Performance Evaluation of a Multi-Zone Application in Different OpenMP Approaches

Haoqiang Jin
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000
hjin@nas.nasa.gov

Barbara Chapman, Lei Huang
Department of Computer Science, University of Houston, Houston, TX 77004
{chapman,leihuang}@cs.uh.edu

Abstract

We describe a performance study of a multi-zone application benchmark implemented in several OpenMP approaches that exploit multi-level parallelism and deal with unbalanced workload. The multi-zone application was derived from the well-known NAS Parallel Benchmarks (NPB) suite that involves flow solvers on collections of loosely coupled discretization meshes. Parallel versions of this application have been developed using the Subteam concept and Workqueuing model as extensions to the current OpenMP. We examine the performance impact of these extensions to OpenMP on a large shared memory machine and compare with hybrid and nested OpenMP programming models.

1. Introduction

Since the 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 [8] 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 the support on 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 of thread affinity at a single level. To remedy this deficiency, the NANOS compiler team [1] has introduced the "GROUPS" clause to the outer parallel region to specify a thread group composition priori to the start of nested parallel regions, and Zhang [13] proposed extensions for thread mapping and grouping.

Chapman and collaborators [5] 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 subteam is similar to 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 “onthreads” clause to a work-sharing directive so that the work will be performed amongst the subset of threads, including the implicit barrier at the end of the construct.

One of the prominent extensions to the current OpenMP is the Workqueuing (or Taskq) model first introduced by Shah et al. [9] and implemented in the Intel C++ compiler [10]. It was designed to work with recursive algorithm 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 [4]. Although the final “tasking” directive in OpenMP 3.0 might not be the same as the original Intel Taskq proposal, it would still be quite intuitive to understand what potential of the more dynamic approach could offer in more applications.

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

2. Multi-Zone Application Benchmark

The multi-zone application benchmarks were developed [6,11] as an extension to the original NAS Parallel Benchmarks (NPBs) [2]. 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 structured-mesh production flow solver codes, provides relatively, easily exploitable 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 D. The difference comes from how the number of zones and the size of each zone in each benchmark are defined. We focus our study to the BT-MZ benchmark because it was designed to have uneven sized zones, which allows us to test various load balance strategies. As an example, the Class B problem has 64 zones with size ranging from 3K to 60K. Previously, the hybrid MPI+OpenMP [6] and nested OpenMP [1] 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 or its extension to implement the multi-zone BT-MZ benchmark. Three of the approaches exploit multi-level parallelism and the other two concern about 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 to 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 additional adjustment from OpenMP threads [6]. In the strategy, multiple zones are clustered into zone groups among which the computational workload is evenly distributed. Each zone group is then assigned to an MPI process for parallel execution. This process 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. [6] 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 in the 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 sketch of the iteration loop using the nested OpenMP code is illustrated in Figure 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 opposite to private data in the MPI version. The load balance 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 data 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 "omp do" constructs. This version, namely, version 2, will be discussed in the result section.

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

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

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 Figure 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 (*proc_thread_team*). This process could be simplified by introducing a runtime function for subteam formation and management, which is not included in the subteam proposal [5].

The same load-balancing scheme as described in previous sections 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 to the number of subteams. We use an environment variable to specify the number of subteams at the 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 as described in previous sections for handling load balance. The tradeoff 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 tradeoff, we developed an OpenMP version that solely focuses on the coarse-grained parallelization at different zones of the multi-zone benchmark. As illustrated in Figure 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.

```
!$omp parallel private(zone,..)
do step=1,niter
  call exch_qbc(u,...)
  !$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],..);
}}
```

Figure 2: Code segment using OpenMP runtime scheduling (left) and taskq directives (right).

3.5. Workqueuing Model

We have 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 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:

- 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 have the C version, the Taskq implementation of the BT-MZ benchmark (Figure 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. 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 a large shared memory system. We will first give a brief system description and programming support.

4.1. Testing Environment

For our study we used an SGI Altix 3700BX2 system that was one of the 20 nodes in the Columbia supercomputer installed at NASA Ames Research Center [3]. The Altix BX2 node has 512 Intel Itanium 2 processors, each with 1.6 GHz and 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 (NUMAflex) architecture. The underlying NUMAlink4 interconnect provides 6.4 GB/s bandwidth and scales linearly with the number of processors. A single Linux operating system runs on each Altix system, providing an ideal environment for shared-memory programming such as OpenMP.

The system is equipped with SGI multi-processing 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 SGI "dplace" placement tool was used to bind processes/threads to physical processors.

For testing the OpenMP Subteam concept as described in Section 3.3, we employed the Open64 research compiler [7] that was extended to support the "onthreads" 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 has been developed to support *subteaming* such as loop iteration scheduling and synchronization for subteam threads.

4.2. Multi-level Parallelism

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 Figure 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. N_g in the hybrid MPI+OpenMP version is the same as the number of MPI processes and N_t is the number of OpenMP threads per MPI process.

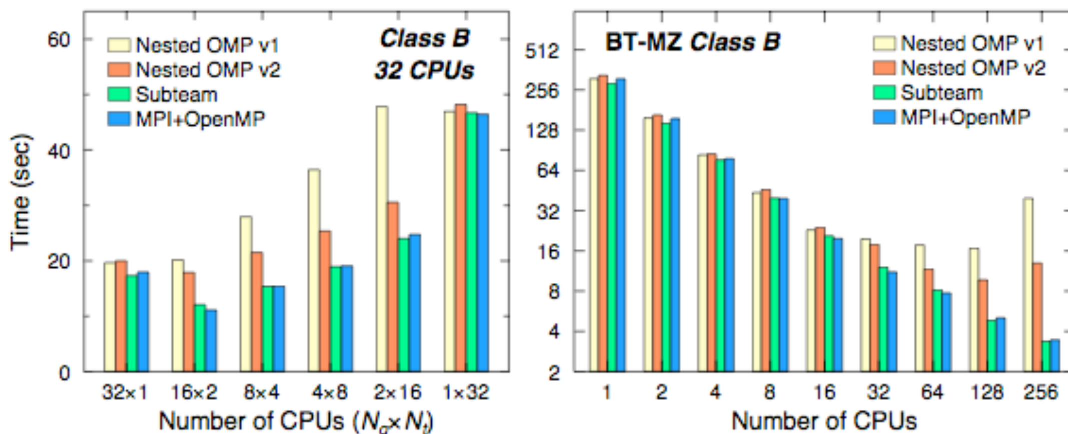

Figure 3: Timing comparison of nested OpenMP, Subteam, and MPI+OpenMP versions of BT-MZ for the Class B problem, on the left for a given number of CPUs and on the right as a function of CPU counts.

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 a single level parallelization, either N×1 or 1×N, the performance of three approaches is very close. Between the two ends, the nested-OpenMP v1 performed consistently worse by 30-80% than the other two versions. By reducing the number of inner-level parallel regions in the second version (v2), the performance of nested OpenMP has 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 to reuse threads efficiently at the second level. Even though the *dplace* tool binds the first-level threads properly, it has no control over the second-level threads. This result is consistent with the previous observation by Ayguade et al. [1]

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. Both subteam and hybrid versions follow this analysis, but the nested OpenMP tends to prefer 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 Figure 3. Both the subteam and hybrid versions scaled well up to the measured CPU counts. Up to 16 CPUs, when only the outer-level parallelism was employed, the nested OpenMP versions performed similarly to other two versions. Beyond 16 CPUs, nested OpenMP suffered from large ovehead associated with the second-level parallelism and became much worse at larger CPU counts.

4.3. Unbalanced Workload

To test the effectiveness of OpenMP runtime schedule kinds and more dynamic approaches on unbalanced workload, we ran the single-level OpenMP versions of BT-MZ as described in Sections 3.4 and 3.5.

4.3.1. Impact of Schedule Kind

The results from 16-thread runs using different runtime schedule kinds and chunk sizes are shown in Figure 4. The "dynamic,1" schedule produced 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 shows its limitation in dealing with unbalanced workload and is as much as 50% worse than "dynamic,1". The "static,1" (or cyclic) schedule improved the performance but not sufficient.

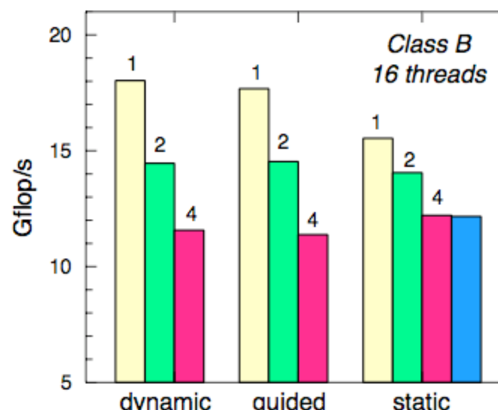

Figure 4: Performance comparison of different schedule kinds for BT-MZ Class B, 16 threads. Numbers in the graph indicate chunk sizes.

4.3.2. 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 Figure 5 on the left, range from 3K to 60K. The right graph in Figure 5 shows the performance impact on the "dynamic,1" schedule from three different orderings of zones in size: natural (original) order, descending order, and ascending order. We observe that by sorting zones into descending order, the performance can improve as much as 45% (18 to 26 Gflop/s on 16 threads). This result supports the observation reported by Van Zee et al. [12] in their FLAME code using the workqueuing model.

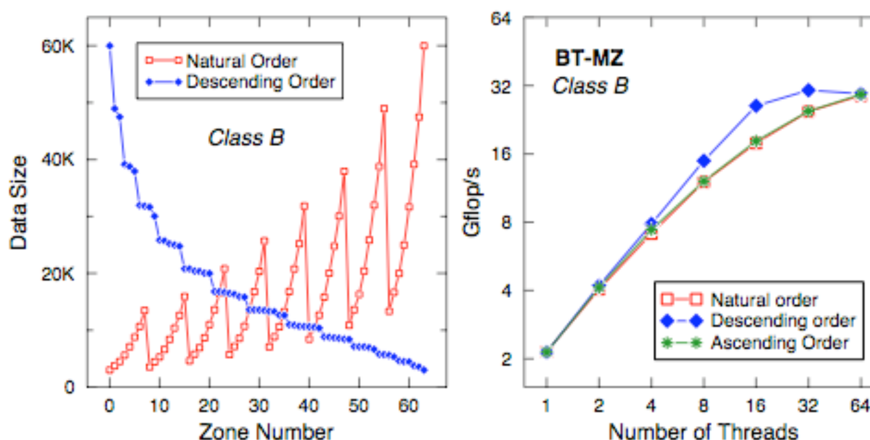

Figure 5: Performance impact from different workload orderings.

4.3.3. Workqueuing Model

Before going into the workqueuing (or taskq) model, we first examine the performance change as a result of converting from Fortran to C language. 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 "-fno-alias" for this purpose. We have used different combinations of compiler aliasing options to test the OpenMP C version of BT-MZ. For Class B and 16 threads, the no-alias option produced more than twice as much improvement in performance over the default aliasing option (49.9 secs versus 23.5 secs). Combining with the "restrict" modifier, the C code is only slightly worse than the Fortran counterpart (23.4 secs versus 23.0 secs). This combined option (-restrict -fno-alias) was used in collecting the C results below.

Figure 6 compares the Intel Taskq version of BT-MZ with the standard OpenMP versions using dynamic scheduling and static bin-packing algorithm for load balancing. It is encouraging to note that Taskq has similar performance to the OpenMP version using the "dynamic,1" schedule. Sorting workload into descending order improves overall performance. It is also worth to note that the dynamic approach for unbalanced workload is only slightly worse than the static bin-packing approach. However, the programming effort in the former case is considerably less.

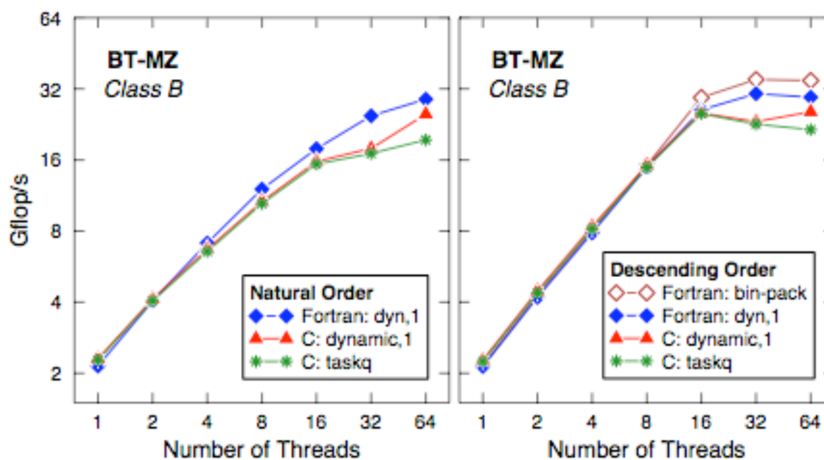

Figure 6: Performance comparison of Taskq with other approaches.

5. Conclusion

We have presented performance evaluation of four different OpenMP approaches in dealing with multi-level parallelism and unbalanced workload. The approach based on the Subteam extension to OpenMP overcame some of the limitations with nested OpenMP and showed promises in achieving performance close to that of the hybrid MPI+OpenMP method. By minimizing the number of inner level parallel regions we have improved nested OpenMP performance dramatically. Another way to reduce overhead associated with nested parallel regions is by predefining a thread tree structure as proposed in [1] and [13] so that the runtime can perform better scheduling optimization. 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 benefit from using a weight factor in scheduling tasks. For future work, we would like to conduct our experiments on more platforms and extend our experience from a single benchmark application to more realistic applications.

Acknowledgements

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

References

[1] E. Ayguade, M. Gonzalez, X. Martorell, and G. Jost, "Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications," J. of Parallel and Distributed Computing, special issue, ed. B. Monien, Vol. 66, No. 5, 2006, p686.

[2] D. Bailey, J. Barton, T. Lasinksi, and H. Simon, "The NAS Parallel Benchmarks," NAS Technical Report RNR-91-002, NASA Ames Research Center, 1991.

[3] R. Biswas, M.J. Djomehri, R. Hood, H. Jin, C. Kiris, and S. Saini, "An Application-Based Performance Characterization of the Columbia Supercluster," in Proc. of SC05, Seattle, WA, November 12-18, 2005.

[4] M. Bull, "OpenMP 3.0 Overview", presented at the OpenMP BoF at the SC06 conference, 2006. http://www.compunity.org/futures/.

[5] B. Chapman, L. Huang, H. Jin, G. Jost, and B. de Supinski, "Toward Enhancing OpenMP's Work-Sharing Directives," in the Euro-Par Conference, Dresden, Germany, 2006; LNCS 4128, 2006 pp.645-654.

[6] H. Jin and R.F. Van der Wijngaart, "Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks," J. of Parallel and Distributed Computing, special issue, ed. B. Monien, Vol. 66, No. 5, 2006, p674.

[7] Open64 Research Compiler, http://www.open64.net/.

[8] The OpenMP Standard, http://www.openmp.org/.

[9] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible Control Structure for Parallelism in OpenMP," in the European Workshop on OpenMP (EWOMP), 1999.

[10] E. Su, X. Tian, M. Girkar, G. Haab, S. Shah, and P. Petersen, "Compiler Support of the Workqueuing Execution Model for Intel SMP Architectures," in the European Workshop on OpenMP (EWOMP), 2002.

[11] R.F. Van der Wijngaart and H. Jin, "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/.

[12] F. Van Zee, P. Bientinesi, T.M. Low, R. Van de Geijn, "Scalable Parallelization of FLAME Code via the Workquenuing Model," submitted to ACM Trans. on Math. Software, 2006.

[13] G. Zhang, "Extending the OpenMP Standard for Thread Mapping and Grouping," in the International Workshop on OpenMP (IWOMP), Reims, France, 2006.