Employing nested OpenMP for the parallelization of multi-zone computational fluid dynamics applications

Eduard Ayguade\textsuperscript{a,}\textsuperscript{*}, Marc Gonzalez\textsuperscript{a}, Xavier Martorella\textsuperscript{a}, Gabriele Jost\textsuperscript{b,1}

\textsuperscript{a}Centre Europeu de Parall\'elisme de Barcelona, Computer Architecture Department (UPC), cr. Jordi Girona 1-3, Modul D6, 08034, Barcelona, Spain

\textsuperscript{b}NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA

Received 15 July 2004; received in revised form 15 February 2005; accepted 6 June 2005

Available online 15 August 2005

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 that 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 (MPI+OpenMP and MLP) and discuss OpenMP implementation issues that affect the performance of multi-level parallel applications.

© 2005 Elsevier Inc. All rights reserved.

Keywords: OpenMP; Parallel programming models; Nested parallelism; NAS benchmarks

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 [12], which provides the functionality for process communication and synchronization. OpenMP [13] was introduced as an industrial standard for shared-memory programming with directives. The directives support loop level parallelization. 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 [15] model, which uses OpenMP at the innermost level, developed at NASA Ames. However, there is not much experience in the parallelization of applications with multiple levels of parallelism using OpenMP only. Avoiding mixed programming models simplifies the task of parallel programming when multiple levels of parallelism needs to be exploited.

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 [4], 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 code containing OpenMP directives and generates plain Fortran-77 code with calls to the NthLib thread library [10]. 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 allowing 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.

In this paper we show the application nested parallelism in multi-zone codes 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 yield the name multi-zone code. It is common to solve the flow equations independently within each zone. After that, 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. This computation is performed inside a time step loop that is repeated for a large number of times.

Fig. 1 shows the nested parallelism that may exist in multi-zone codes. In Fig. 1a all zones have the same size. In this case, just exploiting the parallelism across zones may not be sufficient to feed a large number of processors since the number of zones is not usually very large. Just exploiting parallelism inside zones may be too fine grained when the number of processors is large. So in this situation, the programmer needs to distribute the total number of processors (e.g. $N \times M$) by assigning a number of processors to the outer level (e.g. $N$) and then assign the same number of processors to exploit parallelism inside zones (e.g. $M$). When zones have different sizes (see Fig. 1b) it is necessary to assign a number of processors to exploit the innermost level of parallelism proportional to the size of each zone. This results in a well-balanced parallel execution. 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]. The NPB Multi-Zone (NPB-MZ) are described in [16]. 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 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.

When a thread in a team encounters a new PARALLEL construct, it creates a new team and it becomes its master thread. Nesting PARALLEL regions is the mechanism for creating multilevel parallel applications in OpenMP. In order to control the number of threads that is used in each PARALLEL region, OpenMP v2.0 provides the NUM_THREADS clause. The information provided in this clause overrides the number of threads that is specified either by theOMP_NUM_THREADS environment variable or by invocations to theomp_set_num_threads() intrinsic routine in the serial part of the application. IfNUM_THREADS is not used, all parallel regions create teams with as many threads as specified byOMP_NUM_THREADS oromp_set_num_threads().

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 PARALLEL construct defining groups is encountered, a new team of threads is created. 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)

Different formats for the GROUPS clause argument gspec are allowed [5]. The simplest specifies the number of groups and performs an equal partition of the total number of threads to the groups:

\[ gspec = \text{ngroups} \]

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. Another possible format is

\[ gspec = \text{ngroups}, \text{weight} \]

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 can either be computed from information available within the application itself (for instance iteration space, data size or computational complexity) or from dynamically collected information about execution unbalance (e.g. execution time of the zones). In this paper, the vector is always computed based on the size of the zones assigned to each group of threads.

The GROUPS clause allows more modularity in the nested parallel application than the NUM_THREADS clause, as we will show in Section 3.2.

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 CFD applications. Multi-zone versions of the NAS Parallel Benchmarks LU, BT, and SP were 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:

- LU is a simulated CFD application that uses symmetric successive overrelaxation (SSOR) method to solve a seven block diagonal system resulting from finite difference discretization of the NavierStokes equations in 3D by splitting to into block Lower and Upper triangular systems.

- BT is a simulated CFD application that uses an implicit algorithm to solve three-dimensional (3D) compressible NavierStokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the $x$, $y$, and $z$ dimensions. The resulting systems are BlockTridiagonal of $5 \times 5$ blocks and are solved sequentially along each dimension.

- Finally, SP is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a BeamWarming approximate factorization that decouples the $x$, $y$, and $z$ dimensions. The resulting system has scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.

Exchange of boundary values takes place after each time step. A detailed discussion of the NPB-MZ can be found in [16]. Fig. 2a shows the general structure for all benchmarks. 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

The source code distribution of the NPB-MZ includes two different hybrid implementations, as shown in Fig. 2b. 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 [6]. If the number of MPI processes is one, then we are just exploiting
the innermost level of parallelism. If the number of threads per process is one, then we are just exploiting the outermost level. These two particular solutions will show the benefits of exploiting nested parallelism by assigning the appropriate number of resources to each level of parallelism.

The second hybrid implementation that is part of the NPB-MZ is based on the MLP programming model developed by Taft [15] 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 approach was developed for ccNUMA architectures and explicitly takes advantage of the availability of shared memory. A shared memory arena, which is accessible by all processes, is required. 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 [7]. 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.

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 [8].

3.2. The nested OpenMP implementations

The nested OpenMP implementation is currently not part of the NPB-MZ distribution and has been developed by the authors of this paper. This implementation combines a coarse-grained parallelization (inter-zone) and parallelization within the zones (intra-zone), 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 (pzone id) and how many threads are allocated to each group (pn thr). 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 grps) is controlled by an environment variable. Fig. 3a 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).

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
do step = 1, itmax
   call exch_qbc(u, qbc, nx, ...)
C$OMP PARALLEL
C$OMP PRIVATE(iam, zone, ...)
C$OMP NUM_THREADS(num_grps, pn_thr)
   iam = omp_get_thread_num()
   do zone = 1, num_zones
      if (iam .eq. pzone_id(zone)) then
         call ssor(u, rsd, ...)
      end if
   end do
C$OMP END PARALLEL
end do
...
subroutine ssor(u, rsd, ...)
...
integer pn_thr (num_zones)
common /thr_mapping/ pn_thr
C$OMP PARALLEL DEFAULT(SHARED)
C$OMP& PRIVATE(m,i,j,k,...)
C$OMP& NUM_THREADS(pn_thr)
   do k = 2, nz-1
     C$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
   C$OMP END DO nowait
   end do
   ...
C$OMP END PARALLEL
...

a)
Fig. 3. Parallelization of LU-MZ using (a) the Nanos GROUPS clause and (b) the NUM_THREADS clause.

... described above. However, it requires that the programmer explicitly controls the allocation of threads at each level of parallelism, as shown in Fig. 3b (equivalent to 3a). This implies 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. In addition, if more levels of parallelism were available, coding the allocation of threads would be painful using NUM_THREADS. The version employing the NanosCompiler GROUPS clause extension is more modular since the context is implicit in the OpenMP runtime support 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 Fig. 3b this would be the number specified by the NUM_THREADS clause. This is the case for example in the runtime system of the IBM XL compiler [17]. However, there is no guarantee that a particular thread is always executed on the same processor, so that data locality is not necessarily 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. In other words, the definition of GROUPS is more static than the definition of NUM_THREADS, which we consider more dynamic.
4. Timing results

We ran the BT-MZ, LU-MZ, and SP-MZ benchmarks of problem classes W, A, and B. The aggregate sizes for all benchmarks are $64 \times 64 \times 8$ grid points (Class W), $128 \times 128 \times 16$ grid points (Class A) and $304 \times 208 \times 17$ grid points (Class B).

Our tests were executed on two hardware 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 cc-NUMA 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 [7]. The MPI-Spro 7.4 Fortran Compiler [11] 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. Executions were done using CPU sets to limit the interaction with other applications simultaneously running on the machine.

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 supporting 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 Fig. 3b) 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. All the executions were done with dedicated machine.

In all charts, we use the following notation to refer to the different versions:

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI + OpenMP</td>
<td>Hybrid version implemented with MPI and OpenMP.</td>
</tr>
<tr>
<td>MLP</td>
<td>Hybrid version implemented using the MLP approach.</td>
</tr>
<tr>
<td>NTH</td>
<td>Nested OpenMP implementation using the NanosCompiler and the <code>NUM_THREADS</code> clause.</td>
</tr>
<tr>
<td>IBM Nested</td>
<td>Nested OpenMP implementation using the native IBM compiler and the <code>NUM_THREADS</code> clause.</td>
</tr>
<tr>
<td>NP × NT</td>
<td>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 or threads used at the outer parallel level.</td>
</tr>
</tbody>
</table>

### Fig. 4. Timings for 20 iterations of BT-MZ.

![Fig. 4. Timings for 20 iterations of BT-MZ.](image)

### Fig. 5. Timings for 20 iterations of BT-MZ with 16 processors.

![Fig. 5. Timings for 20 iterations of BT-MZ with 16 processors.](image)

#### 4.1. The BT-MZ benchmark

The number of zones grows with the problem size. The number of zones is $4 \times 4$ for Class W and A, and $8 \times 8$ 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.

Fig. 4 shows results for the hybrid MPI/OpenMP version 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 achieving the scalability of the hybrid version. The reason for this performance degradation is that the IBM runtime implements nested parallelism with a pool of threads that does not exploit data locality. In order to show this, we have analyzed the particular execution of the NAS BT-MZ benchmark using the following parameters: two threads at the outer level of parallelism (groups) and `OMP_NUM_THREADS` set to 8, so each group should have 4 active threads. Then, we collected traces consisting of which pairs of physical processors/threads are executing inside each group and the thread identifier assigned by OpenMP. We have observed that in several executions of
the same parallel region, the same physical processor can be executing different user-level threads (pthreads in the IBM implementation), in different groups, and that each pthread can be assigned different OpenMP identifier, so it can get different sets of iterations. For instance, this portion of the trace:

<table>
<thead>
<tr>
<th>Group</th>
<th>physcpu</th>
<th>pthread</th>
<th>omp_get_thread_num</th>
<th>where/what</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>1800</td>
<td></td>
<td>executing z_solve with 4 cpus</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1800</td>
<td>3</td>
<td>iter 13</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1800</td>
<td>3</td>
<td>iter 14</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1800</td>
<td>3</td>
<td>iter 15</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>1800</td>
<td>3</td>
<td>iter 16</td>
</tr>
</tbody>
</table>

indicates that the physical processor 3 is executing the pthread identified with 1800, and that pthread can be dynamically assigned to groups 0 and 1 by the runtime system. This causes a break in the locality because groups 0 and 1 touch different memory regions. It can even occur than the same pthread is assigned a different OpenMP identifier in two executions of the same parallel region, as shown in the following section of the trace:

<table>
<thead>
<tr>
<th>Group</th>
<th>physcpu</th>
<th>pthread</th>
<th>omp_get_thread_num</th>
<th>where/what</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>18</td>
<td>1543</td>
<td></td>
<td>executing z_solve with 4 cpus</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>1543</td>
<td>3</td>
<td>iter 13</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>1543</td>
<td>3</td>
<td>iter 14</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>1543</td>
<td>3</td>
<td>iter 15</td>
</tr>
<tr>
<td>0</td>
<td>18</td>
<td>1543</td>
<td>3</td>
<td>iter 16</td>
</tr>
</tbody>
</table>

Observe that the group and the physical processor to which the pthread 1543 is assigned are the same, but as the OpenMP identifier is changed from 3 to 2 by the runtime, the same physical processor gets a different set of iterations, causing again a break in the locality.

Timings for the execution of the BT-MZ benchmark on the IBM Regatta with 16 processors and different allocations of threads to the outer and inner levels are shown in Fig. 5. 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. Notice that the performance of the IBM Nested implementation is best when all threads are used on the inner level because in this case the same threads are always used to execute the inner level of parallelism. The hybrid MPI/OpenMP version behave better when nested parallelism is used, taking advantage of load balancing on the inner level of parallelism and data locality.

Fig. 6 shows the speedup achieved by the hybrid MPI/OpenMP and the NTH versions for BT-MZ class A on the IBM Regatta system. 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
BT-MZ Class A Timings on IBM Regatta

![BT-MZ Class A Timings](image)

Fig. 6. Timings for the complete execution of BT-MZ class A on the IBM Regatta system.

BT-MZ Timings, SGI Origin 3000

![BT-MZ Timings, SGI Origin 3000](image)

Fig. 7. Timings for 20 iterations of BT-MZ.

process or group in the timings charts. Although the performance of the NTH version is slightly worse than the performance of the hybrid MPI/OpenMP version, the behavior is the same. The runtime system used to support the NTH versions is not as well tuned as the native runtime system provided by IBM and used in the hybrid MPI/OpenMP. In both cases, notice that the parallel versions that just exploit one level of parallelism (i.e. $N \times 1$ or $1 \times N$) are always worse than assign resources to the two levels of parallelism.

The MIPSplo compiler and runtime environment on the SGI Origin do not support nested parallelism. The execution times of the hybrid MPI/OpenMP and the NTH versions are shown in Fig. 7. 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 guarantees 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.

Fig. 8 shows the impact of different combinations 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. Since the number of threads per process or group varies, we report the average number of threads per process.

To demonstrate the scalability of the different implementations on the SGI Origin 3000 the GigaFlop rate as reported by the benchmark is shown in Fig. 9. The Class B perfor-
mance for the number of processes or thread groups that produced the best results is reported. This 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.

4.2. The SP-MZ benchmark

In SP-MZ 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 $4 \times 4$ for Class W and A, and $8 \times 8$ for Class B. The computations are naturally load balanced on the coarse level. Timings for the different implementations and different benchmark classes on the SGI Origin are shown in Fig. 10. 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. Fig. 11 shows the timings for Class B on different process or group and thread combinations.

On the IBM Regatta it was advantageous to use multiple threads per process or group for the Class A benchmark. The timings for both MPI/OpenMP and NTH are comparable, as shown in Fig. 12. Again, notice that when the number of processors is small and less than the number of zones, the best parallelization strategies are those that assign all resources to the outer level. However, when the number of processors is larger than the number of zones, assigning threads at the innermost level is worth. Notice that just exploiting the innermost level of parallelism (which is independent of the number of zones) is the worst solution in almost all cases, specially when the number of processors is higher than 4.

4.3. The LU-MZ benchmark

In LU-MZ the number of zones is $4 \times 4$ 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. In the nested OpenMP code the number of thread groups is limited to 16.

The timings for the SGI Origin are shown in Fig. 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 nested OpenMP code, respectively. The best timings were achieved by the same combinations $NP \times NT$ in the hybrid and the nested OpenMP codes. In the case of LU-MZ the nested OpenMP code does not achieve the performance of the hybrid implementations.

The major difference between LU-MZ and the two previous benchmark implementations is that both BT-MZ and SP-MZ perform one time step before timing of the actual iteration loop. This ensures efficient data placement in case of a first-touch data-placement policy. This is not the case for LU-MZ. While it does not effect the hybrid codes, the lack of touching the data before the start of the iteration yields to a dramatic increase in time for the first iteration in the nested OpenMP code, which had a significant impact due to the fact that we were only timing the first 20 iterations. We have modified the nested OpenMP code to include an iteration to touch the data appropriately, analogous to BT-MZ and SP-MZ. Fig. 14 shows the scalability of the nested OpenMP and the MPI/OpenMP code for problem size of Class B. The figure includes the performance achieved by the modified code that touches data before the start of the iteration: column NTH (touch data).

The timings for LU-MZ Class A on the IBM Regatta are shown in Fig. 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.
distribution and data locality exploitation. The IBM XL [17] 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 Section 3.2.

There are a number of papers reporting experiences in combining multiple programming paradigms to exploit multiple levels of parallelism (e.g. [15]). Experiences on employing multiple level of parallelism in OpenMP are reported in [1]. Implementation of nested parallelism by means of controlling the allocation of processors to tasks in a single-level parallelism environment is discussed in [3]. The authors show the improvement due to nested parallelization. The performance of code containing automatically generated nested OpenMP directives is discussed in [9].

6. Conclusions and future work

This paper compares the performance of hybrid MPI/OpenMP with pure OpenMP implementations for the recently released MZ versions of NAS benchmarks. A nested OpenMP implementation of the multi-zone versions of the NAS Parallel Benchmarks has been developed in order to do the comparison directly from the sequential version. 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: SGI Origin 3000 and IBM Regatta. The performance was compared to corresponding hybrid implementations of the benchmarks using the MPI/OpenMP and the MLP programming paradigms. For all three benchmarks, the performance of the OpenMP code was comparable to that of the hybrid implementations. On the SGI Origin the LU-MZ benchmark required touching the data before the start of the iteration in order to achieve the performance of the hybrid codes. This touching has been included in the latest release of the benchmark suite.
Exploiting nested parallelism in this kind of applications is necessary in two situations. First when the number of processors is too high to exploit a single level of parallelism or the granularity of the work results small. And second, when the exploitation of one level of parallelism results in unbalanced executions. In this case, nesting helps to distribute processors among the different levels of parallelism and obtain a well balanced parallel execution.

Several conclusions can be drawn from the study in this paper. The first conclusion is that the nested OpenMP paradigm allowed a very rapid development of the parallel code. MPI usually requires an explicit restructuring of the code in order to expose parallelism and insert communication routines at the appropriate points. This is not necessary in the OpenMP version which simply adds directives to the sequential version.

The second conclusion from this study is that the thread groups implementation in the NanosCompiler and runtime system 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. We have shown that the IBM runtime that supports nested parallelism using a pool of threads (and thus does not guarantee the same mapping of threads in different parallel regions or multiple instances of the same parallel region) performs worse than the hybrid MPI/OpenMP.

Finally, we have shown that with a simple extension to OpenMP the programmer is able to map resources (processors) to the different levels of parallelism and achieve a well balanced work load distribution. The weight vector in the Groups clause can be computed at the beginning of the application using for example the size of the data structures used in the computations (i.e. size of the zones in the NAS MZ benchmarks). The computation of the weight vector could also be done dynamically using time measurements. For example, starting from a simple equi-distribution of zones to processors at the outermost level, we could measure the execution time unbalance and compute the values in the weight vector. We are currently investigating this dynamic composition of thread groups, totally freeing the programmer from the task of computing the weight vector.

Acknowledgements

This work has been supported by NASA contract DTSS59-99-D-00437/A61812D with Computer Sciences Corporation/AMTI, by the CEPBA-IBM Research Institute (CIRI), by the Spanish Ministry of Science and Technology of Spain under contracts TIC2001-0995-C02-01 and TIN2004-07739-C02-01, and by the FET European Project POP. We thank Jesús Labarta for his comments on the experiments given in this paper and Bernd Mohr and FZ Juelich Center in Germany for providing the access to the IBM platform.

References

[7] H. Jin, G. Jost, Performance evaluation of remote memory access programming on shared memory parallel computer architectures,
Eduard Ayguade received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the Technical University of Catalonia (UPC). Since 1987 he has been lecturing on computer organization and architecture and optimizing compilers. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. His research interests cover the areas of parallelizing compilers and runtime systems for high-performance multiprocessor systems. He has published more than 125 papers in these topics and participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs. He is one of the project managers at the CEPBA-IBM Research Institute.

Marc Gonzalez received the Engineering and Ph.D. degrees in Computer Science in 1996 and 2003, respectively, from the Technical University of Catalonia (UPC) in Barcelona (Spain). He is assistant professor in the Computer Architecture Department lecturing in the Computer Science Faculty since 2000. His research interests mainly include parallelizing compilers and runtime systems for high-performance multiprocessor systems. He has been involved in recent long-term research projects in the framework of the European Union ESPRIT and IST programs.

Xavier Martorell received the Engineering degree in Computer Science in 1991 and the Ph.D. degree in Computer Science in 1999, both from the Technical University of Catalonia (UPC). Since 1992 he has been lecturing on operating systems and parallel run-time systems. Currently, and since 2001, he is an associate professor of the Computer Architecture Department at UPC. His research interests cover the areas of operating systems, run-time systems and compilers for high-performance multiprocessor systems. He has participated in several long-term research projects with other universities and industries, mostly in the framework of the European Union ESPRIT and IST programs.

Gabriele Jost received her Doctorate in Applied Mathematics from the University of Goettingen, Germany, in 1986. She then worked in software development, benchmarking, and application optimization for various vendors of high performance computer architectures, including vector and parallel computers. She is now a research scientist in the parallel tools group at the NASA Ames Research Center in Moffett Field, California, USA. Her current projects are performance analysis, automatic parallelization and optimization, and the study of parallel programming paradigms.