Employing Nested OpenMP for the Parallelization of Multi-Zone Computational Fluid Dynamics Applications

Eduard Ayguade, Marc Gonzalez, Xavier Martorell
Centre Europeu de Parallelism de Barcelona, Computer Architecture Department (UPC)
cr. Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain
eduard@ac.upc.es

Gabriele Jost[1]
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
gjost@nas.nasa.gov

Abstract

In this paper we describe the parallelization of the multi-zone code versions of the NAS Parallel Benchmarks employing multi-level OpenMP parallelism. For our study we use the NanosCompiler, which supports nesting of OpenMP directives and provides clauses to control the grouping of threads, load balancing, and synchronization. We report the benchmark results, compare the timings with those of different hybrid parallelization paradigms and discuss OpenMP implementation issues which effect the performance of multi-level parallel applications.

1 Introduction

Parallel architectures are an instrumental tool for the execution of compute intensive applications. Current programming models support distributed memory, shared memory, and clusters of shared memory architectures. An example of the support of distributed memory programming is MPI [16], which provides the functionality for process communication and synchronization. OpenMP [17] was introduced as an industrial standard for shared-memory programming with directives. The directives support loop level parallelization and require a shared address. The OpenMP programming paradigm provides ease of programming when developing parallel applications. For applications exhibiting multiple levels of parallelism the current most common programming paradigms are hybrid approaches such as the combination of MPI and OpenMP, or the MLP [19] model developed at NASA Ames. However, there is not much experience in the parallelization of applications with multiple levels of parallelism using OpenMP only.

The lack of compilers that are able to exploit further parallelism inside a parallel region has been the main cause of this problem, which has favored the practice of combining several programming models to exploit multiple levels of parallelism on a large number of processors. The nesting of parallel constructs in OpenMP is a feature that requires attention in future releases of OpenMP compilers. Some research platforms, such as the OpenMP NanosCompiler [8], have been developed to show the feasibility of exploiting nested parallelism in OpenMP and to serve as testbeds for new extensions in this direction. The OpenMP NanosCompiler accepts Fortran-77

[1] The author is an employee of Computer Sciences Corporation

code containing OpenMP directives and generates plain Fortran-77 code with calls to the NthLib thread library [14]. NthLib allows for multilevel parallel execution such that inner parallel constructs are not being serialized. The NanosCompiler programming model supports several extensions to the OpenMP standard to allow the user to control the allocation of work to the participating threads. By supporting nested OpenMP directives the NanosCompiler offers a convenient path to multilevel parallelism.

Multi-zone codes are a class of applications featuring multiple levels of parallelism. They are commonly used in large scale Computational Fluid Dynamics (CFD) applications. A single mesh is often not sufficient to describe a complex domain and multiple meshes are used to cover it. These meshes are referred to as zones which yields the name *multi-zone* code. An example for a production multi-zone code is OVERFLOW [5] where the flow equations are solved independently within each zone. After each iteration boundary values are exchanged between neighboring zones. Solutions within each zone can be computed independently, providing coarse grain parallelism. Fine grain loop level parallelism can be exploited within each zone. A set of benchmarks has recently been released which captures this behavior and allows the analysis and evaluation of multi-level programming paradigms. These benchmarks are multi-zone versions of the well known NAS Parallel Benchmarks [2], [3]. The NPB Multi-Zone (NPB-MZ) are described in [20]. A serial and two hybrid parallel reference implementations of the NPB-MZ are available. We have developed a nested OpenMP version of the NPB-MZ and used the NanosCompiler to evaluate the efficiency on several hardware platforms.

The rest of the paper is structured as follows: Section 2 summarizes the NanosCompiler extensions to the OpenMP standard. Section 3 describes the implementation of the NPB-MZ. Section 4 presents timing results for the benchmark codes. Related work is discussed in Section 5 and the conclusions of the study are presented in Section 6.

2 The NanosCompiler

OpenMP provides a fork-and-join execution model in which a program begins execution as a single process or thread. This thread executes sequentially until a `PARALLEL` construct is found. At this time, the thread creates a team of threads and it becomes its master thread. All threads execute the statements lexically enclosed by the parallel construct. Work-sharing constructs (`DO`, `SECTIONS` and `SINGLE`) are provided to divide the execution of the enclosed code region among the members of a team. All threads are independent and may synchronize at the end of each work-sharing construct or at specific points (specified by the `BARRIER` directive). Exclusive execution mode is also possible through the definition of `CRITICAL` and `ORDERED` regions. If a thread in a team encounters a new `PARALLEL` construct, it creates a new team and it becomes its master thread. OpenMP v2.0 provides the `NUM_THREADS` clause to restrict the number of threads that compose the team.

The NanosCompiler extension to OpenMP to support multilevel parallelization is based on the concept of *thread groups*. A group of threads is composed of a subset of the total number of threads available in the team to run a parallel construct. In a parallel construct, the programmer may define the number of groups and the composition of each one. When a thread in the current team encounters a `PARALLEL` construct defining groups, the thread creates a new team and it becomes its master thread. The new team is composed of as many threads as the number of groups. The rest of the threads are used to support the execution of nested parallel constructs. In

other words, the definition of groups establishes an allocation strategy for the inner levels of parallelism. To define groups of threads, the NanosCompiler supports the GROUPS clause extension to the PARALLEL directive.

```
C$OMP PARALLEL GROUPS (gspec)
...
C$OMP END PARALLEL
```

Different formats for the GROUPS clause argument gspec are allowed [9]. The simplest specifies the number of groups and performs an equal partition of the total number of threads to the groups:

```
gspec = ngroups
```

The argument ngroups specifies the number of groups to be defined. This format assumes that work is well balanced among groups and therefore all of them receive the same number of threads to exploit inner levels of parallelism. At runtime, the composition of each group is determined by equally distributing the available threads among the groups.

```
gspec = ngroups, weight
```

In this case, the user specifies the number of groups (ngroups) and an integer vector (weight) indicating the relative weight of the computation that each group has to perform. From this information and the number of threads available in the team, the threads are allocated to the groups at runtime. The weight vector is allocated by the user and its values are computed from information available within the application itself (for instance iteration space, computational complexity).

3 The Multi-Zone Versions of the NAS Parallel Benchmarks

The purpose of the NPB-MZ is to capture the multiple levels of parallelism inherent in many full scale applications. Multi-zone versions of the NAS Parallel Benchmarks LU, BT, and SP where developed by dividing the discretization mesh into a two-dimensional tiling of three-dimensional zones. Within all zones the LU, BT, and SP problems are solved to advance the time-dependent solution. The same kernel solvers are used in the multi-zone codes as in the single-zone codes. Exchange of boundary values takes place after each time step. A detailed discussion of the NPB-MZ can be found in [20]. We will refer to the multi-zone versions of the LU, BT, and SP benchmarks as LU-MZ, BT-MZ, and SP-MZ.

3.1 The Hybrid Implementations of the NPB-MZ

The source code distribution of the NPB-MZ includes two different hybrid implementations. The first hybrid implementation is based on using MPI for the coarse grained parallelization on zone-level and OpenMP for fine grained loop level parallelism within each of the zones. The MPI programming paradigm assumes a private address space for each process. Data is transferred by explicitly exchanging messages via calls to the MPI library. This model was originally designed for distributed memory architectures but is also suitable for shared memory systems. In the NPB-

MZ MPI/OpenMP implementation the number of processes is defined at compile time. Each process is assigned a number of zones and spawns a number of OpenMP threads in order to achieve a balanced load. Data is communicated at the beginning of the time step loop using MPI. There is no communication during the solution of the LU, BT, and SP problems within one zone. The OpenMP parallelization is similar to the single-zone versions as described in [10].

The second hybrid implementation that is part of the NPB-MZ is based on the MLP programming model developed by Taft [19] at NASA Ames Research Center. The MLP programming model is similar to MPI/OpenMP, using a mix of coarse grain process level parallelization and loop level OpenMP parallelization. As it is the case with MPI, a private address space is assumed for each process. The MLP programming paradigm, however, also requires a shared memory arena which is accessible by all processes. Communication is done by reading from and writing to the shared memory arena. Libraries supporting the MLP paradigm usually provide routines for process creation, shared memory allocation, and process synchronization. Details about the process level parallelization in the MLP paradigm and corresponding library support can be found in [11]. The MLP implementation of the NPB-MZ is very similar to the MPI/OpenMP implementation. Communication is handled by copying the boundary values to and from the shared memory arena. The OpenMP parallelization is identical in both versions. The MLP approach was developed for ccNUMA architectures and explicitly takes advantage of the availability of shared memory.

Both hybrid implementations apply a load balancing algorithm to determine the number of threads that each process spawns. A detailed description of the reference implementations which are part of the benchmark distribution can be found in [12].

3.2 The Nested OpenMP Implementations of the NPB-MZ

The nested OpenMP implementation is currently not part of the NPB-MZ distribution. It has been developed by the authors using the thread group extensions mentioned before. This implementation combines a coarse grained parallelization (inter-zone) and parallelization within the zones (intra-zone) parallelization, but employing OpenMP on both levels. The intra-zone parallelization is identical in the hybrid and the nested OpenMP implementations. The inter-zone parallelism is implemented by creating groups of threads and by assigning one or more zones to a thread group. The whole address space is shared by default among the threads working at both levels of parallelism. Data exchange at the zone boundaries is done in parallel by reading from and writing to the original application data structures. There is no need for using any special primitives such as MPI communication routines or MLP synchronization routines. This implementation just requires the addition of less than half a dozen OpenMP directives in each application. The same function that maps zones to MPI or MLP processes is used to map zones to thread groups. The mapping function generates two vectors that indicate which group executes each zone (`proc_zone_id`) and how many threads are allocated to each group (`proc_num_threads`). Since zones are not mapped in a consecutive way and the number of zones assigned to each group may be different, a couple of statements in the parallel regions at the outer level to control the execution of zones had to be added. The number of groups (`num_groups`) is controlled by an environment variable. Figure 1 shows an excerpt of the parallelization for the LU-MZ benchmark. The first part shows the parallelism at the inter-zone level. The intra-zone

parallelization occurs in routine `ssor`, which is identical to the parallelization used in the other two strategies (MPI+OpenMP and MLP).

```
      ...
      call map_zones(num_groups, num_zones, nx, ny, nz)
      ...
      do step = 1, itmax
        call exch_qbc(u, qbc, nx, nxmax, ny, nz, start5, qstart_west,
     $                qstart_east, qstart_south, qstart_north)

C$OMP PARALLEL PRIVATE(whoami, zone, flux, a, b, c, d, au, bu, cu,
C$OMP&                 du, tu, tv)
C$OMP& GROUPS(num_groups, proc_num_threads)
        whoami = omp_get_thread_num()
        do zone = 1, num_zones
          if (whoami .eq. proc_zone_id(zone)) then
            call ssor(u(start5(zone)), rsd(start5(zone)),
     $              frct(start5(zone)), qs(start1(zone)),
     $              rho_i(start1(zone)), flux,
     $              tv, a, b, c, d, tu, au, bu, cu, du,
     $              nx(zone), nxmax(zone), ny(zone), nz(zone))
          end if
        end do
C$OMP END PARALLEL
      end do
      ...

      subroutine ssor(u, rsd, frct, qs, rho_i, flux, tv, a, b, c, d,
     $                tu, au, bu, cu, du, nx, nxmax, ny, nz)
      ...

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(m,i,j,k,tmp,iam_cap,mthnum_cap)
!$OMP&                         SHARED(omega,dt,timeron,nx,nxmax,ny,nz)
      do k = 2, nz-1
!$OMP DO
        do j = 2, ny-1
          do i = 2, nx-1
            do m = 1, 5
              rsd(m,i,j,k) = dt * rsd(m,i,j,k)
            end do
          end do
        end do
!$OMP END DO nowait
      end do
      ...
!$OMP END PARALLEL
```

Figure 1: Parallelization of LU-MZ using GROUPS clause.

At this point the reader may want to know why there is a necessity for extending OpenMP to support thread groups. The current specification for OpenMP includes the NUM_THREADS

clause which tells the runtime environment the number of threads to be used in the execution of the PARALLEL region. With this extension it is possible to implement a nested parallel strategy similar to the one described above. However, it requires that the programmer explicitly controls the allocation of threads at each level of parallelism, as shown in Figure 2 (equivalent to Figure 1). This requires that the vectors that control the allocation of zones to groups are visible to the thread that is going to spawn the inner level of parallelism (common block inside routine `ssor`). Two problems are worth mentioning about this implementation. The first one is the lack of modularity of the approach. For example, now the programmer has coded in the application itself the fact that this routine is called from inside a parallel region; if called from a serial part of the application the behavior would not be appropriate. The version employing the NanosCompiler GROUPS clause extension is more modular since the context is implicit in the OpenMP runtime and the code is valid in all possible situations. The second problem is related to the usual implementation of nested parallelism in OpenMP. It is common practice to implement a pool of threads, so that when a thread arrives at a PARALLEL region the desired number of threads is taken from the pool. In the example depicted in Figure 2 this would be the number specified by the NUM_THREADS clause. This is the case for example in the runtime of the IBM XL compiler [21]. However, there is no guarantee that a particular thread is always executed on the same processor, so that data locality is not exploited. The definition of thread groups establishes an allocation strategy for the inner levels of parallelism, so that multiple instances of the same PARALLEL region or different regions with the same GROUPS definition will always use the same thread/processor mapping.

```
      ...
      call map_zones(num_groups, num_zones, nx, ny, nz)
      ...
      do step = 1, itmax
        call exch_qbc(u, qbc, nx, nxmax, ny, nz, start5, qstart_west,
     $                qstart_east, qstart_south, qstart_north)

C$OMP PARALLEL PRIVATE(whoami, zone, flux, a, b, c, d, au, bu, cu,
C$OMP&                 du, tu, tv)
C$OMP& NUM_THREADS(num_groups)
        whoami = omp_get_thread_num()
        do zone = 1, num_zones
          if (whoami .eq. proc_zone_id(zone)) then
            call ssor(u(start5(zone)), rsd(start5(zone)),
     $              frct(start5(zone)), qs(start1(zone)),
     $              rho_i(start1(zone)), flux,
     $              tv, a, b, c, d, tu, au, bu, cu, du,
     $              nx(zone), nxmax(zone), ny(zone), nz(zone))
          end if
        end do
C$OMP END PARALLEL
      end do
      ...

      subroutine ssor(u, rsd, frct, qs, rho_i, flux, tv, a, b, c, d,
     $                tu, au, bu, cu, du, nx, nxmax, ny, nz)
      ...
      integer proc_zone_id(num_zones), proc_num_threads(num_zones)
      common /thr_mapping/ num_groups, proc_zone_id, proc_num_threads
      ...

!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(m,i,j,k,tmp,iam_cap,mthnum_cap)
!$OMP&                         SHARED(omega,dt,timeron,nx,nxmax,ny,nz)
!$OMP& NUM_THREADS(proc_num_threads(omp_get_thread_num()+1))
      do k = 2, nz-1
!$OMP DO
        do j = 2, ny-1
          do i = 2, nx-1
            do m = 1, 5
              rsd(m,i,j,k) = dt * rsd(m,i,j,k)
            end do
          end do
        end do
!$OMP END DO nowait
      end do
      ...
!$OMP END PARALLEL
```

Figure 2: Parallelization of LU-MZ using NUM_THREADS clause.

4 Timing Results

We have run the BT-MZ, LU-MZ, and SP-MZ benchmarks of problem classes W, A, and B. The aggregate sizes for all benchmarks are:

- Class W: 64x64x8 grid points
- Class A: 128x128x16 grid points
- Class B: 304x208x17 grid points

We ran our tests on two platforms: an SGI Origin 3000 located at the NASA Ames Research Center and one frame of an IBM Regatta p690 located at the FZ Juelich Center in Germany.

The SGI Origin 3000 is a ccNUMA architecture with 4 CPUs per node. The CPUs are of type R12K with a clock rate of 400 MHz, 2 GB of local memory per node, and 8 MB of L2 cache. The peak performance of each CPU is 0.8 Gigaflops. The MLP implementations use the SMPlib library as described in [11]. The MIPSpro 7.4 Fortran Compiler [15] is used to compile the hybrid codes and the NanosCompiler for the nested OpenMP code. The compiler options –mp –O3 and –64 are set in both cases.

The IBM Regatta frame has 32 processors of type Power4+, running at 1.7 GHz. The main memory is 64 MB and the cache hierarchy has three levels: internal L1 cache with 64 KB instruction and 32 KB data (per processor), shared L2 cache with 1.5 MB (per chip = 2 processors), and shared L3 cache with 512 MB. The IBM XL Fortran compiler with the option -qsmp=omp is used to compile the hybrid MPI/OpenMP codes. The NanosCompiler and the GROUPS extension is used for the nested OpenMP codes. On the IBM platform there was no library support for the MLP programming model available. The native IBM compiler supports nested parallelism. Some tests were run employing the native IBM compiler together with the NUM_THREADS clause (as shown in Figure 2) to achieve nested parallelism. The option -qsmp=omp:nested_par was set in this case to compile the nested OpenMP version. The option -O3 was used for all cases.

In the charts we use the following notation:

- **NTH**: Nested OpenMP implementation using the NanosCompiler and the GROUPS clause.
- **IBM Nested**: Nested OpenMP implementation using the native IBM compiler and the NUM_THREADS clause.
- **NPxNT**: Number of CPUs expressed as number of processes (NP) times number of threads (NT). For the nested OpenMP code NP refers to the number of thread groups.

4.1 The BT-MZ Benchmark

The number of zones grows with the problem size. The number of zones is 4x4 for Class W and A, and 8x8 for Class B. The sizes of the zones vary widely. The ratio of the largest to the smallest zone is approximately 20. In order to achieve a good load balance a different number of threads has to be assigned to each group in the nested OpenMP codes. The same is true for the number of threads that are spawned by the processes in the hybrid codes.

Figure 3 shows results for the hybrid MPI/OpenMP version, the NanosCompiler using the GROUPS clause (NTH), and the nested OpenMP version compiled with the IBM native compiler and runtime system on the IBM Regatta. The timings show that the current implementation

of nested parallelism in the native IBM system is not able to achieve the scalability of the hybrid version or the NTH version. We suspect that the implementation of nested parallelism using a pool of threads is not taking benefit of the data locality that the hybrid version and the NTH version have. Although the runtime environment may ensure that the outer level of parallelism always uses the same kernel thread to execute each OpenMP thread, this is not guaranteed at the inner level. At the inner level, the threads that compose each team are dynamically selected from the pool, so there is no guarantee that the same kernel threads are used in all parallel regions. The timings for 16 CPUs and different combinations of processes/threads and groups/threads are shown in Figure 4. Notice that the performance of the IBM Nested implementation is best when all threads are used on the outer level. The hybrid and the NTH version behave better when nested parallelism is used, taking advantage of load-balancing on the inner level of parallelism and data locality.

The MIPSpro compiler and runtime environment on the SGI Origin do not support nested parallelism. The performance of the hybrid codes and the NTH implementation is very similar. Some timings for different numbers of CPUs and different benchmark classes are shown in Figure 5.

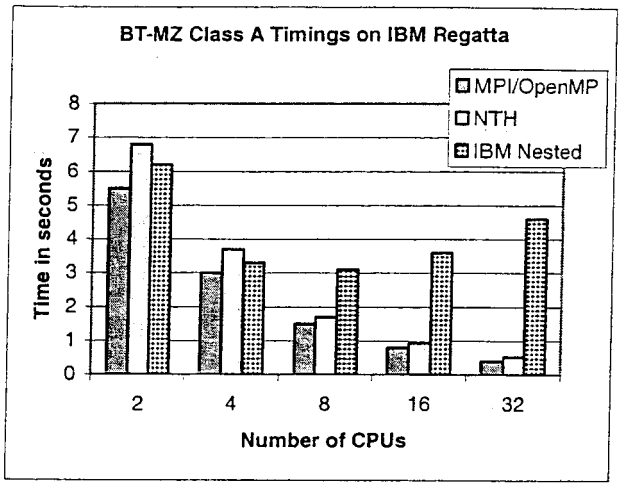

Figure 3: Timings for 20 iterations of BT-MZ. The best timings over all combinations of processes or groups and threads are reported

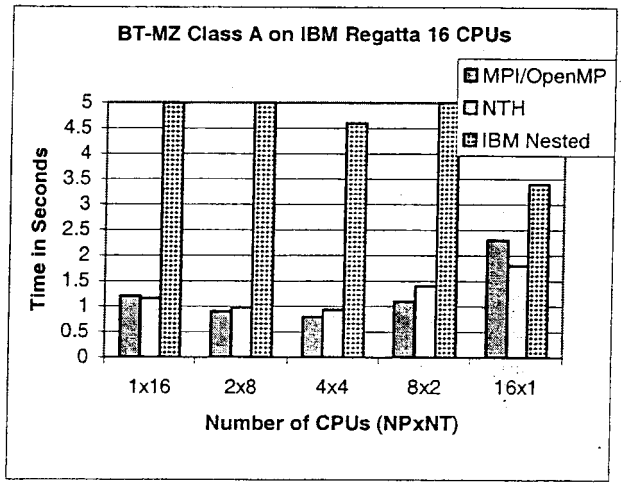

Figure 4: Timings for 20 iterations of BT-MZ Class A for different combinations of processes or groups and threads

Due to load balancing, the number of threads per process and the number of threads per group varies. We indicate the average number of threads per process or group in the timings charts. The performance of the nested OpenMP implementation is nearly identical to that of the hybrid codes. The thread groups implementation in the Nanos compiler and runtime environment guarantee the same mapping of kernel threads to OpenMP threads in all parallel regions, both at the outer and inner levels. This improves memory behavior and results in performance levels that are comparable to the hybrid versions. This demonstrates the importance of having these extensions in OpenMP and provides an efficient implementation for nested parallelism in OpenMP.

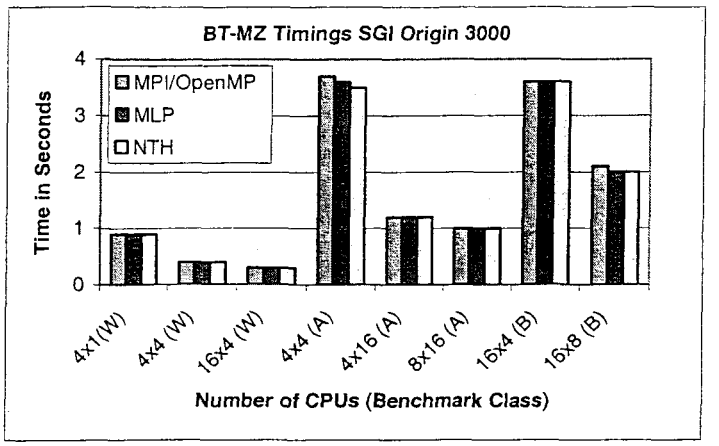

Figure 5: Timings for 20 iterations of different classes of BT-MZ. Reported are the best timings for each implementation over all numbers of processes or thread groups. The benchmark class is indicated in brackets

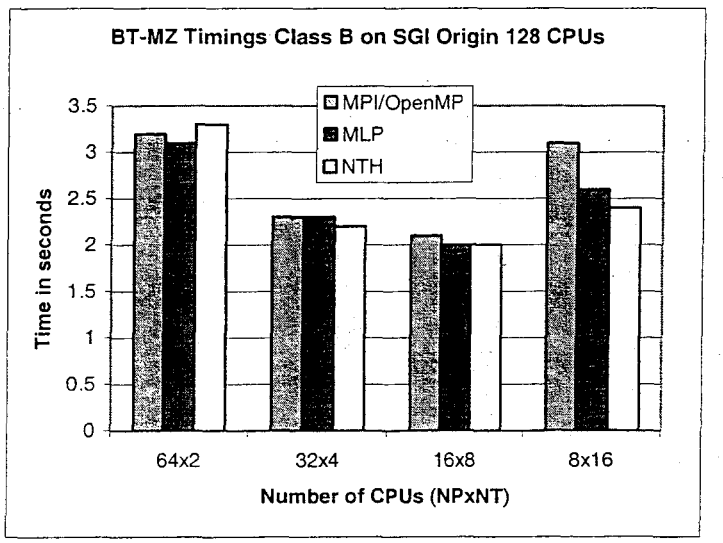

Figure 6: Timings for 20 iterations of BT-MZ for different processes or thread groups for a fixed number of CPUs

Figure 6 shows the impact of different combination of processes or groups and threads. The timings are shown for the problem Class B and 128 CPUs. The problem Class B has 64 zones. Using 64 processes or 64 thread groups did not allow the most efficient load balancing. The best load balancing was achieved using 16 processes in the hybrid codes and 16 groups in the NanosCompiler nested OpenMP code. The number of threads per process or group varies. In the chart we report the average number of threads per process. To illustrate the load balancing issue we show timeline views of time spent in useful calculations for different numbers of thread groups in Figure 7.

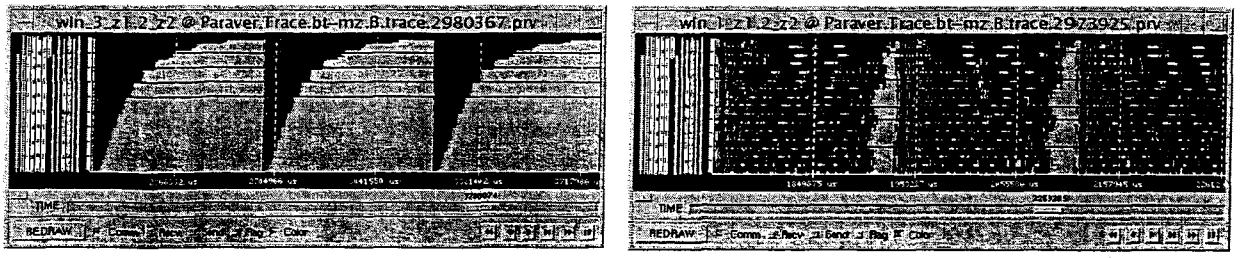

Figure 7: Timeline views of two BT-MZ Class B runs on 64 threads. Dark shading indicates useful computation time, light shading indicates idle time. The views show the timeline for 3 iterations. The left image results from a run using 64 thread groups, the right images from a run with 16 thread groups configuration. The large amount of useful computation time in the right image demonstrates a well balanced workload. The time scale in the right view is about 1/3 of the one in the left which demonstrates the high efficiency

To demonstrate the scalability of the different implementations on the SGI Origin 3000 the Gigaflop rate as reported by the benchmark is shown in Figure 8. The Class B performance for the number of processes or thread groups that produced the best results is reported. The number was the same for all three implementations. This is not surprising since the load balancing issue is the same in all versions. The three implementations show almost identical scalability, achieving about 28 Gigaflops/s for 128 CPUs.

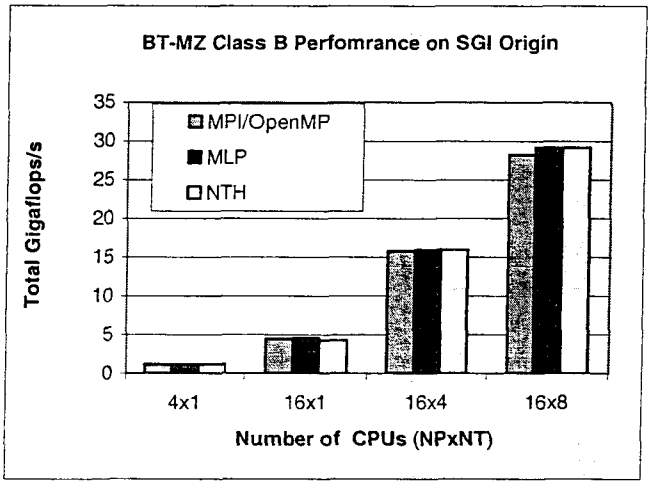

Figure 8: Performance of the BT-MZ Class B benchmark in terms of the total number of Gigaflops per second for an increasing number of CPUs

4.2 The SP-MZ Benchmark

Here the mesh is partitioned such that zones are identical in size. The number of zones grows with the problem size. The number of zones is 4x4 for Class W and A, and 8x8 for Class B. The SP-MZ benchmark is naturally load-balanced on the coarse level.

Timings for the different implementations and different benchmark classes on the SGI Origin are shown in Figure 9. As before, we report the timings for the best combinations of processes or groups and threads. The hybrid implementations achieve the best performance when employing a maximum number of processes on the coarse level. The use of multiple threads per process is only advantageous when the number of CPUs exceeds the number of zones. The situation is similar for the nested OpenMP code: It is best to employ groups consisting of only 1 thread, unless the number of CPUs exceeds the number of zones. As an example we show in Figure 10 the timings for problem Class B on different process or group and thread combinations.

Figure 11 shows the scalability of the different implementations in terms of the achieved Gigaflop rate. The scalability of the nested OpenMP code compiled with the NanosCompiler is comparable to that of the hybrid implementations, achieving about 23 Gigaflops on 128 CPUs versus 26 Gigaflops of the MPI/OpenMP implementation.

On the IBM Regatta it was advantageous to use multiple threads per process or group for the Class A benchmark. The timings are shown in Figure 12. The timings obtained with the NanosCompiler are comparable to the MPI/OpenMP timings.

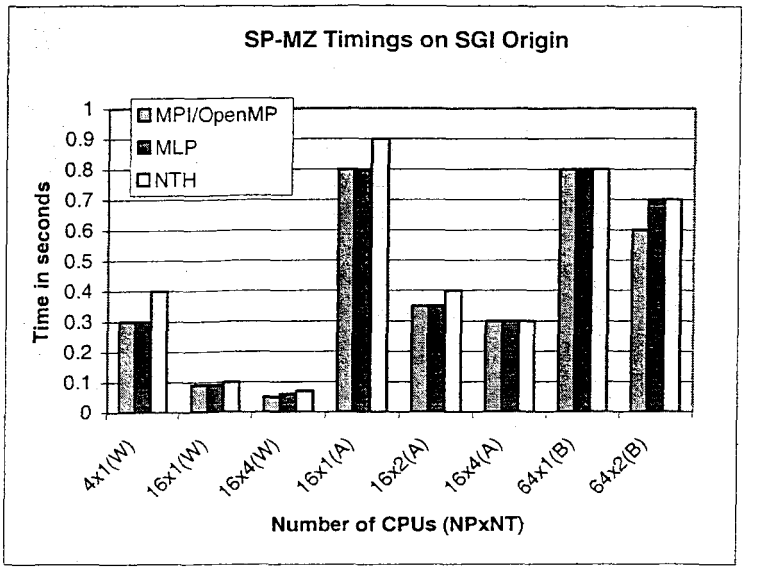

Figure 9: Timings for 20 iterations of different classes of SP-MZ. Reported are the best timings for each implementation over all combinations of processes or groups and threads.

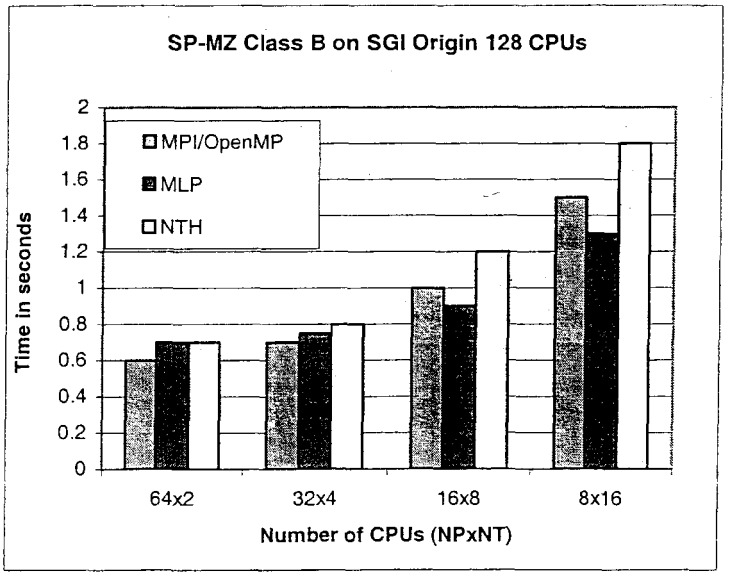

Figure 10: Timings for 20 iterations of SP-MZ for different process (or group) and thread combinations for a fixed number of CPUs. The number of zones for this case is 64

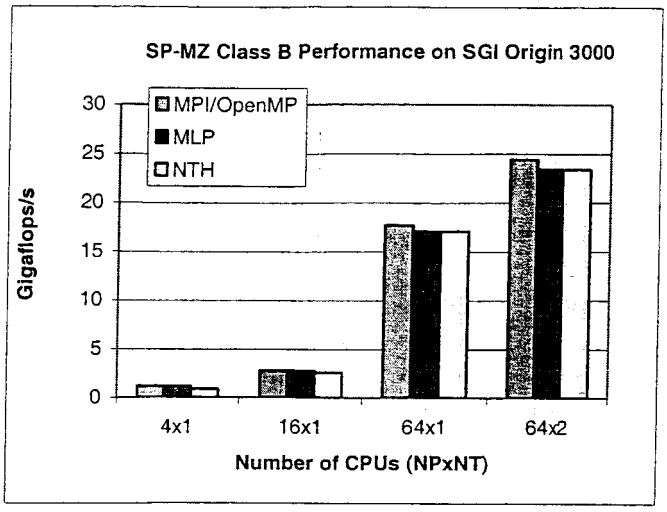

Figure 11: Performance of the SP-MZ Class B benchmark in terms of the total number of Gigaflops per second with an increasing number of CPUs

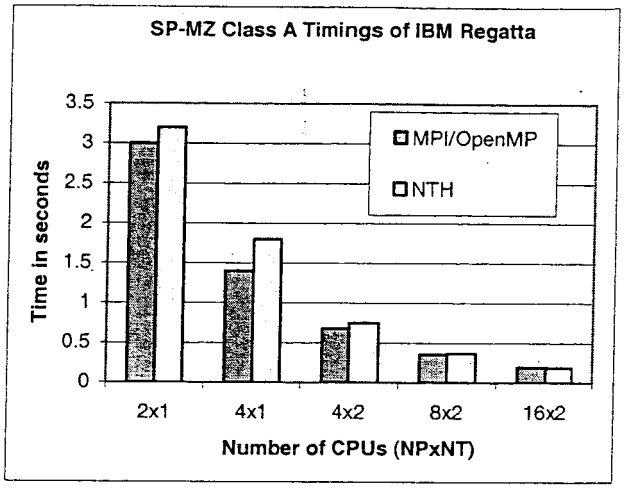

Figure 12: Timings reported are for 20 iterations. Reported are the best timings for each implementation over all combinations of processes or groups and threads

4.3 The LU-MZ Benchmark

In this case the number of zones is 4x4 for all problem sizes. The overall mesh is partitioned such that the zones are identical in size. This makes load balancing easy. The coarse grain parallelism in the hybrid codes is limited to 16 processes due to the structure of the benchmark. Parallelism beyond that has to be obtained at the fine grained level. Similarly, in the nested OpenMP code the number of thread groups is limited to 16. The timings for the SGI Origin are shown in Figure 13. As before, we show the combinations of processes or groups and threads that yielded the best results for the hybrid codes and the NanosCompiler

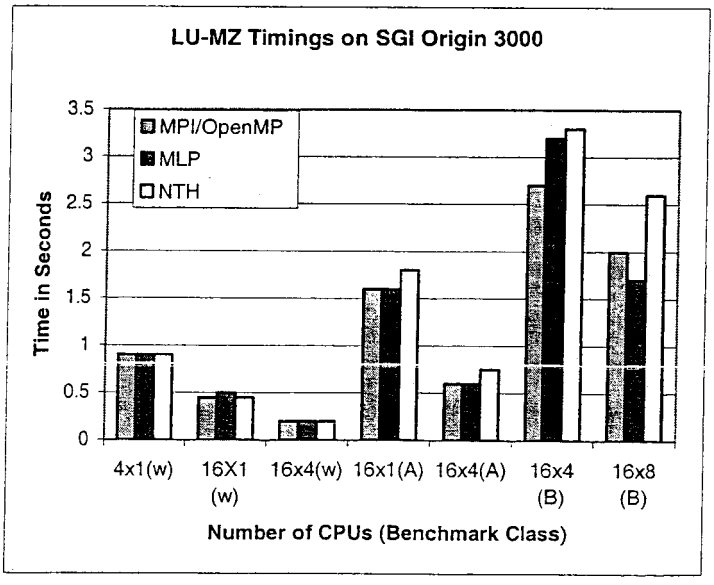

Figure 13: Timings for 20 iterations of different classes of the LU-MZ benchmark

nested OpenMP code, respectively. The best timings were achieved by the same combinations in the hybrid and the nested OpenMP codes. In the case of the LU-MZ the nested OpenMP code does not achieve the performance of the hybrid implementations. Figure 14 shows the scalability of the nested OpenMP code for problem size of Class B. The performance of MPI/OpenMP and nested OpenMP is compared to the Gigaflop rate that would be achieved in case of linear speed-up. The major difference between LU-MZ and the two previous benchmarks is, that LU-MZ requires explicit thread synchronization in order to achieve pipelined execution on the inner level. This is implemented by usage of a vector containing synchronization variables. The vector is shared by all threads across all groups, even though only the threads within one group need to synchronize with their neighbors. We are currently investigating whether the access to the shared synchronization vector causes the performance decrease, and how to remedy the problem.

The timings for LU-MZ Class A on the IBM Regatta are shown in Figure 15. Due to the small number of CPUs on a single node, the scalability problem observed on the SGI Origin does not show. Hybrid and nested parallelism are advantageous for more than 16 CPUs.

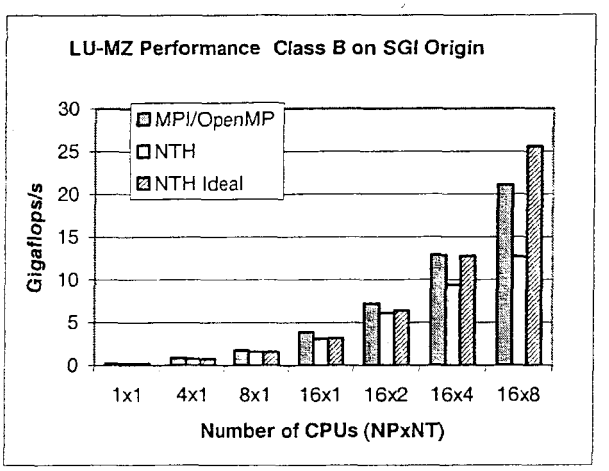

Figure 14: Performance of the LU-MZ Class B benchmark in terms of the total number of Gigaflops per second. *NTH ideal* indicates the ideal Gigaflop rate of the nested OpenMP code in case of linear speed-up

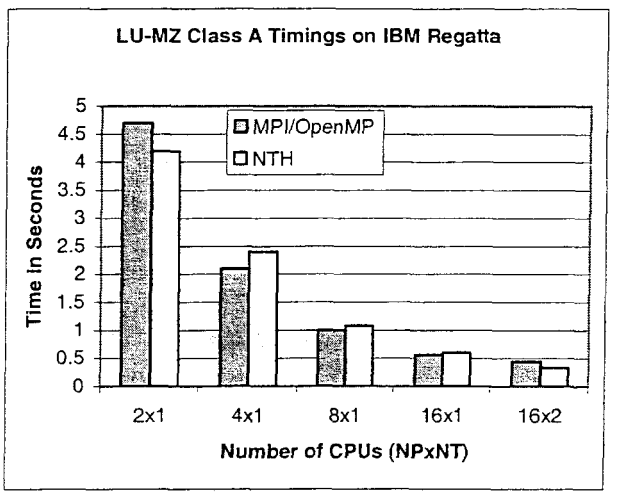

Figure 15: Timings reported are for 20 iterations. Reported are the best timings for each implementation over all numbers of processes or groups

5 Related Work

Most current commercial and research compilers mainly support the exploitation of a single level of parallelism and special cases of nested parallelism (e.g. double perfectly nested loops as in the SGI MIPSpro compiler [15]). The KAI/Intel compiler offers, through a set of extensions to OpenMP, work queues and an interface for inserting application tasks before execution (WorkQueue proposal [18]). The KAI/Intel proposal mainly targets dynamic work generation schemes (recursions and loops with unknown loop bounds). At the research level, the Illinois--Intel Multithreading library [6] provides a similar approach based on work queues. In both cases, there is no explicit (at the user or compiler level) control over the allocation of threads so they do

not support the logical clustering of threads in the multilevel structure, which we think is necessary to allow good work distribution and data locality exploitation. The IBM XL [21] Fortran compiler supports nested parallelism. The execution environment provides a pool of threads from which any parallel region can take some for parallel execution. The user has the possibility to limit the number of threads on the outer level or parallelism by using the NUM_THREADS clause in the PARALLEL directive. We have discussed the problems that may result from this approach in subsection 3.2.

There are a number of papers reporting experiences in combining multiple programming paradigms to exploit multiple levels of parallelism (e. g. [19]). However, there is not much experience in the parallelization of applications with multiple levels of parallelism simply using OpenMP. Implementation of nested parallelism by means of controlling the allocation of processors to tasks in a single-level parallelism environment is discussed in [4]. The authors show the improvement due to nested parallelization. The performance of code containing automatically generated nested OpenMP directives is discussed in [13].

6 Conclusions and Future Work

A nested OpenMP implementation of the multi-zone versions of the NAS Parallel Benchmarks was developed. The nested OpenMP code makes use of the NanosCompiler extensions to OpenMP, allowing the creation of thread groups and load-balancing among the thread groups. The NanosCompiler was then used to evaluate the performance of the nested OpenMP code on two different hardware platforms. The performance was compared to corresponding hybrid implementations of the benchmarks using the MPI/OpenMP and the MLP programming paradigms. For two of the three benchmarks the performance of the OpenMP code was comparable to the hybrid implementations. The scalability of the third benchmark, which requires pipelined thread execution within a thread group, was limited. The reason for this is currently under investigation.

The first conclusion of the study is that the OpenMP paradigm allowed a very rapid development of the parallel code. The second observation is that the thread groups implementation in the NanosCompiler and runtime was crucial to obtaining good performance. The reason is that the implementation guarantees the same mapping of kernel threads to OpenMP threads in all parallel regions, both at the outer and inner levels. This improves memory access time and results in performance levels that are comparable to the hybrid versions. Another important feature of the NanosCompiler is the possibility to assign weights to the thread groups in order to achieve a well balanced work load distribution.

We plan to conduct further case studies to compare the performance of parallelization based on nested OpenMP directives with hybrid and pure message passing parallelism. We will consider other hardware platforms, larger benchmark classes, and full-scale applications.

Acknowledgments

This work was supported by NASA contract DTTS59-99-D-00437/A61812D with Computer Sciences Corporation/AMTI and Corporation and by the Spanish Ministry of Science and Technology under contract CICYT 98-511.

References

[1] E. Ayguade, X. Martorell, J. Labarta, M. Gonzalez and N. Navarro, *Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study,* Proc. Of the 1999 International Conference on Parallel Processing, Ajzu, Japan, September 1999.

[2] D. Baily, E. Barscz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Fredrickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, S. Weeratuga. *The NAS Parallel Benchmarks*, NAS Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994.

[3] D. Bailey, T. Harris, W. Saphir, R. Van der Wijngaart, A. Woo, and M. Yarrow, *The NAS Parallel Benchmarks 2.0, RNR-95-020*, NASA Ames Research Center, 1995. NPB2.3, http://www.nas.nasa.gov/Software/NPB/.

[4] R. Blikberg and T. Sorevik. *Nested Parallelism: Allocation of Processors to Tasks and OpenMP Implementation.* 2nd European Workshop on OpenMP. Edinburgh. September 2000.

[5] M. J. Djomehri, R. Biswas, M. Potsdam, R.C. Strawn, *An Analysis of Peformance Enhancement Techniques for Overset Grid Applications*, Proceedings of the 17th International Parallel and Distributed Processing Symposium, Nice, France, 2003.

[6] M. Girkar, M. R. Haghighat, P. Grey, H. Saito, N. Stavrakos and C.D. Polychronopoulos. *Illinois-Intel Multithreading Library: Multithreading Support for Intel. Architecture--based Multiprocessor Systems.* Intel Technology Journal, Q1 issue, February 1998.

[7] M. Gonzalez, E. Ayguadé, X. Martorell and J. Labarta. *Defining and Supporting Pipelined Executions in OpenMP.* 2nd International Workshop on OpenMP Applications and Tools. July 2001.

[8] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver. *NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP.* Concurrency: Practice and Experience. Special issue on OpenMP. vol. 12, no. 12. pp. 1205-1218. October 2000.

[9] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguadé, J. Labarta and N. Navarro. *OpenMP Extensions for Thread Groups and Their Run-time Support.* 13th International Workshop on Languages and Compilers for Parallel Computing (LCPC'2000), New York (USA). pp. 317-331. August, 2000.

[10] H. Jin, M. Frumkin, and J. Yan, *The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance, NAS Technical Report NAS-99-011,* 1999.

[11] H. Jin, G. Jost, *Performance Evaluation of Remote Memory Access Programming on Shared Memory Parallel Computer Architectures*, NAS Technical report NAS-03-001, NASA Ames Research Center, Moffett Field, CA, 2003.

[12] H. Jin, R. F. Van der Wijngaar, *Performance Characteristics of the Multi-Zone NAS Parallel Benchmarks*, to appear

[13] H. Jin, G. Jost, E. Ayguade, M. Gonzalez, X. Martorell, *Automatic Multilevel Parallelization Using OpenMP,* Scientific Programming Vol. 11, No 2, 2003.

[14] X. Martorell, E. Ayguadé, N. Navarro, J. Corbalan, M. Gonzalez and J. Labarta. *Thread Fork/join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors.* 13th International Conference on Supercomputing (ICS'99), Rhodes (Greece). pp. 294-301. June 1999.

[15] MIPSPro 7 Fortran 90 Commands and Directives Reference Manual 007-3696-03

[16] MPI 1.1 Standard, http://www-unix.mcs.anl.gov/mpi/mpich.
[17] OpenMP Fortran Application Program Interface, http://www.openmp.org/.
[18] S. Shah, G. Haab, P. Petersen and J. Throop. *Flexible Control Structures for Parallelism in OpenMP*. In 1st European Workshop on OpenMP, Lund (Sweden), September 1999.
[19] J. Taft, "*Achieving 60 GFLOP/s on the Production CFD Code OVERFLOW-MLP*", Parallel Computing, 27 (2001) 521.
[20] R. F. Van Der Wijngaart, H. Jin, *"NAS Parallel Benchmarks, Multi-Zone Versions,"* NAS Technical Report NAS-03-010, NASA Ames Research Center, Moffett Field, CA, 2003.
[21] XL Fortran for AIX User's Guide Version 8.11, IBM sc09-4948-01, IBM Corp. Second Edition, June 2003, http://www-3.ibm.com/software/awdtools/fortran/xlfortran/library