP 3.0 release [7,8]. Although the final *tasking* directive in OpenMP 3.0 will not be the same as the original Intel Taskq proposal, it should still be quite intuitive to understand what potential the more dynamic approach can offer to applications.

In this study, we will compare different OpenMP approaches for the parallelization of a multi-zone application benchmark and report performance results from three large-scale parallel machines. In Sect. 2, we briefly discuss the application under consideration. The different implementations of our benchmark code are described in Sect. 3 and the machine description and performance results are presented in Sect. 4. We conclude our study in Sect. 5, where we also elaborate on future work.

2 Multi-Zone Application Benchmark

The multi-zone application benchmarks were developed [9,10] as an extension to the original NAS Parallel Benchmarks (NPBs) [11]. These benchmarks involve solving

the application benchmarks BT, SP, and LU on collections of loosely coupled discretization meshes (or zones). The solutions on the meshes are updated independently, but after each time step they exchange boundary value information. This strategy, which is common among many production structured-mesh flow solver codes, provides relatively easy to exploit coarse-grain parallelism between zones. Since the individual application benchmark also allows fine-grain parallelism within each zone, this NPB extension, named NPB Multi-Zone (NPB-MZ), is a good candidate for testing hybrid and multi-level parallelization tools and strategies.

NPB-MZ contains three application benchmarks: BT-MZ, SP-MZ, and LU-MZ, with problem sizes defined from Class S to Class F. The difference between classes comes from how the number of zones and the size of each zone are defined in each benchmark. We focus our study on the BT-MZ benchmark because it was designed to have uneven-sized zones, which allows us to test various load balancing strategies. For example, the Class B problem has 64 zones with sizes ranging from 3 K to 60 K mesh points. Previously, the hybrid MPI+OpenMP [4] and nested OpenMP [2] programming models have been used to exploit parallelism in NPB-MZ beyond a single level. These approaches will be briefly described in the next section.

3 Benchmark Implementations

In this section, we describe five approaches of using OpenMP and its extension to implement the BT-MZ benchmark. Three of the approaches exploit multi-level parallelism and the other two are concerned with balancing workload dynamically.

3.1 Hybrid MPI+OpenMP

The hybrid MPI+OpenMP implementation exploits two levels of parallelism in the multi-zone benchmark in which OpenMP is applied for fine grained intra-zone parallelization and MPI is used for coarse grained inter-zone parallelization. Load balancing in BT-MZ is based on a bin-packing algorithm with an additional adjustment from OpenMP threads [9]. In this strategy, multiple zones are clustered into zone groups among which the computational workload is evenly distributed. Each zone group is then uniquely assigned to each MPI process for parallel execution. The procedure involves sorting zones by size in descending order and bin-packing into zone groups. Exchanging boundary data within each time step requires MPI many-to-many communication. The hybrid version is fully described in Ref. [4] and is part of the standard NPB distribution. We will use the hybrid version as the baseline for comparison with other OpenMP implementations.

3.2 Nested OpenMP

The nested OpenMP implementation is based on the two-level approach of the hybrid version except that OpenMP is used for both levels of parallelism. The inner level parallelization for loop parallelism within each zone is essentially the same as that of the

```
!$omp parallel private(myid,...)
 myid = omp_get_thread_num()
 call map_zones(myid,..,nthreads,&
      & num_proc_zones,proc_zone_id)
 do step=1,niter
   call exch_qbc(u,...,nthreads)
   do iz = 1, num_proc_zones
     zone = proc_zone_id(iz)
     call adi(u(zone),..,nthreads)
   end do
 end do
!$omp end parallel

 subroutine adi(u,..,nthreads)
!$omp parallel do &
!$omp& num_threads(nthreads)
 do k=2,nz-1
   solve for u in the current zone
 end do
```

```
!$omp parallel private(myid,...)
 myid = omp_get_thread_num()
 call map_zones(myid,..,mytid,&
      & proc_thread_team)
 t1 = proc_thread_team(1,mytid)
 t2 = proc_thread_team(2,mytid)
 do step=1,niter
   call exch_qbc(u,...,t1,t2)
   do iz = 1, num_proc_zones
     zone = proc_zone_id(iz)
     call adi(u(zone),..,t1,t2)
   end do
 end do
!$omp end parallel

 subroutine adi(u,..,t1,t2)
!$omp do onthreads(t1:t2:1)
 do k=2,nz-1
   solve for u in the current zone
 end do
```

Fig. 1 Sample nested OpenMP code on the left and Subteam code on the right

hybrid version. The only addition is the "num_threads" clause to each inner parallel region to specify the number of threads. The first (outer) level OpenMP exploits coarse-grained parallelism between zones.

A code sketch of the iteration loop using nested OpenMP is illustrated in Fig. 1. The outer level parallelization is adopted from the MPI approach: workloads from zones are explicitly distributed among the outer-level threads. The difference is that OpenMP now works on the shared data space as opposed to private data in the MPI version. The load balancing is done statically through the same bin-packing algorithm where zones are first sorted by size, then assigned to the least loaded thread one by one. The routine "map_zones" returns the number and list of zones (num_proc_zones and proc_zone_id) assigned to a given thread (myid) as well as the number of threads (nthreads) for the inner parallel regions. This information is then passed to the "num_threads" clause in the solver routines. The MPI communication calls inside "exch_qbc" for boundary data exchange are replaced with direct memory copy and proper barrier synchronization.

In order to reduce the fork-and-join overhead associated with the inner-level parallel regions, a variant was also created: a single parallel construct is applied to the time step loop block and all inner-parallel regions are replaced with orphaned "do" constructs. This version, namely *version 2*, will be discussed together with the first version in the results section.

3.3 Subteam in OpenMP

The subteam version was derived from the nested OpenMP version. Changes include replacing the inner level parallel regions with orphaned "`omp do`" constructs and adding the "`onthreads`" clause to specify the subteam composition. The sample subteam code is listed in the right panel of Fig. 1. The main difference is in the call to "`map_zones`." This routine determines which subteam (`mytid`) the current thread belongs to and what members are in the current subteam (`pard_thread_team`) based on the load balancing scheme. Defining subteams could be simplified by introducing a runtime function for subteam formation and management, which is not included in the subteam proposal [4].

The same load-balancing scheme as described in the previous section is applied in the subteam version to create zone groups. Each subteam works on one zone group, and thus, the number of zone groups equals the number of subteams. We use an environment variable to specify the number of subteams at runtime. Threads assigned to each subteam will work on loop-level parallelism within each zone. There is no overlapping of thread ids among different subteams. Similar to the nested OpenMP version, the routine "`exch_qbc`" uses direct array copy and proper global barrier synchronization for boundary communication.

3.4 OpenMP at Outer Level

One of the advantages of OpenMP is its ability to handle unbalanced workload in a dynamic fashion without much user intervention. The programming effort is much less than the explicit approach described in previous sections for handling load balance. The trade-off is potentially higher overhead associated with dynamic scheduling and less thread-data affinity as would be achieved in a static approach. To examine the potential performance trade-off, we developed an OpenMP version that solely focuses on the coarse-grained parallelization of different zones of the multi-zone benchmark. As illustrated in Fig. 2 left panel, this version is much simpler and compact. The "`omp do`" directive is applied to the loop nest over multiple zones. There is no explicit coding for load balancing, which is achieved through the OpenMP dynamic runtime schedule. The use of the "`schedule(runtime)`" clause allows us to compare different OpenMP loop schedules. A "`zone_sort_id`" array is used to store zone ids in different sorting schemes.

3.5 Workqueuing Model

We developed a Taskq version of the BT-MZ benchmark based on the Intel workqueuing model. Because Intel implemented Taskq only in its C++ compiler for C/C++ applications and there is no other vendor compiler available at this point for testing the concept, we had to first convert the Fortran implementation of BT-MZ to the C counterpart. To minimize the performance impact from such a conversion, we did the following:

```
!$omp parallel private(zone,...)
 do step=1,niter
   call exch_qbc(u,...)
   !$omp do schedule(runtime)
   do iz = 1, num_zones
     zone = zone_sort_id(iz)
     call adi(u(zone),...)
   end do
 end do
!$omp end parallel
```

```
#pragma omp parallel private(zone)
 for (step=1;step<=niter;step++) {
   exch_qbc(u,...);
   #pragma intel omp taskq
   for (iz=0;iz<num_zones;iz++) {
     zone = zone_sort_id[iz];
     #pragma intel omp task \
                captureprivate(zone)
       adi(&u[zone],...);
 }}
```

Fig. 2 Code segment using OpenMP runtime scheduling (left) and Intel taskq directives (right)

- Fortran multi-dimensional arrays are converted to linearized C arrays, such as `u(m,i,j,k) -> u[m+5*(i+nxmax*(j+ny*k))]`,
- The `restrict` qualifier is added to pointer variables in subroutine argument to enable compiler to perform optimization without pre-assumed pointer aliasing, for example
  `void add(double *restrict u, double *restrict rhs,..)`.

Once we had the C version, the Taskq implementation of the BT-MZ benchmark (Fig. 2 right panel) is straightforward. Each work unit for a task is defined by the solver on an individual zone. The "`intel omp taskq`" directive is added to the loop nest over zones. Inside the zone loop nest, the "`intel omp task`" directive is used to generate tasks for each loop iteration and the *zone* value is preserved for each task by the "`captureprivate`" clause. The implicit synchronization at the end of the `taskq` construct guarantees the completion of all tasks before going to the next iteration. Again, to test the performance impact of workload ordering, we use "`zone_sort_id`" to store the sorted zone ids.

4 Performance Results

In this section, we present performance results obtained on three large parallel systems. We will first give a brief description of the systems and programming support.

4.1 Testing Environment

Most of our performance studies was conducted on an SGI Altix 3700BX2 system that is one of the 20 nodes in the Columbia supercomputer installed at NASA Ames Research Center [12]. The Altix BX2 node has 512 Intel Itanium 2 processors, each clocked at 1.6 GHz and containing 9 MB on-chip L3 data cache. Approximately 1 TB of global shared-access memory is provided through the SGI scalable non-uniform memory access flexible (NUMA ex) architecture. The underlying NUMAlink4 inter-

connect provides 6.4 GB/s bandwidth. A single Linux operating system runs on the Altix system, providing an ideal environment for shared-memory programming such as OpenMP.

The system is equipped with SGI message-passing toolkit (MPT 1.12) that supports MPI programming. We used the Intel Fortran, C/C++ 9.1 compilers for IA64 that support OpenMP 2.5 as well as the Taskq model. All of our experiments were run under the PBSpro batch system in a shared environment. In order to reduce variation in timing and improve performance, the "`dplace`" placement tool was used to bind processes/threads to physical processors.

Two other systems were used to study architecture-dependent issues of the benchmark application. An IBM Power5 cluster installed at NASA Ames consists of 40 p575+ nodes, each of which contains 8 dual-core Power5+ chips, clocked at 1.9 GHz, 36MB shared L3 cache between two cores within a chip, and 32 GB shared memory within a node. The memory bandwidth between L2 and L3 is 30 GB/s. The IBM XLF (version 10.1) compiler supports OpenMP 2.5, but not the nested OpenMP. The Sun-Fire E25K system at RWTH Aachen University is a 72-way shared memory NUMA architecture, populated with dual-core UltraSPARC IV processors and operated on Solaris 10. The clock speed of the processor, 1.05 GHz, is relatively low in today's standard, but the Sun Studio compiler on the system supports runtime thread af nity, which is very important for the nested OpenMP.

For testing the OpenMP Subteam concept as described in Section 3.3, we employed the OpenUH research compiler [13] that was extended to support the "`numthreads`" clause. This is essentially a source-to-source translation process and the generated code is then compiled with a native compiler. A small runtime library was developed to support basic subteam functions, such as loop iteration scheduling and synchronization for subteam threads.

4.2 Multi-level Parallelism on the SGI Altix

In order to compare different multi-level parallel versions of the BT-MZ benchmark, we first examine the performance impact from varying the number of zone groups on a given number of CPUs. The left panel of Fig. 3 plots benchmark timing in seconds as a function of the number of groups at 32 CPUs for the Class B problem size. The notation "$N_g \times N_t$" denotes the number of zone groups (N_g) formed for the first level parallelism and the number of threads (N_t) for the second level parallelism within each group. In the hybrid MPI+OpenMP version, N_g is the same as the number of MPI processes and N_t is the number of OpenMP threads per MPI process. In the nested OpenMP versions is the number of outer-level threads, and in the subteam version is the number of subteams.

Overall the subteam version is very close in performance to the MPI+OpenMP hybrid version. This indicates that the data layout of the subteam version is very similar to that of the hybrid version, even though Subteam uses shared data arrays and MPI uses private data arrays. At single level parallelization, either $N \times 1$ or $1 \times N$, the performance of three approaches is very close. Between the two ends, the nested-OpenMP v1 performs consistently 30–80% worse than the other two versions. By

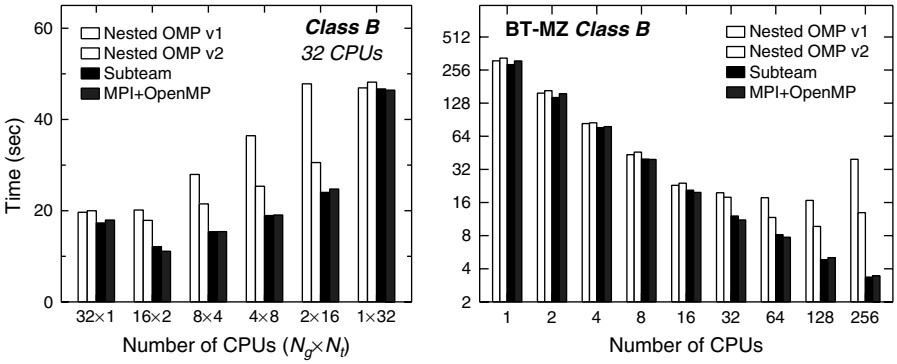

Fig. 3 Timing comparison of nested OpenMP, Subteam, and MPI+OpenMP versions of BT-MZ for the Class B problem, on the left for a given number of CPUs and on the right as a function of CPU counts. The results are from the SGI Altix

reducing the number of inner-level parallel regions in the second version (v2), the performance of nested OpenMP improved substantially, although it still lags behind. The large overhead associated with the inner parallel regions is likely due to the inability of the OpenMP runtime library to reuse threads ef ciently at the second level. Even though the *dplace* tool binds the rst-level threads properly, it has no control over the second-level threads. This result is consistent with the previous observation by Ayguade et al. [2].

The best performance is achieved by maximizing the number of zone groups as long as the workload can be balanced. For Class B, the optimal number of zone groups is 16. Beyond 16 CPUs, multi-level parallelism is needed for additional performance gain. Both subteam and hybrid versions follow this analysis, but the nested OpenMP tends to prefer a larger number of threads at the outer level, especially when the total number of CPUs increases.

The scaling results of BT-MZ from the best combinations of zone-groups and threads are summarized in right panel of Fig. 3. Both the subteam and hybrid versions scale well up to the measured CPU counts. Up to 16 CPUs, when only the outer-level parallelism is employed, the nested OpenMP versions performs similarly to the other two versions. Beyond 16 CPUs, nested OpenMP suffers from large overhead associated with the second-level parallelism and becomes much worse at larger CPU counts.

To understand better why the nested OpenMP codes suffer from performance degradation in the multi-level mode, we collected additional performance information from hardware counters available on the Altix and the results from the 48 runs are compared with the hybrid MPI+OpenMP runs in Fig. 4. The nested OpenMP v1 has the highest stalled cycles and L3 cache misses, which is an indication of thread-data mismatch. Stalled cycle is usually a result of waiting on resources, in particular memory. Although the nested OpenMP v2 reduced stalled cycles, but it has large L3 cache misses. Three pure OpenMP codes have somewhat higher TLB misses; but on the

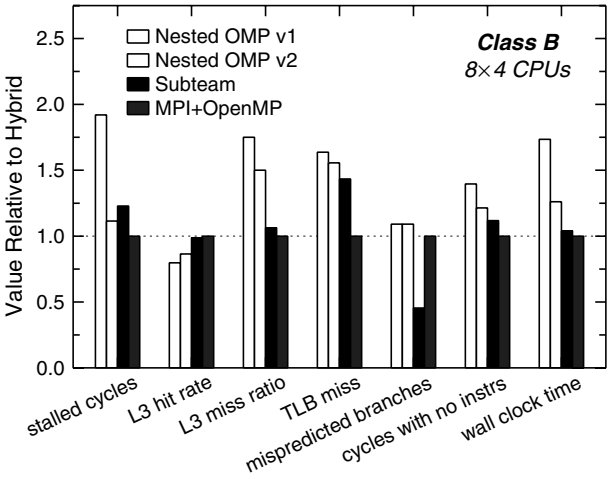

Fig. 4 Comparison of hardware performance counter results obtained on the Altix for the 8×4 runs of the four BT-MZ versions

Altix, a TLB miss has less impact on the overall performance. Other counters, such as L1 and L2 cache misses, have similar values for all four codes and are not included in the graph.

4.3 Impact from Different Architectures

The IBM p575 cluster supports hybrid MPI+OpenMP across multiple nodes, but OpenMP is only limited to within a node up to 16 threads. Since the IBM XLF compiler does not support nested OpenMP, we compare in Fig. 5 timing of the hybrid and subteam BT-MZ obtained on a p575 node with the Altix results at a given CPU count. The hybrid and subteam versions perform very similarly on both systems. At $N_t = 1$ without fine-grain OpenMP parallelization, BT-MZ shows timing very close on both systems. When the number of the OpenMP threads at the second level increases, the timing stays relatively constant on the IBM p575 but increases quickly on the SGI Altix. The difference is about a factor of two at $N_t = 16$. Obviously OpenMP runs more efficiently on the IBM p575, which can be partly attributed to its large memory bandwidth and relatively flat memory.

Figure 6 includes a comparison of the Altix results with the SunFire results for nested OpenMP in a wide range of N_s and N_t combinations. The speedup is relative to the single CPU timing on each system. Because of slow clock speed of the SunFire system, its single CPU performance is only about 1/4 of the Altix. However, at large CPU counts, the SunFire shows its strength for nested OpenMP and outperforms the Altix in many cases. In fact, the best nested OpenMP speedup on the SunFire is 70 at 16× 8, which matches closely with the hybrid result on the Altix. The best nested OpenMP result on the Altix is only 16.5 from 32×2. The improved performance of

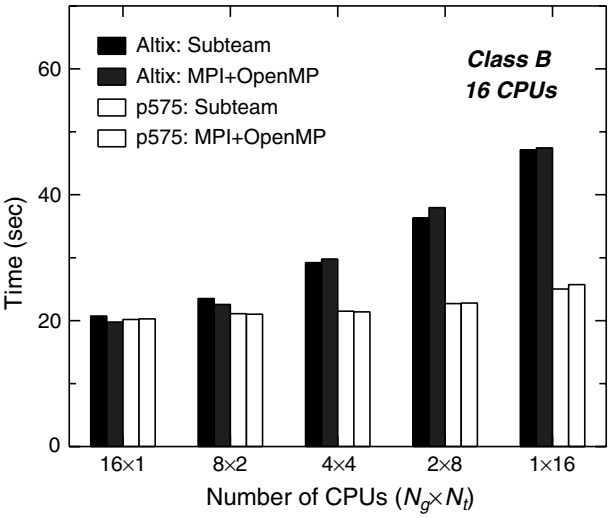

Fig. 5 Timing comparison of the hybrid and subteam BT-MZ on the SGI Altix and the IBM p575 at a given CPU count (16)

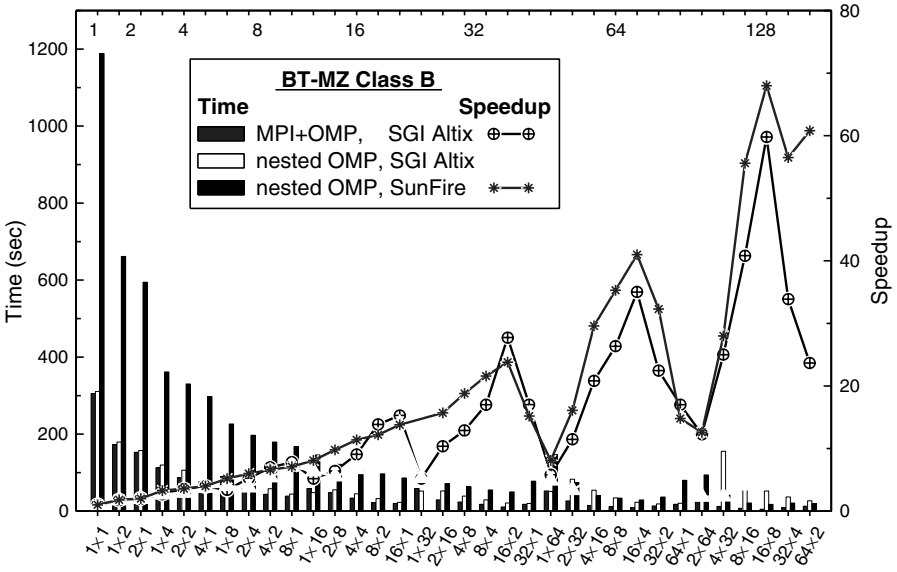

Fig. 6 Comparison of timing and speedup of the BT-MZ benchmark on both SGI Altix and SunFire in different N_g and N_t combinations

nested OpenMP on the SunFire is due to the support of thread af nity with thread reuse by the Sun Studio compiler, which is currently missing in the Intel compiler on the Altix.

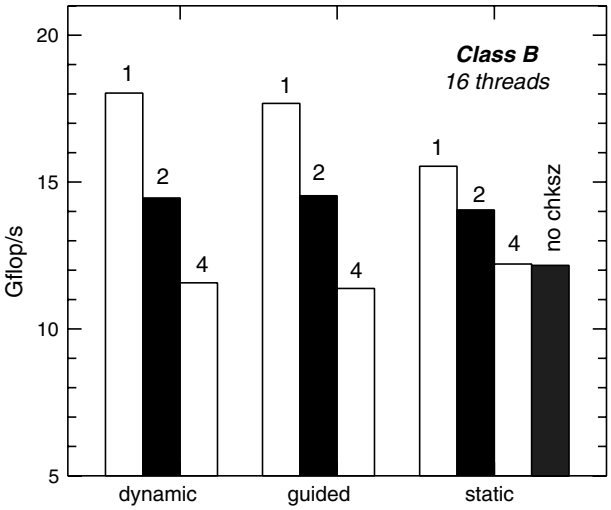

Fig. 7 Performance comparison of different schedule kinds for BT-MZ Class B, 16 threads. Numbers in the graph indicate chunk sizes. The last bar is for a `static` schedule without chunk size

4.4 Unbalanced Workload

To test the effectiveness of OpenMP runtime schedule kinds and more dynamic approaches on unbalanced workload, we focus on the single-level OpenMP versions of BT-MZ as described in Sects. 3.4 and 3.5, which exploit parallelism among unbalanced zones. No nested parallelism is considered here. All results discussed in the section were obtained on the SGI Altix.

4.4.1 Impact of Schedule Kind

The results from 16-thread runs using different runtime schedule kinds and chunk sizes are shown in Fig. 7. The "`dynamic,1`" schedule produces the best result for the given problem. As the chunk size increases, the performance decreases. The "`guided`" schedule is only slightly worse. The "`static`" schedule without chunk size (the last bar in the graph) shows its limitation in dealing with unbalanced workload and is as much as 50% worse than the "`dynamic,1`" schedule. The "`static,1`" (or cyclic) schedule improves the performance but not sufficiently.

4.4.2 Workload Ordering on Performance

As noted in the benchmark description, the zone workload in BT-MZ was designed to be uneven. Class B contains 64 zones whose sizes, shown in Fig. 8 on the left, range from 3 K to 60 K mesh points. The right graph in Fig. 8 shows the performance impact of three different orderings of zones in size on the "`dynamic,1`" schedule: natural (original) order, descending order, and ascending order. For comparison, the graph also includes results from a single-level OpenMP version that uses the static

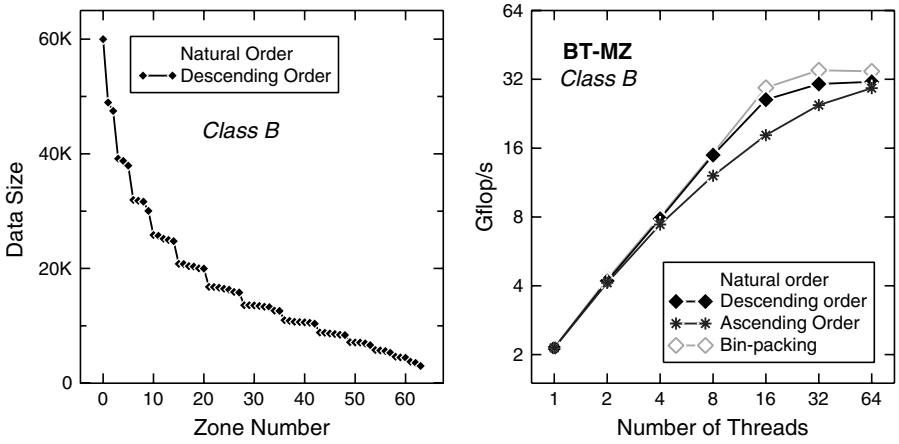

Fig. 8 Performance impact of different workload orderings on the "dynamic" schedule. Results from the static bin-packing approach are included for comparison

bin-packing algorithm for load balancing. This version is essentially the same as the nested OpenMP v1 described in Sect. 3.2 without the nested parallelism. We observe that by sorting zones into descending order, the performance can improve by as much as 45% (18–26 G op/s on 16 threads). This result supports the observation reported by Van Zee et al. [14] in their FLAME code using the workqueuing model.

The impact of different workload orderings on the "guided" schedule (not shown in the graph) is very similar to that on the "dynamic" schedule. It is worth noting that the dynamic approach for unbalanced workload is only slightly worse (5% beyond 16 threads) than the static bin-packing approach. However, the programming effort in the former case is considerably less.

4.4.3 Workqueuing Model

Before going into the workqueuing (or taskq) model, we first examine the performance change as a result of converting the code from Fortran to C. Due to pointer aliasing, a C code can suffer from the constraint in compiler optimization for pointers. In order to reduce or even eliminate pointer aliasing, one can either use the "restrict" modifier or rely on compiler flags. The Intel compiler provides the option "-no-alias" for this purpose. Table 1 summarizes the results of the OpenMP C version of BT-MZ using different compiler aliasing options and compares with the Fortran version. The no-alias option produces more than twice as much improvement in performance as the default aliasing option. Combining with the "restrict" modifier, the C code performs very close to the Fortran counterpart. This combined option was used in collecting the C results below.

Figure 9 compares the Intel Taskq version of BT-MZ with the single-level OpenMP versions (both C and Fortran) using dynamic scheduling for load balancing. It is encouraging to note that the Taskq version has similar performance to the single-level OpenMP C version using the "dynamic,1" schedule up to 32 threads. Only at 64

Table 1 Comparison of results from different aliasing options for the Class B, BT-MZ on 16 threads

Case		Time (s)	G op/s	Compiler option
C	Default aliasing	49.86	12.059	
	"`restrict`" keyword	37.17	16.175	`-restrict`
	No aliasing	23.49	25.594	`-fno-alias`
	Combination	23.38	25.718	`-restrict -fno-alias`
Fortran		23.01	26.124	

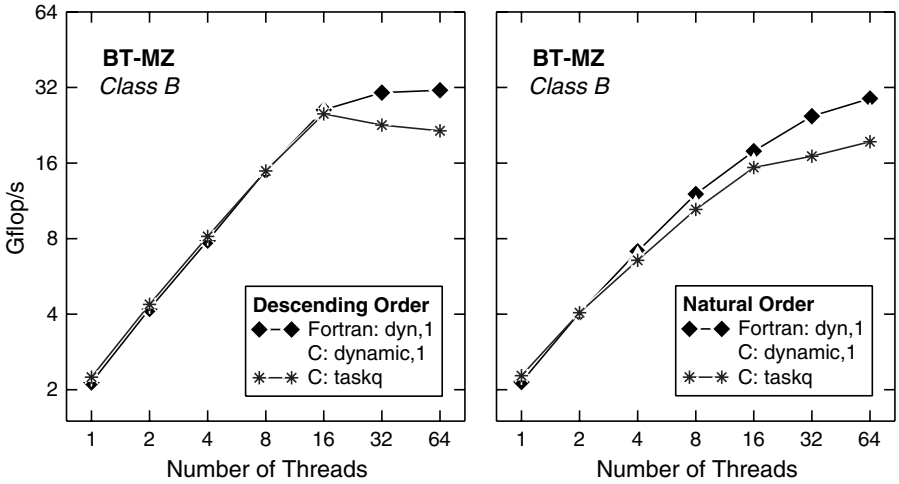

Fig. 9 Performance comparison of the Taskq version with the single-level OpenMP versions (in both C and Fortran) using the "`dynamic,1`" schedule

threads the dynamic-schedule version outperforms the Taskq version by about 20%. As illustrated by the two panels in the gure, sorting workload into descending order improves overall performance for Taskq as well. Comparing to the Fortran version, the performance of Taskq gets worse at larger thread counts, primarily due to the difference between Fortran and C.

5 Conclusion

We have presented performance evaluation of four different OpenMP approaches in dealing with multi-level parallelism and unbalanced workload, and compared with a hybrid MPI+OpenMP method. The nested OpenMP approach suffered from performance degradation as a result of large overhead and lack of thread reuse when invoking the inner level parallelism on the SGI Altix. By minimizing the number of inner level parallel regions we improved nested OpenMP performance dramatically. Another potential way to reduce overhead associated with nested parallel regions is by proper support of thread af nity with thread reuse as demonstrated on the SunFire system.

The approach based on the Subteam extension to OpenMP overcame some of the limitations with nested OpenMP and showed promise in achieving performance close to that of the hybrid MPI+OpenMP method. Our study also points out the importance of extending the Subteam proposal to include API for subteam creation and management.

It is very encouraging that the more dynamic approach provided by the workqueuing model showed great potential in dealing with unbalanced workload. This model can bene t from using a weight factor in scheduling tasks.

For future work, we would like to conduct our experiments on more platforms, in particular to study the support of nested parallelism from different compilers and runtime systems. A natural extension is to investigate the performance characteristics of nested parallelism under the workqueuing model. It is also important to extend our experience from a single benchmark application to more realistic applications.

Acknowledgements The authors would like to acknowledge fruitful discussions with Johnny Chang, Robert Hood, Piyush Mehrotra, and support from the staff at NAS division for many experiments conducted on the NAS supercomputers.

References

1. The OpenMP Standard. http://www.openmp.org/
2. Ayguade, E., Gonzalez, M., Martorell, X., Jost, G.: Employing nested OpenMP for the parallelization of multi-zone computational uid dynamics applications. In: Monien, B. (ed.) J. Parallel Distr. Comput., special issue 66(5), 686 (2006)
3. Zhang, G.: Extending the OpenMP standard for thread mapping and grouping. In: The International Workshop on OpenMP (IWOMP06), Reims, France (2006)
4. Chapman, B., Huang, L., Jin, H., Jost, G., de Supinski, B.: Toward enhancing OpenMP's work-sharing directives. In: The Euro-Par06 Conference, Dresden, Germany, 2006; LNCS 4128, pp. 645–654 (2006)
5. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structure for parallelism in OpenMP. In: The European Workshop on OpenMP (EWOMP99) (1999)
6. Su, E., Tian, X., Girkar, M., Haab, G., Shah, S., Petersen, P.: Compiler support of the workqueuing execution model for Intel SMP architectures. In: The European Workshop on OpenMP (EWOMP02) (2002)
7. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in OpenMP. In: The International Workshop on OpenMP (IWOMP07). Beijing, China (2007)
8. Bull, M.: OpenMP 3.0 Overview. Presented at the OpenMP BoF at the SC06 conference, 2006. http://www.compunity.org/futures/
9. Jin, H., Van der Wijngaart, R.F.: Performance characteristics of the multi-sone NAS parallel benchmarks. In: Monien, B. (ed.) J. Parallel Distr. Comput., special issue 66(5), 674 (2006)
10. Van der Wijngaart, R.F., Jin, H.: The NAS Parallel Benchmarks, Multi-Zone Versions. NAS Technical Report NAS-03-010, NASA Ames Research Center, 2003. http://www.nas.nasa.gov/Software/NPB/
11. Bailey, D., Barton, J., Lasinksi, T., Simon, H.: The NAS Parallel Benchmarks. NAS Technical Report RNR-91-002, NASA Ames Research Center (1991)
12. Biswas, R., Djomehri, M.J., Hood, R., Jin, H., Kiris, C., Saini, S.: An application-based performance characterization of the Columbia supercluster. In: Proceedings of the SC05 Conference. Seattle, WA, November 12–18 (2005)
13. OpenUH Research Compiler. http://www.cs.uh.edu/openuh
14. Van Zee, F., Bientinesi, P., Low, T.M., Van de Geijn, R.: Scalable parallelization of FLAME code via the workquenuing model. ACM Trans. Math. Softw. 34(2), 1–29, article 10 (2008)