Automatic Multilevel Parallelization Using OpenMP

Haoqiang Jin, Gabriele Jost*, Jerry Yan
NAS Division, NASA Ames Research Center, Moffett Field, CA 94035-1000 USA
{hjin,gjost,yan}@nas.nasa.gov

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,marc,xavier}@ac.upc.es

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 first 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 [19].

OpenMP was introduced as an industrial standard for shared-memory programming with directives. Recently, it has gained significant popularity and wide compiler support. However, relevant performance issues must still be addressed which concern programming model design as well as implementation. In addition to that, extensions to the standard are being proposed and evaluated in order to widen the applicability of OpenMP to a broad class of parallel applications without sacrificing portability and simplicity.

What has not been clearly addressed in OpenMP is the exploitation of multiple levels parallelism. 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 address scalability of applications 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 [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 threads 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 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 way to multilevel parallelism.

In this study, we have extended the automatic parallelization tool, CAPO, to allow for the generation of nested OpenMP parallel constructs in order to support multilevel shared memory parallelization. CAPO automates the insertion of OpenMP 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.

To this point there is little reported experience with shared memory multilevel parallelism. By being able to generate nested directives automatically in a reasonable amount of time we hope to be able to gain a better understanding of performance issues and the needs of application programs when in comes to exploiting multilevel parallelism.

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.

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

*The author is an employee of Computer Sciences Corporation

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 is 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 vector weight 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 as 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 from CAPTools [11] and also benefits from a directive browser to allow the user to examine and refine 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

The single loop level parallelism automatically exploited in CAPO can be defined by the following three stages (see [13] for more details of these stages and their implementation):

1) Identification of parallel loops and parallel regions – this includes a comprehensive breakdown of the different loop types, such as serial, parallel including reductions, and pipelines. The outermost parallel loops are considered for parallelization so long as they provide sufficient granularity. Since the dependence analysis is interprocedural, the parallel regions can be defined as high up in the call tree as possible. This provides an efficient placement of the directives.

2) Optimization of parallel regions and parallel loops – the fork-and-join overhead (associated with starting a parallel region) and the synchronizing cost are greatly lowered by reducing the number of parallel regions required. This is achieved by merging together parallel regions where there is no violation of data usage. In addition, the synchronization between successive parallel loops is removed if it can be proved that the loops can correctly execute asynchronously (using the NOWAIT clause).

3) Code transformation and insertion of OpenMP directives – this includes the search for and insertion of possible THREADPRIVATE common blocks. There is also special treatment for private variables in non-threadprivate common blocks. If there is a

usage conflict then the routine is cloned and the common block variable is added to the argument list of the cloned routine. Finally, the call graph is traversed to place OpenMP directives within the code. This includes the identification of necessary variable types, such as `SHARED`, `PRIVATE`, and `REDUCTION`.

3.2 Extension to multilevel parallelization

Our extension to OpenMP multilevel parallelism is based on parallelism at different loop nests. Multilevel parallelism can also be exploited with task parallelism but this is not considered, partly because task parallelism is not well defined in the current OpenMP specification. 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 Figure 1. Besides the data dependence analysis in the beginning he 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.

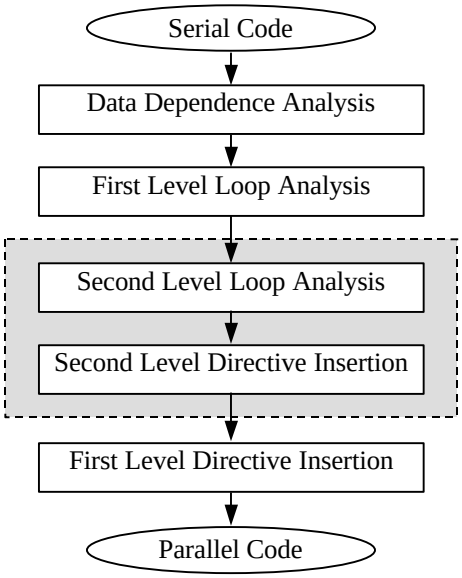

Figure 1: Steps in multilevel parallelization

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. The step performed before inserting any directives in the first-level is to ensure a consistent picture is maintained for any variables and codes that may be changed or introduced during the code transformation.

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 Figure 1.

3.3 Implementation consideration

In order to maintain consistency during the code transformations that occur during the parallelization process we need to update data dependencies 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 reduction in OpenMP v1.x. The data dependence graph needs to be updated to reflect the change due to this transformation, such as associating dependence edges related to the original variable to the local variable and adding new dependences for the local variable from the local updates to the global update. Performing a full data dependence analysis for the modified code block is another possibility but this would not take advantage of the information already obtained from the earlier dependence analysis.

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 and treat any private variables defined in common blocks by a special transformation as mentioned in Section 3.1.

The scope of the synchronization directives should be carefully followed. For example, the `MASTER` directive is not allowed in the extent of a `PARALLEL DO`. This changes the way a software pipeline (see [13] for further explanation) can be implemented if it is nested inside an outer parallel loop.

When implementing a pipeline, the outer loop needs to be considered as well. This is illustrated by the following example. Assume we have a nest containing two loops:

```
DO K=1,NK
  DO J=2,NJ
    A(J,K) = A(J,K) + A(J-1,K)
```

The outer loop K is parallel and the inner loop J can be set up with a pipeline. After inserting directives at the second level to set up the pipeline, we have

```
!$OMP PARALLEL
  DO K=1,NK
!..point-to-point sync directive
!$OMP DO
    DO J=2,NJ
      A(J,K) = A(J,K) + A(J-1,K)
```

The implementation of the point-to-point synchronization with directives is illustrated in Section 4.2. In order to parallelize the K loop at the outer level, we need to first transform the loop into a form such that the outer-level directives can be added. It is achieved by explicitly calculating the K-loop bound for each outer-level thread as shown in the following codes:

```
!$OMP PARALLEL DO GROUPS(ngroups)
   DO IT=1,omp_get_num_threads()
     CALL calc_bound(IT,1,NK,
  >                  low,high)
!$OMP PARALLEL
     DO K=low,high
!..point-to-point sync directive
!$OMP DO
       DO J=2,NJ
         A(J,K) = A(J,K) + A(J-1,K)
```

The function "calc_bound" calculates the K loop bound (low, high) for a given IT (the thread number) from the original K loop limit. Only then are the first-level directives added to the IT loop (instead of the K loop). The method is not as elegant as one would prefer, but it points to some of the limitations with the nested OpenMP directives. In particular we would not be able to set up a two-dimensional pipeline, since it 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.

4 Case Studies

In this section we show examples for successful and not so successful automatic mulitlevel parallelization. We have parallelized the three application benchmarks (BT, SP, and LU) from the NAS Parallel Benchmarks [4] and the ARC3D [21] 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 and building of the executables. 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 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.

The timings were obtained on an 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 5x5 blocks. The systems are solved sequentially along each dimension. SP uses a diagonalization method that decouples each block-tridiagonal system into three independent 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 (64x64x64 grid points), B (102x102x102 grid points), and C (162x162x 162 grid points) for the BT

and SP benchmarks. As an example we show timings for problem class A for both benchmarks in Figure 2.

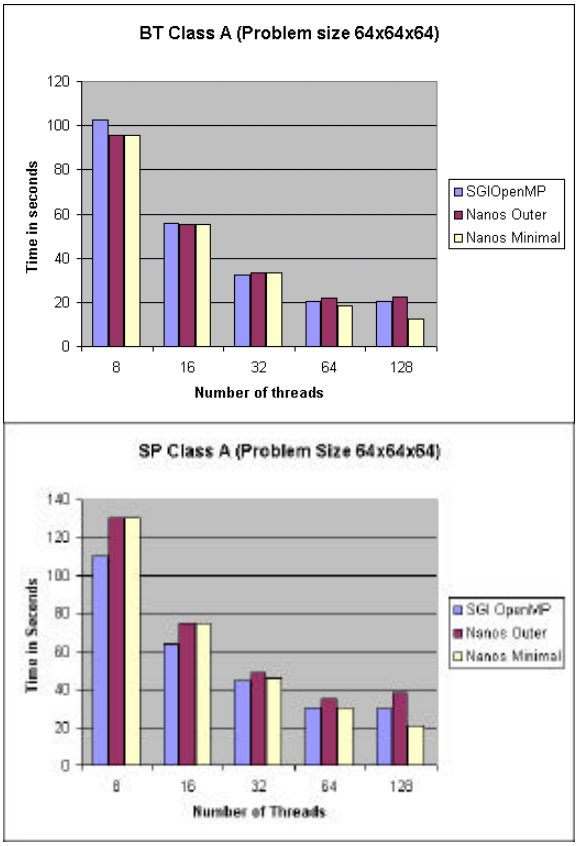

Figure 2: Timing results for class A benchmarks.

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 Nanos Compiler runs somewhat slower than the code generated by the SGI compiler, but its relative performance improves with increasing number of threads. When increasing from 64 to 128 threads, 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. We denote by

- SGI OpenMP: the time for outer loop parallelization using just the native SGI compiler,
- Nanos Outer: the time for outer loop parallelization using the NanosCompiler,
- Nanos Minimal: the minimal time for nested parallelization using the NanosCompiler.

For the BT benchmark CAPO parallelized 28 loops, 13 of which were suitable for nested parallelization. For the SP benchmark CAPO parallelized 31 loops, 17 of which were suitable for multilevel 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, he number of iterations is 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.

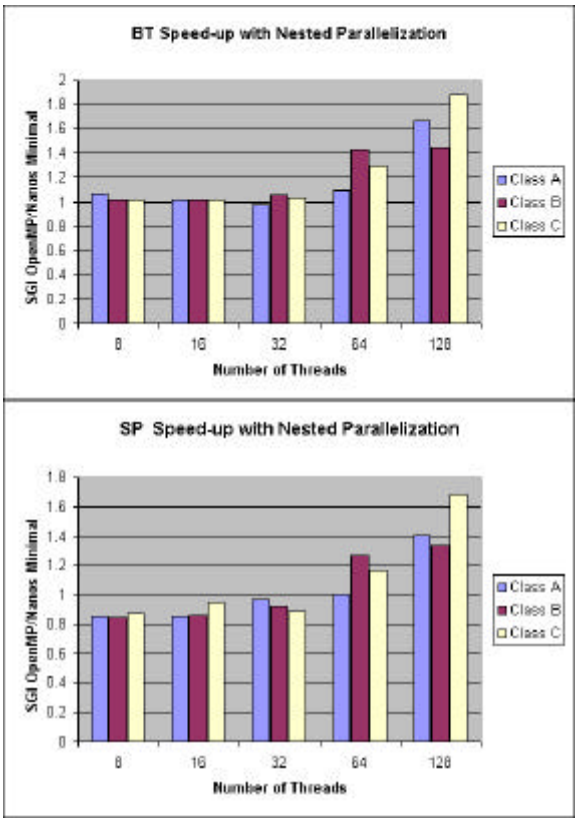

Figure 3: Speed-up due to nested parallelism.

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.

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

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

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 Figure 4. 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 small. 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 large enough to ensure a good balancing of work among the threads.

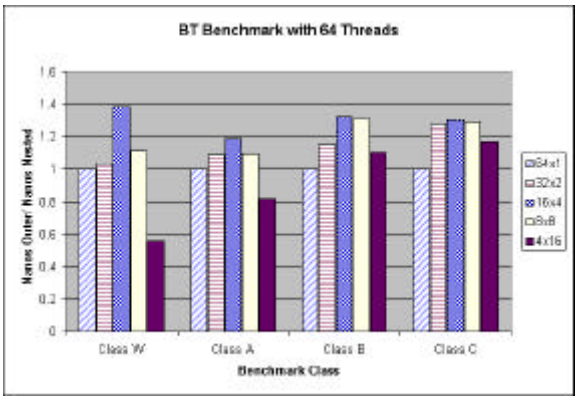

Figure 4: Timings of BT with varying number of thread groups.

4.2 The need for OpenMP extensions: the LU benchmark

The LU application benchmark is a simulated CFD application that uses the symmetric successive over-relaxation (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
```

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 next K. CAPO detects such opportunities for pipelined parallelism. The directives generated by CAPO to implement the pipeline for the outer loop are shown in Figure 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
```

Figure 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 performance of the pipelined parallel implementation of the LU benchmark is discussed in [12]. The timings 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 wait during startup and finishing. 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 structure of the Loop_Body depicted in Figure 5 looks like:

```
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 we would need to have the possibility of nested **`OMP DO`** directives within the same parallel region. The NanosCompiler team is currently implementing OpenMP extensions to address this problem. A brief overview on this work is given in Section 6.

4.3 Unsuitable loop structure in ARC3D

ARC3D uses an implicit scheme to solve Euler and Navier-Stokes equations in a three-dimensional (3D) 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 Figure 6. It is very similar to the SP benchmark.

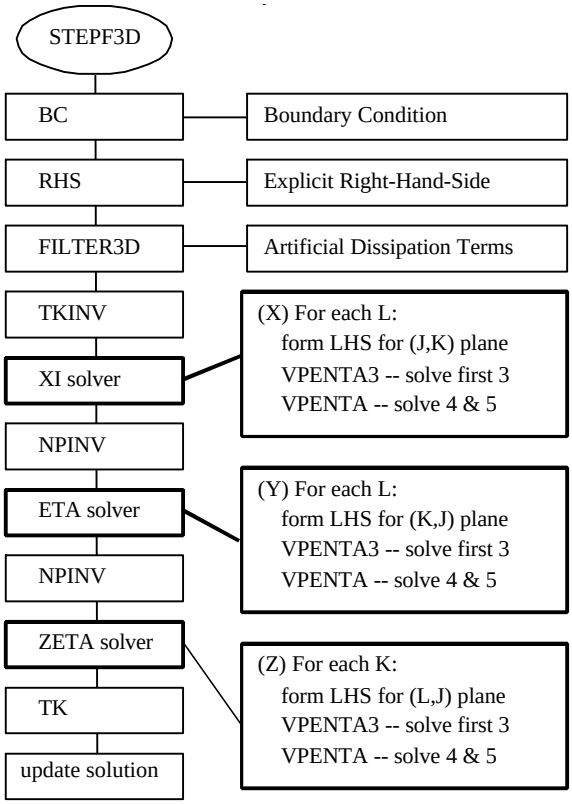

Figure 6: The schematic flowchart of the ADI solver in ARC3D.

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 5x5 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 Figure 7. The timings for outer level parallelism are given in Figure 8.

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.

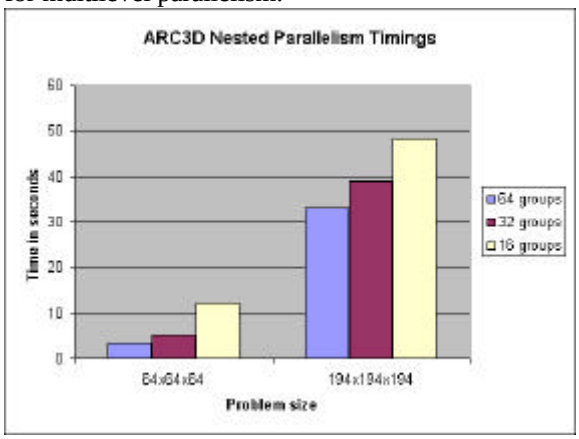

Figure 7: Timings of ARC3D with varying number of thread groups for a given total of 64 threads.

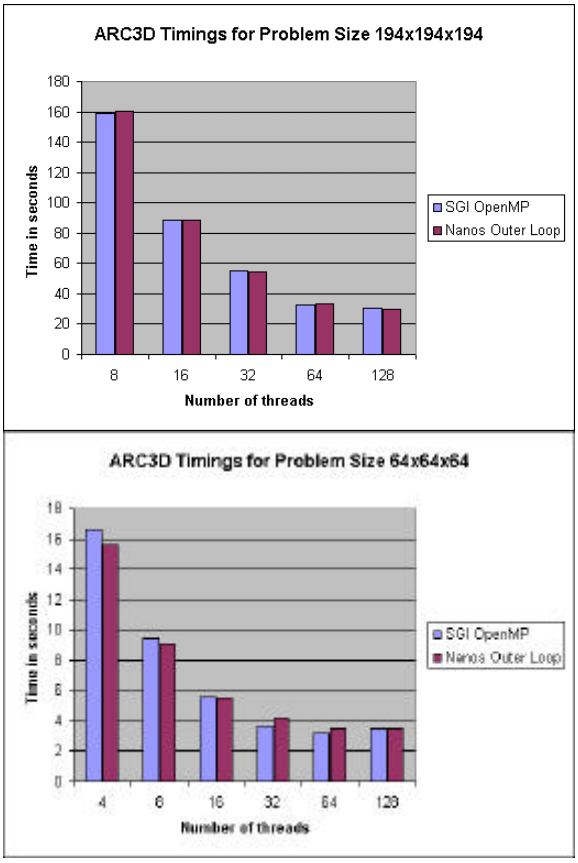

Figure 8: Timings from the outer level parallelization of ARC3D.

The example of ARC3D shows that parallelizing all loops in an application indiscriminately on two levels with the same name number of groups and the same weight for each group may actually increase the execution time. At the least we will need to extend the CAPO directives browser to allow the user inspection of all multilevel parallel loops and possibly perform code transformations or disable nested 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 [23] KAI's toolkit [15], VAST/Parallel [20], 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 [22]). 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 [18], 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.
- 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.

OpenMP extensions are currently being implemented in the framework of the NanosCompiler to easily specify precedence relations causing pipelined executions. 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).
- The ability to specify predecessor and successor relationships between named work-sharing constructs (PREC 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]. These relationships can cross the boundaries of parallel loops and therefore avoid the problems that CAPO currently has to implement two-dimensional pipelines.

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 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 and by the Spanish Ministry of Science and Technology under contract CICYT 98-511.

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

[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 issue, 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. pp. 1205-1218. October 2000.

[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). pp. 317-331. August, 2000.

[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, 197-226, 1996.

[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). pp. 294-301. June 1999.

[18] Omni: RCWP OpenMP Compiler Project, http://pdplab.trc.rwcp.or.jp/pdperf/Omni.

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

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

[21] 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.

[22] 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.

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

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