Scientific Programming 11 (2003) 177–190
IOS Press

Automatic multilevel parallelization using OpenMP[1]

Haoqiang Jin[a], Gabriele Jost[a,*,**], Jerry Yan[a], Eduard Ayguade[b], Marc Gonzalez[b] and Xavier Martorell[b]
[a]*NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000, USA*
[b]*Centre Europeu de Parallelism de Barcelona, Computer Architecture Department* (*UPC*), *cr.Jordi Girona 1-3, Modul D6,08034 – Barcelona, Spain*

Abstract. In this paper we describe the extension of the CAPO parallelization support tool to support multilevel parallelism based on OpenMP directives. CAPO generates OpenMP directives with extensions supported by the NanosCompiler to allow for directive nesting and definition of thread groups. We report some results for several benchmark codes and one full application that have been parallelized using our system.

1. Introduction

Parallel architectures are an instrumental tool for the execution of computational intensive applications. Simple and powerful programming models and environments are required to develop and tune such parallel applications. Current programming models offer either library-based implementations (such as MPI [16]) or extensions to sequential languages (directives and language constructs) that express the available parallelism in the application, such as OpenMP [20].

The compiler directives provided by the OpenMP standard support a fork/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. OpenMP allows the nesting of parallel regions, but does not clearly address the issue of how to exploit it efficiently.

It has become increasingly popular to combine several programming models to exploit multiple levels of parallelism but at this point there are few reported results on experiences using mulitlevel OpenMP parallelism. The lack of compilers supporting this feature has been the main cause of this problem. In turn, since the benefits of nested parallel regions are not clear, there is little incentive for compiler developers to support it.

Some research platforms, such as the OpenMP NanosCompiler [9], 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 [17] (currently implemented for the SGI Origin). In contrast to the SGI MP library, NthLib allows for multilevel parallel execution such that inner parallel constructs are not being serialized. The NanosCompiler 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 way to multilevel parallelism.

We have extended the automatic parallelization tool CAPO [13], to allow for the generation of nested OpenMP parallel constructs and NanosCompiler extensions in order to support multilevel shared memory parallelization. CAPO automates the insertion of OpenMP

[1]A Preliminary version of this paper was presented at the 3rd European Workshop on OpenMP (EWOMP01).

*Corresponding author: GJost@nAS.nASA.gov.
**The author is an employee of Computer Sciences Corporation.

ISSN 1058-9244/03/$8.00

directives with nominal user interaction to facilitate parallel processing on shared memory parallel machines. It is based on CAPTools [11], a semi-automatic parallelization tool for the generation of message passing codes, developed at the University of Greenwich. By being able to generate nested directives automatically in a reasonable amount of time we hope to gain a better understanding of performance issues and the needs of application programs when it comes to exploiting multilevel parallelism. The goal of our evaluation is threefold: We want to study the performance impact of nested OpenMP parallelization, we discuss potential extensions to the OpenMP standard, and we identify requirements for tools to automate the multilevel parallelization process.

The paper is organized as follows: Section 2 summarizes the NanosCompiler extensions to the OpenMP standard. Section 3 discusses the extension of CAPO to generate multilevel parallel codes. Section 4 presents case studies on several benchmark codes and one full application. Section 5 gives an overview on related work and Section 6 concludes with describing the current project status and future plans.

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 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 [10]. 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 CAPO parallelization support tool

The main goal of developing parallelization support tools is to eliminate much of the tedious and sometimes error-prone work that is needed for manual parallelization of serial applications. With this in mind, CAPO [13] was developed to automate the insertion of OpenMP compiler directives with nominal user interaction. This is achieved largely by use of the very

accurate interprocedural analysis module from CAPTools [11]. Furthermore CAPO provides a directive browser to allow the user to examine and optimize the directives automatically placed within the code. CAPTools provides a fully interprocedural and value-based dependence analysis engine [14] and has successfully been used to parallelize a number of mesh-based applications for distributed memory machines.

3.1. Single level parallelization

After an extensive dependence analysis CAPO inserts OpenMP directives into sequential Fortran code. Details about this process can be found in [13]. The three main steps to generate the directives can be summarized as follows:

1) *Identification of parallel loops and parallel region:* Based on the dependence analysis information loops are identified as serial or parallel. The outermost parallel loops are considered for parallelization. The dependence analysis is interprocedural, and the parallel regions are defined as high up in the call tree as the analysis results will allow it to achieve an efficient placement of the directives. If the outermost loop contains prohibitive dependences, the next nesting level is considered for the insertion of directives.
2) *Optimization of parallel regions and parallel loops:* The goal of this phase is to lower the fork-and-join overhead associated with starting parallel regions and the thread synchronization costs. This is achieved by merging parallel regions whenever possible. In addition, the synchronization between successive parallel loops is minimized by using the NOWAIT clause if the dependence analysis shows that the loops can correctly execute asynchronously.
3) *Code transformation and insertion of OpenMP directive:* This is the final stage where the call graph is traversed to place OpenMP directives within the code. This includes the identification of variable types, such as `SHARED`, `PRIVATE`, and `REDUCTION`. In addition possible `THREADPRIVATE` common blocks are identified and proper directives are inserted.

The transformations described above are portable to all platforms where OpenMP parallelization is supported. In addition to this, CAPO provides the possibility to generate some platform dependent extensions. One of the extensions supported by CAPO is the SGI NEST clause. Although the SGI compiler does not support nested parallelism, the user can exploit parallelism across multiple loop nests in a limited manner. The SGI compiler accepts the NEST clause on the OMP DO directive [18]. The NEST clause requires at least 2 variables as arguments to identify indices of subsequent DO-loops. The identified loops must be perfectly nested and no code is allowed between the identified DO statements and the corresponding END DO statements. The NEST clause on the OMP DO directive informs the compiler that the entire set of iterations across the identified loops can be executed in parallel. The compiler can then linearize the execution of the loop iteration and divide them among the available single level of threads. This is not nested parallelism but merely a distribution of work in multiple dimensions within a single level of parallelism. CAPO has the capability to identify suitable loop nests and generate the SGI NEST clause. We have extended this feature of CAPO to support true nested parallelism.

3.2. Extension to multilevel parallelization

According to the OpenMP standard if a thread in a team executing a parallel region encounters another parallel region, it creates a new team and it becomes the master of that new team. Our extension to OpenMP multilevel parallelism makes use of the extensions offered by the NanosCompiler. Currently, we limit our approach to only two-level loop parallelism, which is of more practical use. The approach to automatically exploit two-level parallelism is extended from the single level parallelization and is illustrated in Fig. 1. After performing the data dependence analysis the approach can be summarized in the following four steps:

1) *First-level loop analysis:* This is essentially the combination of the first two stages in the single level parallelization where parallel loops and parallel regions are identified and optimized at the outermost loop level.
2) *Second-level loop analysis:* This step involves the identification of parallel loops and parallel regions nested inside the parallel loops that were identified in Step 1. These parallel loops and parallel regions are then optimized as before but limited to the scope defined by the first level.
3) *Second-level directive insertion:* This includes code transformation and OpenMP directives insertion for the second level. This step is performed before inserting any directives on the first-

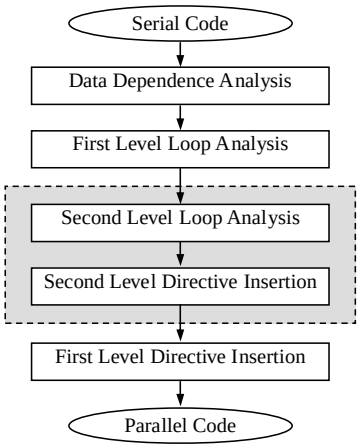

Fig. 1. Steps in multilevel parallelization.

level. It ensures that a consistent picture is maintained for any variables and code that may be changed during the parallelization process.

4) *First-level directive insertion:* Lastly code transformation and OpenMP directives insertion are performed for the outer level parallelization. All the transformations of the last stage of the single level parallelization are being performed, with the exception that we disallow the `THREADPRIVATE` directive. Compared to single level parallelization, the two-level parallelization process requires the additional steps indicated in the dash box in Fig. 1.

3.3. Implementation consideration

In order to maintain consistency during necessary code transformations of the parallelization process we need to update data dependences properly. Consider the example where CAPO transforms an array reduction into updates to a local variable. This is followed by an update to the global array in a `CRITICAL` section to work around the limitation on array reductions in OpenMP v1.x. In this case the data dependence graph needs to be updated to reflect the change due to this transformation

When nested parallel regions are considered, the scope of the `THREADPRIVATE` directive is not clear any more, since a variable may be threadprivate for the outer nest of parallel regions but shared for the inner parallel regions, and the directive cannot be bound to a specific nest level. The OpenMP specification does not properly address this issue. Our solution is to disallow the `THREADPRIVATE` directive when nested parallelism is considered. In case that a variable in a non-threadprivate common block needs to be privatized and causes a usage conflict, the common block variables are added to the argument list of and removed properly from the common blocks inside the relevant subroutines.

CAPO detects opportunities for software-pipelined execution of loops where data dependences prevent parallelization (see [13]). Such loops are enclosed by a parallel region. The iteration space of the loops is divided up among the threads using the OMP DO directive. The threads then explicitly synchronize their execution with their neighbors. This is discussed in greater detail in Section 4.2 and an example for a one-dimensional pipeline is shown in Fig. 5. Setting up a two-dimensional pipeline would involve synchronization of threads from two different nest levels. We will discuss the problem of two-dimensional pipelining in one of our case studies in Section 4.2.

One of the contributions by the NanosCompiler to support nested directives is the GROUPS clause, which can be used to define the number of thread groups to be created at the beginning of an outer-nest parallel region. In our implementation, the GROUPS directive (containing a single shared variable '`ngroups`') is generated for all the first-level parallel regions. The `ngroups` variable is placed in a common block and can be defined by the user at run time. Although it would be better to generate the `GROUPS` clause with a `weight` argument based on different workloads of parallel regions, this is not considered at the moment.

As an example, the following nested loop:

```
DO K = 1, NK
   RHO = 1/NORMK(K)
   DO J = 2, NJ
      A(J,K) = A(J,K) + RHO * B(J,K)
   END DO
END DO
```

will be transformed by CAPO into:

```
!$OMP PARALLEL GROUPS(ngroups)
!$OMP & PRIVATE (RHO, K)
!$OMP DO
DO K = 1, NK
   RHO = 1/NORMK(K)
!$OMP PARALLEL DO PRIVATE (J)
   DO J = 2, NJ
      A(J,K) = A(J,K) + RHO * B(J,K)
   END DO
!$OMP END PARALLEL DO
END DO
```

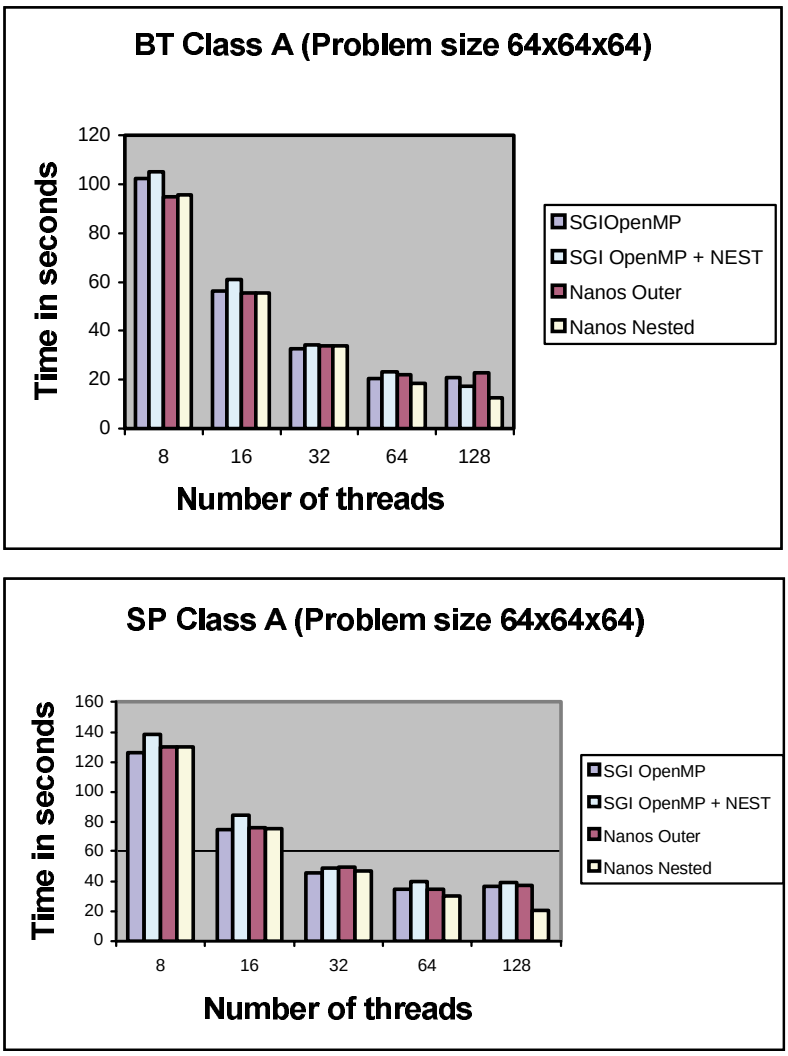

Fig. 2. Timing results for class A benchmarks.

```
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Note that for this loop the SGI `NEST` clause is not applicable, since there is a statement between DO K and DO J.

4. Case studies

In this section we show examples for successful and not so successful automatic multilevel parallelization. We have parallelized the three application benchmarks (BT, SP, and LU) from the NAS Parallel Benchmarks [4] and the ARC3D [22] application code using the CAPO multilevel parallelization feature and examined its effectiveness.

In each of our experiments we generate nested OpenMP directives and use the NanosCompiler for compilation. As discussed in Sections 2 and 3, the nested parallel code contains the `GROUPS` clause at the outer level. According to the OpenMP standard, the number of executing threads can be specified at runtime by the environment variable `OMP_NUM_THREADS`. We introduce the environment variable `NANOS_GROUPS` and modify the source code to have the main routine check the value of this variable and set the argument to the `GROUPS` clause accordingly. This allows us to run the same executable not only with different numbers of threads, but also with different numbers of groups. We compare the timings for different numbers of groups to each other. Note that single level parallelization of the outer loop corresponds to the case that the number of

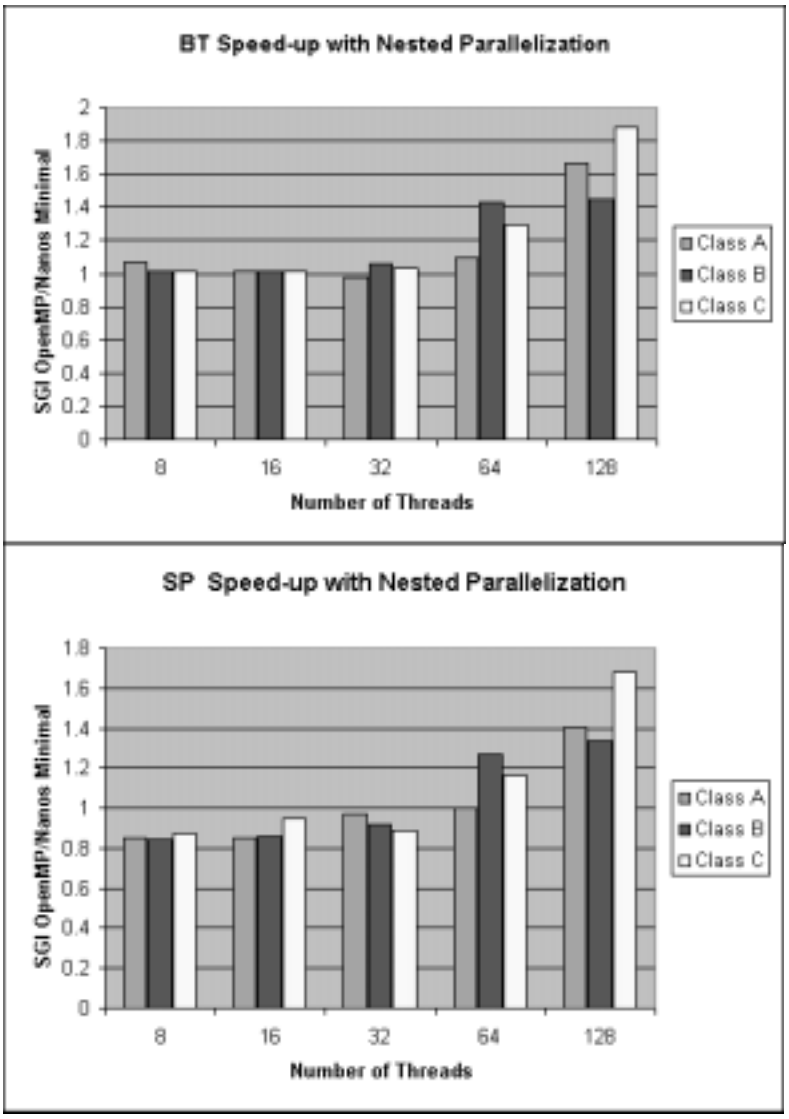

Fig. 3. Speed-up due to nested parallelism.

executing threads is equal to the number of groups, i.e. there is only one thread in each group. We compare these timings to those resulting from compilation with the native SGI compiler, which supports only the single level OpenMP parallelization and serializes inner parallel loops. We will also give timings for the SGI compiler using the NEST clause which we described in 3.1. These timings show how true multilevel parallelism compares to single parallelization employing a 2-dimensional work distribution.

The timings were obtained on a SGI Origin 2000 with R12000 CPUs, 400MHz clock, and 768MB local memory per node.

4.1. Successful multilevel parallelization: The BT and SP benchmarks

The NAS Parallel Benchmarks BT and SP are both simulated CFD applications with a similar structure. They use an implicit algorithm to solve the 3D compressible Navier-Stokes equations. The x, y, and z dimensions are decoupled by usage of an Alternating Direction Implicit (ADI) factorization method. In BT, the resulting systems are block-tridiagonal with 5×5 blocks. The systems are solved sequentially along each dimension. SP uses a diagonalization method that decouples each block-tridiagonal system into three inde-

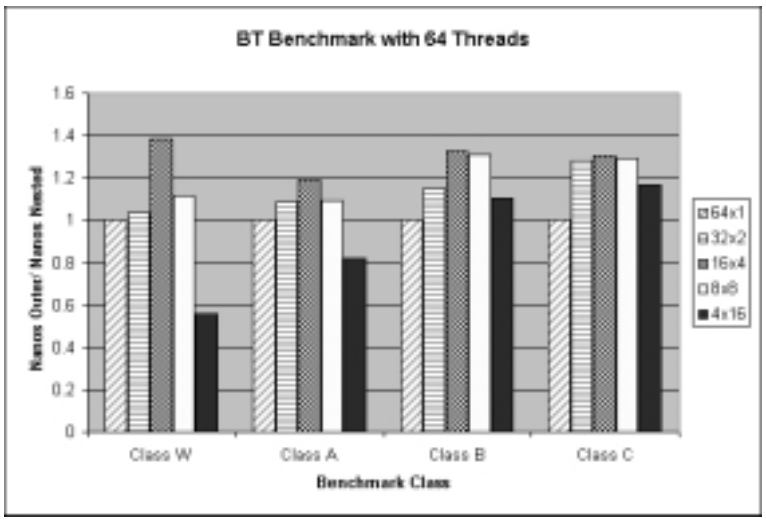

Fig. 4. Timings of BT with varying number of groups and threads per group.

pendent scalar pentadiagonal systems that are solved sequentially along each dimension.

A study about the effects of single level OpenMP parallelization of the NAS Parallel Benchmarks can be found in [12]. In our experiments we started out with the same serial implementation of the codes that was the basis for the single level OpenMP implementation as described in [12]. We ran class A ($64 \times 64 \times 64$ grid points), B ($102 \times 102 \times 102$ grid points), and C ($162 \times 162 \times 162$ grid points) for the BT and SP benchmarks. As an example we show timings for problem class A for both benchmarks in Fig. 2. We denote by:

- *SGI OpenMP:* the time for outer loop parallelization using just the native SGI compiler,
- *SGI OpenMP+NEST:* The time for outer loop parallelization using the SGI NEST clause if applicable.
- *Nanos Outer:* the time for outer loop parallelization using the NanosCompiler,
- *Nanos Nested:* the minimal time for nested parallelization using the NanosCompiler.

The programs compiled with the SGI OpenMP compiler scale reasonably well up to 64 threads, but do not show any further speed-up if more threads are being used. For a small number of threads (up to 64), the outer level parallel code generated by the NanosCompiler performs about the same as the code generated by the SGI compiler. When increasing the number of threads from 64 to 128, the multilevel parallel code still shows a speed-up, provided the number of groups is chosen in an optimal way. We observed a speed-up of up to 85% for 128 threads. In Figure 3 we show the speed-up resulting from nested parallelization for three problem classes of the SP and BT benchmarks.

Table 1
Thread workload for the class A problems BT and SP

# Groups	Max # Iters	Min # Iters
64	62	0
32	62	31
16	64	45
8	64	49
4	64	45

The timings show that the SGI NEST clause is of limited benefit. It improves the performance of the BT benchmark slightly, but it does not help the SP benchmark. The time consuming routines in the two benchmarks are the three solvers in x, y, and z-direction and the computation of the right hand side. In case of BT, CAPO parallelized 28 loops, 11 of which were suitable for the NEST clause. This includes the major loops in the three solver routines. The time consuming loops in the calculation of the right hand side are not suitable for the NEST clause, since they contain statements between the DO statements. The situation is a lot worse for the SP benchmark. CAPO parallelized 31 loops. The NEST clause could be generated for 11 of them. The three main loops in the solver routines were not suitable for the NEST clause, because the inner loops are enclosed in subroutine calls. The computation of the right hand side contains nested loops that are not tightly nested, just like in the case of BT. The NEST clause could only be applied to loops with a very low workload. In this case, distributing the work in multiple

dimensions leads to a slight decrease of performance for a small number of threads. Neither the occurrence of code between the DO statements nor inner loops enclosed within subroutine calls poses an obstacle to nested parallel regions supported by the NanosCompiler. For the BT benchmark CAPO parallelized 13 of the 28 parallel loops employing nested parallel regions and the GROUPS clause. For the SP benchmark CAPO identified 17 of the 31 parallel 31 loops, as suitable for nested parallelism. In both benchmarks the most time consuming loops are parallelized in two dimensions. All of the nested parallel loops are at least triple nested. The structure of the loops is such that the two outer most loops can be parallelized. The inner parallel loops enclose one or more inner loops and contain a reasonably large amount of computational work.

The reason that multilevel parallelism has a positive effect on the performance of these loops is mainly due to the fact that load balancing between the threads is improved. For class A, for example, the number of iterations is typically 62. If only the outer loop is parallelized, using more than 62 threads will not improve the performance any further. In the case of 64 threads, 2 of them will be idling. If, however, the second loop level is also parallelized, all 64 threads can be put to use. Our experiments show that by choosing the number of groups too small, the performance will actually decrease. Setting the number of groups to 1 effectively moves the parallelism completely to the inner loop, which will in most cases be less efficient than parallelizing the outer loop.

In Table 1 we show the maximal and minimal number of iterations (for class A) of the inner parallel loop that a thread has to execute, depending on the number of groups.

To give a flavor of how the performance of the multilevel parallel code depends on the grouping of threads we show timings for the BT benchmark on 64 threads and varying number of groups in Fig. 4. In the figure we indicate by NxM the situation where N groups are being used with M threads each. The timings indicate that good criteria to choose the number of groups are:

- Efficient granularity of the parallelism, i.e., the number of groups has to be sufficiently large thereby avoiding excessive parallelization overhead that occurs when parallelism is moved to inner loop level. In our experiments we observe that the number of groups should not be smaller than the number of threads within a group.
- The number of groups has to be small enough to allow a good balancing of work among the threads.

An enhancement to CAPO in support of multilevel OpenMP parallelization would be to automatically choose an appropriate number of groups based on the workload for each thread.

4.2. The need for OpenMP extensions: The LU benchmark

The LU application benchmark is a simulated CFD application that uses the symmetric successive overrelaxation (SSOR) method to solve a seven band block-diagonal system resulting from finite-difference discretization of the 3D compressible Navier-Stokes equations by splitting it into block lower and block upper triangular systems.

As starting point for our tests we choose the pipelined implementation of the parallel SSOR algorithm, as described in [12]. The example below shows the loop structure of the lower-triangular solver in SSOR. The lower-triangular and diagonal systems are formed in routine `JACLD` and solved in routine BLTS. The index K corresponds to the third coordinate direction.

```
...
DO K = KST, KEND
   CALL JACLD (K)
   CALL BLTS (K)
END DO
...
SUBROUTINE BLTS
...
DO J = JST, JEND
   Loop\_Body (J, K)
END DO
...
RETURN
END
```

All of the loops involved carry data dependences that prevent straightforward parallelization. The structure of the loop body is such that iteration (J, K) depends on iterations (J-1, K) and (J, K-1). There is, however, the possibility to exploit a certain level of parallelism by using software pipelining as described in Section 3.3. To set up a pipeline for the outer loop, thread 0 starts to work on its first chunk of data in K direction. Once thread 0 finishes, thread 1 can start working on its chunk for the same K and, in the meantime, thread 0 moves on to the K+1. The directives generated by CAPO to implement the pipeline for the outer loop are shown in Fig. 5.

```
!$OMP PARALLEL PRIVATE(K,iam,numt)
   iam = omp_get_thread_num()
   numt = omp_get_num_threads()
   isync(iam) = 0
!$OMP BARRIER
   DO K = KST, KEND
      CALL JACLD (K)
      CALL BLTS (K)
   END DO
!$OMP END PARALLEL
   SUBROUTINE BLTS (K)
   ...
   if (iam .gt. 0 .and.
        iam .lt. numt) then
     do while(isync(iam-1) .eq. 0)
!$OMP FLUSH(isync)
     end do
     isync(iam-1) = 0
!$OMP FLUSH(isync)
   end if
!$OMP DO
   DO J = JST, JEND
     Loop_Body (J,K)
   END DO
!$OMP END DO nowait
   if (iam .lt. numt) then
     do while (isync(iam) .eq. 1)
!$OMP FLUSH(isync)
     end do
     isync (iam) = 1
!$OMP FLUSH(isync)
   endif
   RETURN
   END
```

Fig. 5. The one-dimensional parallel pipeline implemented in LU.

The K loop is placed inside a parallel region. Two OpenMP library functions are called to obtain the current thread identifier (`iam`) and the total number of threads (`numt`). The shared array `isync` is used to indicate the availability of data from neighboring threads. Together with the `FLUSH` directive in a `WHILE` loop it is used to set up the point-to-point synchronization between threads. The first WHILE ensures that thread iam will not start with its slice of the J loop before the previous thread has updated its data. The second `WHILE` is used to signal data availability to the next thread.

The NanosCompiler team is currently defining and implementing OpenMP extensions to easily express the precedence relations that originate pipelined computations. These extensions are also valid in the scope of nested parallelism. They are based on two components:

– The ability to name work-sharing constructs (and therefore reference any piece of work coming out of it).

```
!$OMP PARALLEL PRIVATE(K,iam,numt)

   DO K = KST, KEND
      CALL JACLD (K)
      CALL BLTS (K)
   END DO
!$OMP END PARALLEL
   SUBROUTINE BLTS (K)
   ...
!$OMP DO NAME (inner_loop)
   DO J = JST, JEND
!$OMP PRED (inner_loop, j-1)
     Loop_Body (J,K)
!$OMP SUCC (inner_loop, j+1)
   END DO
!$OMP END DO nowait
   …
   RETURN
   END
```

Fig. 6. One-dimensional pipeline using directives.

– The ability to specify predecessor and successor relationships between named work-sharing constructs (PRED and SUCC clauses).

This avoids the manual transformation of the loop to access data slices and manual insertion of synchronization calls. From the new directives and clauses, the compiler automatically builds synchronization data structures and insert synchronization actions following the predecessor and successor relationships defined [8]. Figure 6 shows the pipelined loop from Fig. 5 when using the new directives.

In Fig. 7 we show the timings for LU benchmark comparing the one-level pipelined implementation using the synchronization mechanism from Fig. 5, the one-level pipelined implementation using the new NanosCompiler directives, and a 2-dimensional pipelined implementation based on MPI. The compiler directives based implementation shows about the same performance as the hand-coded synchronization.

The timings in Fig. 7 show that the directive based implementation does not scale as well as a message passing implementation of the same algorithm. The cost of pipelining results mainly from waiting (at start-up and termination). The message-passing version employs a 2 dimensional pipeline where the wait cost can be greatly reduced. The use of nested OpenMP directives offers the potential to achieve similar scalability to the message passing implementation.

There is, however, a problem in setting up a directive-based two-dimensional pipeline. The new directives allow synchronization of threads within one team and synchronization between different teams.

The structure of the Loop_Body depicted in Fig. 5 looks like:

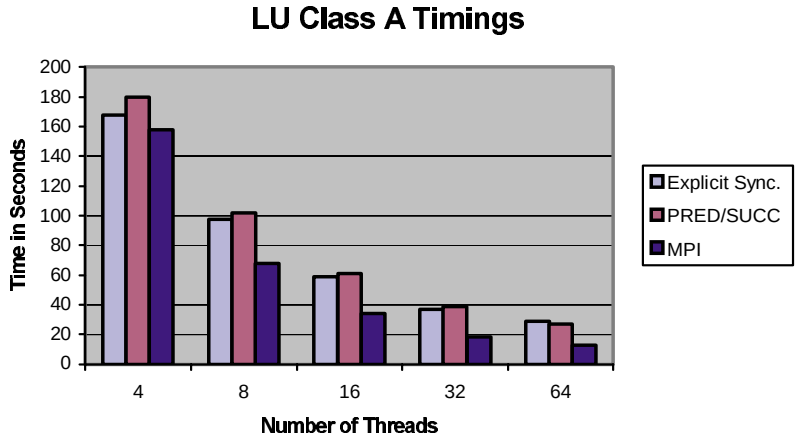

Fig. 7. Timings for different implementations of LU.

```
DO I = ILOW, IHIGH
   DO M = 1, 5
      TV(M, I, J) = V(M, I, J, K-1)
                    + V(M,I,J-1,K)
                    + V(M,I-1,J,K)
   END DO
   ...
   DO M = 1, 5
      V(M,I,J,K)= TV(M,I,J)
   END DO
END DO
```

If both J- and I-loop are to be parallelized employing pipelines, a thread would need to be able to synchronize with its neighbor in the J- and I-directions on different nesting levels. Parallelizing the I-loop with OpenMP directives introduces an inner parallel region, as shown below (see also the discussion in Section 3.3):

```
!$OMP PARALLEL
   Synchronization1
!$OMP DO
   DO JT = ...
!$OMP PARALLEL
      DO J = JLOW, JHIGH
         Synchronization2
!$OMP DO
         DO I = ILOW, IHIGH
            ...
         END DO
!$OMP END DO NOWAIT
         Synchronization2
      END DO
!$OMP END PARALLEL
   END DO
!$OMP END DO NOWAIT
   Synchronization1
```

The end of the inner parallel region forces the threads to join and destroys the multilevel pipeline mechanism. In order to set up a 2-dimensional pipeline, two possibilities should be taken into account. The first one is removing the implicit barrier at the end of the inner parallel region. Such a NOWAIT clause would violate the OpenMP standard. The second alternative is the use of nested `OMP DO` directives within the same parallel region. This is a proposed extension to the OpenMP standard, but is not part of OpenMP at this time.

The SGI compiler provides the NEST clause, which simply uses one level of parallelism but performs a two-dimensional distribution of work. As discussed in 3.1, the loops need to be tightly nested for the NEST clause to be applicable. The loop structure of loops allowing pipelined execution in the LU benchmark is suitable for the SGI NEST clause. However, the SGI compiler does not provide extensions for explicit thread synchronization which is necessary for pipelined execution of the loop.

As we have seen in Section 4.1, the restrictions to application of the NEST clause greatly limit its usage for many time consuming loops. It would be desirable to have these restrictions removed. Allowing nested `OMP DO` directives within the same parallel region would remove these restrictions. Code between the DO statements could be handled by having only part of the threads executing these statements. In case that the inner loop is enclosed in a subroutine call, more complicated techniques, involving procedure in-lining are necessary.

4.3. Unsuitable loop structure in ARC3D

ARC3D uses an implicit scheme to solve Euler and Navier-Stokes equations in a three-dimensional (3D)

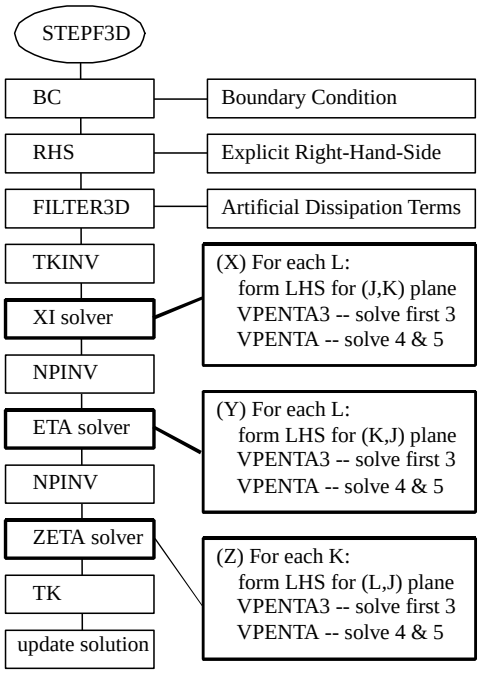

Fig. 8. The schematic flowchart of the ADI solver in ARC3D.

rectilinear grid. The main component is an ADI solver, which results from the approximate factorization of finite difference equations. The actual implementation of the ADI solver (subroutine STEPF3D) in the serial ARC3D is illustrated in Fig. 8. It is very similar to the SP benchmark.

For each time step, the solver first sets up boundary conditions (`BC`), forms the explicit right-hand-side (`RHS`) with artificial dissipation terms (`FILTER3D`), and then sweeps through three directions (X, Y and Z) to update the 5-element fields, separately. Each sweep consists of forming and solving a series of scalar pentadiagonal systems in a two-dimensional plane one at a time. Two-dimensional arrays are created from the 3D fields and are passed into the pentadiagonal solvers (`VPENTA3` for the first 3 elements and `VPENTA` for the 4 and 5th elements, both originally written for vector machines), which perform Gaussian eliminations. The solutions are then copied back to the three-dimensional residual fields. Between sweeps there are routines (`TKINV`, `NPINV` and `TK`) to calculate and solve small, local 5×5 eigensystems. Finally the solution is updated for the current time step.

We ran ARC3D for two different problem sizes. In both cases the performance dropped by 10% to 70% when the number of groups was smaller than the number of threads, i.e. when multilevel parallelism was used. Example timings for both problem sizes and 64 threads are given in Fig. 9. Figure 10 shows the timings for outer level parallelism.

Even though the time consuming solver in ARC3D is similar to the one in the SP benchmark, our approach to automatic multilevel parallelization was not successful. For ARC3D CAPO identified 58 parallel loops, 35 of which were suitable for nested parallelization. 19 of the 35 nested parallel loops had very little work in the inner parallel loop and inefficient memory access. An example is shown below:

```
!$OMP PARALLEL DO GROUPS(ngroups)
!$OMP &  PRIVATE(AR,BR,CR,DR,ER)
DO K = KLOW, KUP
   ...
!$OMP PARALLEL DO
   DO L = 2, LM
      DO J = 2, JM
         AR(L,J) = AR(L,J) + V(J,K,L)
         BR(L,J) = BR(L,J) + V(J,K,L)
         CR(L,J) = CR(L,J) + V(J,K,L)
         DR(L,J) = DR(L,J) + V(J,K,L)
         ER(L,J) = ER(L,J) + V(J,K,L)
         CR(L,J) = CR(L,J) + 1.
      END DO
   END DO
END DO
```

Parallelizing the L loop increases the execution time of the loop considerably due to a high number of cache invalidations. The occurrence of many such loops in the original ARC3D code nullifies the benefits of a better load balance and we see no speed-up for multilevel parallelism.

The NEST clause could be applied to the same 35 loops that were suitable for nested parallelization. However, the NEST clause did not improve the performance of the code.

The example of ARC3D shows that parallelizing all loops in an application indiscriminately on two levels with the same number of groups and the same weight for each group may actually increase the execution time.

CAPO provides a browser for first level directives. The browser allows the user to examine the directives that have been automatically placed in the code. The user has the possibility to provide knowledge about input data or code structure so that the placement of directives can be optimized. At the moment this feature is not available for second level directives. The example of ARC3D shows that we will need to extend the

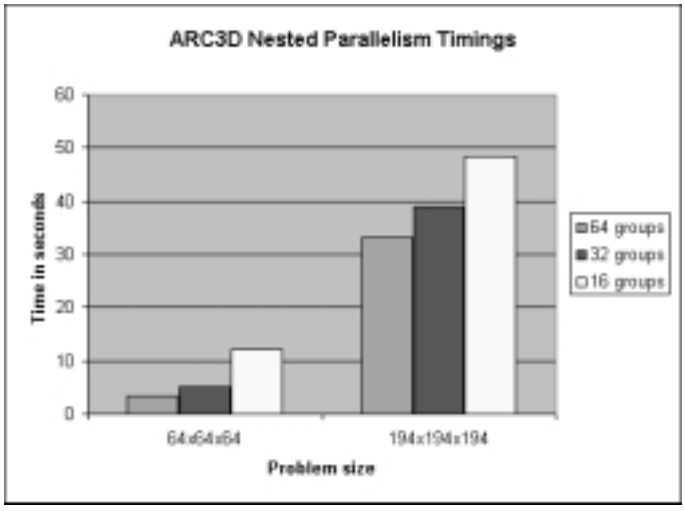

Fig. 9. Timings of ARC3D with varying number of thread groups for a given total of 64 threads.

CAPO directives browser so that the user can inspect all multilevel parallel loops. This will allow optimizing the placement of second level directives.

5. Related work

There are a number of commercial and research parallelizing compilers and tools that have been developed over the years. Some of the more notable ones include Superb [24], Polaris [6], Suif [24], KAIfls toolkit [15], VAST/Parallel [21], and FORGexplorer [1].

Regarding OpenMP directives, 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). 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 [23]). 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 [7] 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.

Compaq recently announced the support of nested parallel region by its Fortran compiler for Tru64 systems [3]. The Omni compiler [19], which is part of the Real World Computing Project, also supports nested parallelism through OpenMP directives.

There are a number of papers reporting experiences in combining multiple programming paradigms (such as MPI and OpenMP) to exploit multiple levels of parallelism. 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 [5]. The authors show the improvement due to nested parallelization.

Other experiences using nested OpenMP directives with the NanosCompiler are reported in [2]. In the examples discussed there, the directives have not been automatically generated.

6. Project status and future plans

We have extended the CAPO automatic parallelization support tool to automatically generate nested OpenMP directives. We used the NanosCompiler to evaluate the efficiency of our approach. We conducted several case studies which, showed that:

- Nested parallelization was useful to improve load balancing.
- Nested parallelization can be counter productive when applied without considering workload distribution and memory access within the loops.

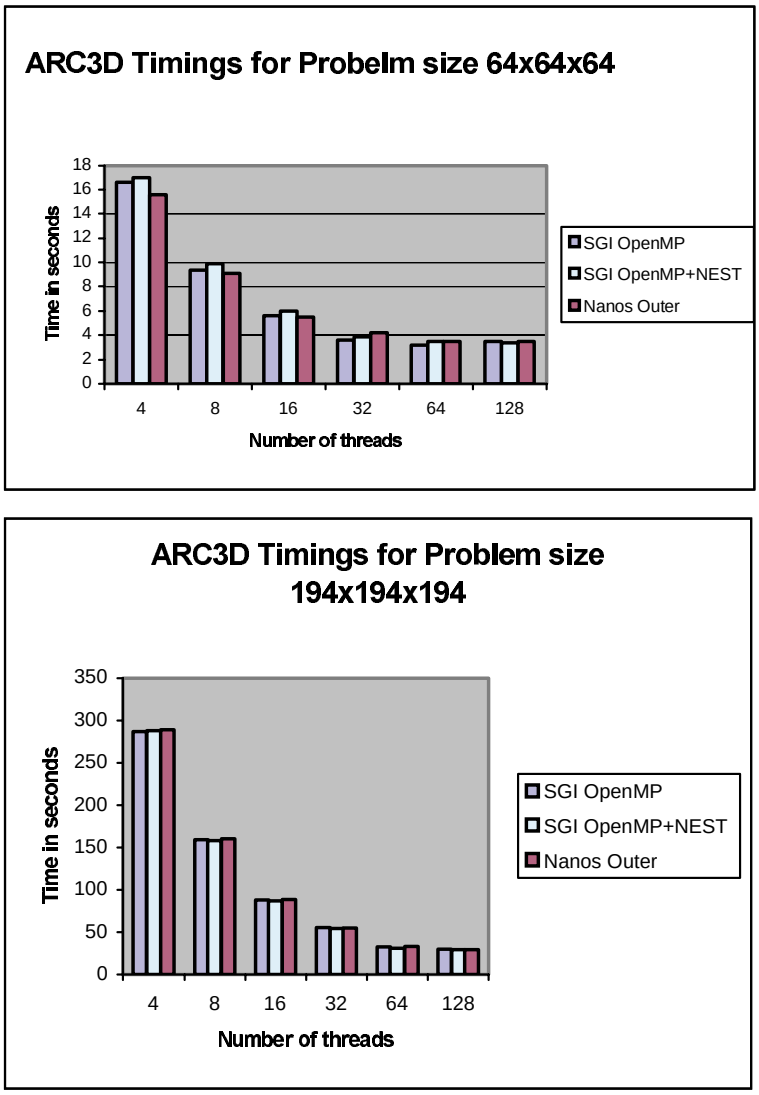

Fig. 10. Timings from the outer level parallelization of ARC3D.

– Extensions to the OpenMP standard are needed to implement nested parallel pipelines.

We are planning to enhance the CAPO directives browser to allow the user to view loops, which are candidates for nested parallelization. Nested parallelization may then be turned on selectively and necessary loop transformations can be performed. We are also considering the automatic determination of an appropriate number of groups and the assignment of different weights to the groups. Currently CAPO is also being extended to support hybrid parallelism which combines coarse-grained parallelization based on message passing and fine-grained parallelization based on directives.

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.

Acknowledgments

The authors would like to thank Rob Van der Wijngaart and Michael Frumkin of NAS and Jesús Labarta from CEPBA for reviewing the paper and the suggestions they made for improving it. The authors also wish to thank the CAPTools team (C. Ierotheou, S. Johnson, P. Leggett, and others) at the University of Greenwich

for their support on CAPTools. This work was supported by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation, by the CEPBA and by the Spanish Ministry of Science and Technology and the European Union FEDER contract TIC2001-0995-C02-01.

References

[1] Applied Parallel Research Inc., FORGE Explorer, http://www.apri.com/.

[2] 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.

[3] Compaq Fortran Release Notes for Compaq Tru64 UNIX Systems April 2001, http://www5.Compaq.com/fortran/docs/unix-um/relno.htm.

[4] 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/.

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

[6] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Lee, T. Lawrence, J. Hoeflinger, D. Padua, Y. Paek, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu and S. Weatherford, Restructuring Programs for High-Speed Computers with Polaris, 1996 ICPP Workshop on Challenges for Parallel Processing, August 1996, pp. 149–162.

[7] 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** (February 1998).

[8] 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.

[9] 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, October 2000, pp. 1205–1218.

[10] 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, August 2000, pp. 317–331.

[11] C.S. Ierotheou, S.P. Johnson, M. Cross and P. Leggett, Computer Aided Parallelisation Tools (CAPTools) – Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes, *Parallel Computing* **22** (1996), 163–195, http://captools.gre.ac.uk/.

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

[13] H. Jin, M. Frumkin and J. Yan, Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes, in Proceedings of Third International Symposium on High Performance Computing (ISHPC2000), Tokyo, Japan, October 16–18, 2000.

[14] S.P. Johnson, M. Cross and M. Everett, Exploitation of Symbolic Information In Interprocedural Dependence Analysis, *Parallel Computing* **22** (1996), 197–226.

[15] Kuck and Associates, Inc., Parallel Performance of Standard Codes on the Compaq Professional Workstation 8000: Experiences with Visual KAP and the KAP/Pro Toolset under Windows NT, Champaign, IL, Assure/Guide Reference Manual, 1997.

[16] Message Passing Interface, http://www-unix.mcs.anl.gov/.

[17] 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, June 1999.

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

[19] Omni: RCWP OpenMP Compiler Project, http://www.hpcc.jp/omni.

[20] OpenMP Fortran/C Application Program Interface, http://www.openmp.org/.

[21] Pacific-Sierra Research, VAST/Parallel Automatic Parallelizer, http://www.psrv.com/.

[22] T.H. Pulliam, Solution Methods In Computational Fluid Dynamics, Notes for the von Kármán Institute For Fluid Dynamics Lecture Series, Rhode-St-Genese, Belgium, 1986.

[23] 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.

[24] R.P. Wilson, R.S. French, C.S. Wilson, S.P. Amarasinghe, J.M. Anderson, S.W.K. Tjiang, S. Liao, C. Tseng, M.W. Hall, M. Lam and J. Hennessy, SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers, Computer Systems Laboratory, Stanford University, Stanford, CA.

[25] H.P. Zima, H.-J. Bast and H.M. Gerndt, SUPERB- A Tool for Semi-Automatic MIMD/SIMD Parallelisation, *Parallel Computing* **6** (1988).