© 2001 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes,creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. http://dx.doi.org/10.1109/ICPP.2001.952074

Complex Pipelined Executions in OpenMP Parallel Applications

M. Gonzalez, E. Ayguade, X. Martorell and J. Labarta
Computer Architecture Department, Technical University of Catalonia,
cr. Jordi Girona 1-3, Mòdul D6, 08034 - Barcelona, Spain

Abstract

This paper proposes a set of extensions to the OpenMP programming model to express complex pipelined computations. This is accomplished by defining, in the form of directives, precedence relations among the tasks originated from work–sharing constructs. The proposal is based on the definition of a name space that identifies the work parceled out by these work–sharing constructs. Then the programmer defines the precedence relations using this name space. This relieves the programmer from the burden of defining complex synchronization data structures and the insertion of explicit synchronization actions in the program that make the program difficult to understand and maintain. This work is transparently done by the compiler with the support of the OpenMP runtime library. The proposal is motivated and evaluated with a synthetic multi-block example. The paper also includes a description of the compiler and run–time support in the framework of the NanosCompiler for OpenMP.

1 Introduction

Parallel processing is being accepted by the computer industry as the path to increase the computational power of low–end workstations and even personal computers. Parallel architectures, ranging from multiprocessor workstations (with 2 to 4 processors) to medium scale shared–memory systems (up to 64 processors) are becoming more and more affordable and common for the development of computing-demanding applications.

However, making these parallel machines truly usable requires easy–to–understand and portable programming models that allow the exploitation of parallelism out of applications written in standard high–level languages. These programming models usually offer new mechanisms or extensions to the sequential language to express the parallelism available in the application. For instance, OpenMP has emerged as the standard for shared–memory parallel programming. Its simplicity and portability across a range of parallel platforms are achieved without significantly sacrificing the performance on the parallel execution.

One of the features available in the current definition of OpenMP is the possibility of expressing multiple–levels of parallelism. This is achieved by nesting parallel constructs, which includes parallel loops and sections. However, the majority of current compiler implementations serialize nested parallel constructs. Once parallelism is activated, new opportunities for parallel work creation are ignored by the execution environment. Exploiting a single–level of parallelism may incur in low performance returns when the number of processors to run the application is increased.

There have been previous attempts in the past to approach the exploitation of multi–level parallelism. For instance, some focused on providing coordination support to allow the interaction of a set of program modules in the framework of data parallel programs for distributed memory architectures. For example, [3] proposed a library–based approach that provides a set of functions for coupling multiple HPF tasks to form task–parallel computations. Other alternatives [2, 6, 10] proposed a small set of Fortran directives to integrate task and data parallelism also in an HPF framework. Our group has proposed extensions to OpenMP in order to allow an efficient exploitation of nested parallelism. The extensions offer the concept of thread groups [5]. This concept is similar to the concept of processor subgroup proposed in similar frameworks (e.g. the FX compiler [6]). However, the definition of subgroups was static and closely related to the exploitation of task parallelism. Our extension to OpenMP allows the dynamic creation of thread groups and the definition of the actual composition of groups at runtime. Groups can be created to exploit both loop and task–level parallelism.

The specification of generic task graphs as well as complex pipelined structures is not an easy task in the framework of OpenMP. In a large number of scientific applications, the exploitation of parallelism requires the specification of a set of tasks and tight relationships among them, which usually crosses the boundaries of timestep loops leading to complex wavefronts. In order to exploit this parallelism, the programmer has to define complex synchronization data structures and use synchronization primitives

along the program, sacrificing readability and maintainability.

Previous researchers proposed to include in the language definition support for pipelining wavefront computations [1] and to express data parallel pipelines when integrating task and data parallelism [6]. In this paper we follow a similar approach and propose an extension to the OpenMP programming model that enables the specification of precedence relations among tasks originated from work–sharing constructs (parallel loops and sections).

In addition to having the appropriate support at the language level to express generic task graphs including pipelined schemes, good scheduling techniques are required to map the different levels of parallelism onto the available processors. A large number of heuristics have been proposed in the literature [12, 9, 10] for mixed task and data parallel scheduling. The proposal in this paper includes a set of new directives and clauses for the emerging indutrial standard OpenMP to specify generic task graphs and an associated processor mapping. The paper does not assume any particular scheduling heuristic.

The paper is organized as follows. Section 2 motivates the proposed extensions with an example. Section 3 overviews one of the proposed extensions (thread groups) and details the extension oriented towards the specification of precedence relations. Section 4 outlines some implementation details in the OpenMP NanosCompiler for Fortran [4]. Section 5 presents experimental results for the motivating example. Finally, Section 6 summarizes the paper.

2 Motivating Example

In this section we describe an example that motivates the extensions to OpenMP proposed in this paper. The example is based on a multi-block code that performs a specific computation over a number of rectangular blocks. Blocks may have different size (defined by vectors `nx`, `ny` and `nz`). The computation on one block may depend on the computation on other spatially contiguous blocks. Blocks are stored consecutively in a vector named `a`. An outline of the kernel is shown in Figure 1.

The kernel has two different phases. An initialization phase (lines 1–18) where all data is read, including the sizes of the blocks and their relationships. The second phase (lines 19–34) contains the solver, which consists of an iterative time-step loop that performs the same computation (inside routine `solve` in lines 35–46) on each point of each block. The initialization is done exploiting two levels of parallelism. The outer level exploits the parallelism at the level of independent blocks (loop `ib`). The inner level of parallelism is exploited in the initialization of the elements that compose each block. The same multilevel structure appears in the solver (computation for each block and computation within each block). However, the interaction between blocks forces their sequential execution. In fact, some parallelism could be exploited if one pipelines the execution of dependent blocks, thus creating a wavefront of independent computations that advances as soon as one block is computed.

```
1   C Initialize blocks and dependence relations
2         ...
3         read *, nblock
4         read *, (nx(i), ny(i), nz(i),i=1,nblock)
5         do ib = 1, nblock
6           read *, numpred(ib)
7           read *,(listpred(ib,i),i=1,numpred(ib))
8           read *, numsucc(ib)
9           read *,(listsucc(ib,i),i=1,numsucc(ib))
10          ...
11          work(ib) = nx(ib) * ny(ib) * nz(ib)
12        enddo
13        ...
14  C Computation of thread groups according to the
15  C weight of blocks and precedence relations
16        call compute_groups(nblock, work, numsucc,
17       + listsucc,masters, howmany)
18        ...
19  C Solver part of the kernel
20  10    continue
21  C$OMP  PARALLEL DO SCHEDULE(STATIC)
22  C$OMP& GROUPS(nblock, masters, howmany)
23  C$OMP& PRED(numpred(ib),listpred(ib))
24  C$OMP& SUCC(numsucc(ib),listsucc(ib))
25        do ib = 1, nblock
26            call solve(a(loc(ib)),
27       +         nx(ib), ny(ib), nz(ib),...)
28            ...
29        enddo
30  C$OMP END PARALLEL DO
31        if ... goto 10
32        ...
33  C Final updates and write results
34        ...
35        subroutine solve(t,nx,ny,nz, ...)
36        ...
37  C$OMP PARALLEL DO SCHEDULE(STATIC)
38  C$OMP& PRIVATE(i,j,k)
39        do 10 k=1,nz
40          do 10 j=1,ny
41            do 10 i=1,nx
42               ...
43  10    continue
44  C$OMP END PARALLEL DO
45        ...
46        end
```

Figure 1. Source code for the motivating example.

Exploiting a single level of parallelism in this code leads to an exploitation of limited parallelism. The exploitation of the outer level (i.e. blocks) is limited to the maximum number of blocks. In addition, if blocks have not the same size, the largest one will determine the execution time and thus reduce the maximum achievable parallelism (i.e. 8). The exploitation of the inner parallelism (inside function solve) can theoretically reach the maximum parallelism (i.e. number of processors). However, the efficiency of the parallelism is clearly reduced when the amount of work per processor in any of the blocks is not sufficient to compensate parallelism overheads. As we will observe in Section 5 for this kernel and a specific input file, this level of parallelism

is not able to efficiently use more than 16 processors. When more processors are available, it is necessary to combine the exploitation of both levels of parallelism.

In the main program shown in Figure 1 the reader can observe the two levels of parallelism in the computational phase of the kernel (i.e. `PARALLEL DO` constructs in lines 21 and 37). The following extensions to OpenMP are used for the outermost parallel loop:

- Definition of groups of threads. The computation of a block is assigned to a group of threads, whose size is proportional to the amount of computation inside the block. As many groups as number of blocks are defined.
- Definition of precedences between groups of threads. These precedences will force the correct relative ordering among dependent blocks.

Next we consider each aspect in turn. For instance, assume that vector `work` is initialized as follows:

```
nblock = 8
work[8] = {8192, 4096, 1024, 4096,
           1024, 1024, 1024, 1024}
```

If no precedences among blocks were specified, a simple directive in line 22 like:

```
22      C$OMP& GROUPS(nblock, work)
```

would force the runtime to distribute the available processors among the groups using a default allocation scheme [5] that considers the numbers in vector `work` as proportions and ensures that each group at least receives one. So for instance, if 16 processors were available, the small groups would receive one processor each, the medium–size groups would receive 3 and the largest one 5 processors. This allocation achieves a theoretical speed–up of 13.125. In this allocation, the processors executing small blocks finish their computation before the other ones. Other allocations that cluster small blocks and force them to share processors may improve the efficiency [5].

However when precedences among blocks exist this may lead to idle processors (i.e. just waiting for other blocks to complete) as shown in Figure 2.b when the precedences graph shown in Figure 2.a is considered (thus reducing the theoretical speed–up to 4.25). For this reason, a different strategy for the composition for thread groups, that takes into account the precedence relations, must be used. For example this is included in the source code in Figure 1 when the main program invokes routine `compute_groups` (lines 14–18). This user–defined routine is in charge of determining, at runtime and according to the total number of processors available, the actual composition of each group of threads. For this purpose, it uses the size of each block (vector `work`) and the list of successors for each block (vector `numsucc` and matrix `listsucc`). Different heuristics have been proposed in the literature for doing this assignment [9, 12, 10].

For instance, assume that the following list of successors is specified (corresponding to the graph shown in Figure 2.a:

```
numsucc[8] = {2, 1, 0, 2, 1, 0, 1, 0}
listsucc[8,2] = {2, 4, 3, 0, 0, 0, 5, 7,
                 6, 0, 0, 0, 8, 0, 0, 0}
```

In this case, for instance block number 4 has two successors (as indicated by `numsucc[4]=2`): blocks 5 and 7 (as indicated by the two valid entries in `listsucc[4]={5, 7}`). A possible allocation strategy could end-up clustering groups as shown in Figure 2.c. In this allocation, first block `1` is executed using all the 16 processors. After that, the allocation of processors to groups is done considering two clusters: one composed of blocks `2--3`, and the other composed of groups `4--8`. The weight of the clusters is 5120 and 8192, respectively. So at this point the allocation of processors is done considering these values as proportions, ending up with 6 and 10 processors, respectively. In this second cluster, the execution can be further divided so that block 4 starts first using all the 10 processors; after that, two sub-clusters are considered including blocks `5--6` and `7--8`. Each block in these sub-clusters will receive half of the processors available in the cluster (i.e. 5 processors). This allocation achieves a theoretical speed–up of 15.75.

The specification of this composition for the groups of threads is easy to perform by using one of the clauses proposed as part of our OpenMP extensions:

```
22      C$OMP& GROUPS(nblock, masters, howmany)
```

Vector `masters` specifies the thread that will behave as master of the group. Vector `howmany` specifies the number of threads used in the group. For the allocation shown in Figure 2.c, routine `compute_groups` should return:

```
masters[8]={0, 0, 0, 6, 6, 6, 11, 11}
howmany[8]={16, 6, 6, 10, 5, 5, 5, 5}
```

A set of predefined `compute_groups` routines could be offered by the runtime system, or the user could simply define its own ones. This topic is not considered in this paper. Several solutions to the problem of automatic scheduling of task and data parallelism have been published elsewhere [12, 9, 10].

3 Extensions to OpenMP

In the fork/join execution model defined by OpenMP [8], 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 enclosed lexically within the parallel constructs. 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

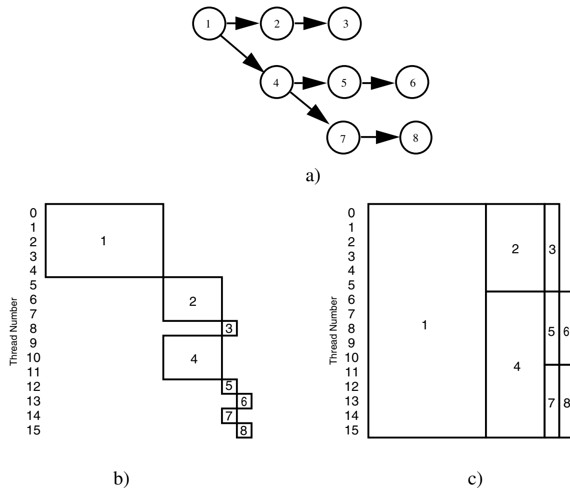

Figure 2. a) Precedence graph, b) initial thread allocation, c) final thread allocation.

by the BARRIER directive). Exclusive execution mode is also possible through the definition of CRITICAL regions.

Next we describe the two extensions proposed to support the specification of complex pipelines that include tasks generated from OpenMP work–sharing constructs. The extensions are in the framework of nested parallelism and target the pipelined execution at the outer levels.

3.1 Extension 1: Thread Groups

In our proposal, a *group of threads* is composed of a subset of the total number of threads available in the team to run a parallel construct. We restrict that the threads are consecutive following the active numeration inside the current team (from 0 to omp_get_num_threads()-1). 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 groups are defined; the rest of threads are reserved to support the execution of nested parallel constructs. In other words, the groups definition establishes the threads that are involved in the execution of the parallel construct plus an allocation strategy or scenario for the inner levels of parallelism that might be spawned.

The GROUPS clause allows the user to specify the definition of thread groups. It can only appear in a PARALLEL construct or combined PARALLEL DO and PARALLEL SECTIONS constructs.

```
C$OMP PARALLEL [DO|SECTIONS] [GROUPS(gspec)]
```

Different formats for the groups specifier gspec are allowed [5]. In this paper we only comment the ones closely related with the precedences proposal. For additional details concerning alternative formats as well as implementation issues, please refer to this publication.

```
GROUPS(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 runtime is in charge of computing the two previous vectors (masters and howmany). This eases the use of groups because the programmer is relieved from the task of determining their exact composition.

The most general format allows the specification of three parameters in the group definition: the number of groups, the identifiers of the threads that participate in the execution of the parallel region, and the number of threads composing each group:

```
GROUPS(ngroups, masters, howmany)
```

The first argument (ngroups) specifies the number of groups to be defined and consequently the number of threads in the team that is going to execute the parallel construct. The second argument (masters) is an integer vector with the identifiers (using the active numeration in the current team) of the threads that will compose the new team. Finally, the third argument (howmany) is an integer vector whose elements indicate the number of threads that will compose each group. The vectors have to be allocated in the memory space of the application and their content and correctness have to be guaranteed by the programmer.

3.2 Extension 2: Precedence Relations

Next we present an extension to the OpenMP programming model that allows the specification of precedence relations among the threads that participate in the execution of a parallel construct. The proposal is divided in two parts. The first one consists in the definition of a name space for the tasks generated by the OpenMP work–sharing constructs. The second one consists in the definition of precedence relations among those named tasks.

3.2.1 The NAME clause

The NAME clause is used to provide a name to a task that comes out of a work–sharing construct. In a SECTIONS work–sharing construct, the NAME directive is used to identify each SECTION:

```
C$OMP SECTIONS
C$OMP SECTION NAME(name_ident)
...
C$OMP SECTION NAME(name_ident)
...
C$OMP END SECTIONS
```

The name_ident identifier is supplied by the programmer and follows the same rules that are used to define variable and constant identifiers.

In a DO work–sharing construct, the NAME clause only provides a name to the whole loop:

```
C$OMP DO NAME(name_ident)
...
C$OMP END DO
```

The number of tasks associated to a DO work–sharing construct is not determined until the associated do statement is going to be executed. Depending on the number of available threads and the chunk size applied in the loop scheduling, the loop is broken into a different number of parallel tasks. We propose to identify each iteration of the parallelized loop by the identifier supplied in the NAME clause plus the value of the loop induction variable for that iteration. This means that the name space for a parallel loop will be big enough to name each of the iterations of the loop. The programmer simply defines the precedences at the iteration level. These precedences are then translated to task precedences, depending on the SCHEDULE strategy specified to distribute iterations to tasks.

3.2.2 The PRED and SUCC clauses and directives

Once a name space has been created, the programmer is able to specify a precedence relation between two tasks using their names. This is done by the use of the PRED and SUCC clauses or directives:

```
[C$OMP] PRED(task_id[,task_id]*) [IF(exp)]
[C$OMP] SUCC(task_id[,task_id]*) [IF(exp)]
```

PRED is used to list all the tasks names that must complete their execution before executing the one affected by it. The SUCC directive is used to define all those tasks that, at this point, may continue their execution. The IF clause is used like the already existent clause in the OpenMP programming model. Expression exp is evaluated at run–time in order to obtain a boolean value that determines if the associated PRED or SUCC clause applies.

As clauses, PRED and SUCC apply at the beginning and end of a task (because they appear as part of the definition of the work–sharing itself), respectively. The same keywords can also be used as directives, in which case they specify the point in the source program where the precedence relationship has to be fulfilled. Code before a PRED directive can be executed without waiting for the predecessor tasks. Code after a SUCC directive can be executed in parallel with the successor tasks.

The PRED and SUCC constructs always apply inside the nearest work–sharing construct where they appear. Any work–sharing construct affected by a precedence clause or directive has to be named with a NAME clause.

The task_id identifier is used to identify the parallel task affected by a precedence definition or release. Depending on the work–sharing construct where the parallel task was coming out from, the task_id identifier presents three different formats:

```
task_id = (name_ident) |
          (name_ident,expr) |
          (name_ident,expr,expr)
```

When the task_id is only composed of a name_ident identifier, the parallel task corresponds to a task coming out from a SECTIONS work–sharing construct. In this case, the name_ident corresponds to an identifier supplied in a NAME clause that annotates a SECTION construct. When the name_ident is followed by one expression, the parallel task corresponds to a chunk of iterations coming from a parallelized loop. The expression must include the loop induction variable and its evaluation must result in an integer value identifying a specific iteration of the loop. The precedence relation is defined with the chunk of iterations containing the iteration indicated by expr. When two expressions are supplied, the task_id syntax allows the programmer to specify a list of predecessor chunks. The first expression indicates the number of parallel tasks that the programmer wants to name. The second expression supplies a pointer reference pointing to a vector containing different values of the iteration space of the loop from where the parallel tasks are coming out. In order to identify the tasks, each value is translated to the loop chunk executing the iteration corresponding to it.

Figure 3 shows a code fragment in which a precedence relation between a SECTION and a DO work–sharing construct is established. In particular, the precedence relation only involves one iteration of the loop, as indicated with the IF clause in the PRED directive. In this case, all iterations can proceed in parallel with the code in section A except for iteration iter. CODE_5 in this iteration will have to wait for the completion of CODE_1 in named section A. Notice that in the PRED clause, the user specifies the successor task using the name of the loop and the specific iteration.

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION NAME (A)
      CODE_1
!$OMP SUCC(loop, iter)
      CODE_2
!$OMP SECTION
      CODE_3
!$OMP END SECTIONS NOWAIT
!$OMP DO NAME (loop)
      do i = 1, N
         CODE_4
!$OMP PRED (A) IF(i.eq.iter)
         CODE_5
      enddo
!$OMP END PARALLEL
```

Figure 3. Precedence between SECTIONS and DO.

Line 23 in Figure 1 shows the specification of precedences for the multi-block kernel. In this case the programmer uses the most complex format for clauses `PRED` and `SUCC`: an expression that evaluates to the number of precedences for the block (in this case `numpred(ib)` or `numsucc(ib)`, respectively) and a pointer to the list of predecessors or successors for the block. As shown in the Figure, the composition of these lists can be computed at runtime.

4 The Run-Time support

In this section we describe the support required from the runtime system to efficiently implement the language extensions to specify precedence relations. The runtime systems usually offer mechanisms to guarantee exclusive execution (critical regions), ordered executions (ticketing) and global synchronizations (barriers). The proposal in this paper requires explicit point–to–point synchronization mechanisms. The description of the runtime support for multiple levels of parallelism and thread groups is not included in this paper and can be found elsewhere [5].

Two main aspects have to be considered to provide support to the programming model defined in section 3. First, the runtime has to provide a mechanism to synchronize two threads according to the precedences introduced by the programmer. Second, there must be a translation mechanism that allows to dynamically identify the thread executing a chunk of iterations in a parallel loop; this means that the runtime has to be able to identify which thread is executing which iteration of the loop.

4.1 Thread Synchronization

Our approach is based on the definition of an address space where threads involved in a precedence relation synchronize. For each pair of named tasks related with precedences, a memory location is allocated. Threads executing these tasks will use this memory location to communicate. This memory location is considered as a dependence counter. The definition of a precedence at runtime implies an increment of this counter. The release of a precedence implies a decrement of the same counter. When a thread checks if a precedence has been released, it spins until the counter reaches zero. This counter is contained in what we name the precedence descriptor (described later).

Two routines are provided to define/release precedences at runtime: `nthf_def_prec` and `nthf_free_prec`, respectively. The main argument for these routines is a precedence descriptor. For each `PRED` directive/clause, the compiler injects a call to routine `nthf_def_prec`. This routine increments the counter contained in the precedence descriptor and spins until the counter reaches zero. For each `SUCC` directive/clause, the compiler injects a call to routine `nthf_free_prec`. This routine mainly decrements the counter contained in the precedence descriptor.

Figure 4 shows the code generated by the compiler for the OpenMP fragment shown in Figure 3. Notice that function `name_A` encapsulating the code for named section `A` performs a call to routine `nthf_free_prec` in order to decrement the associated counter. Similarly, routine `loop` invokes `nthf_def_prec` in order to increment the counter. The thread invoking this routine will wait until the counter reaches zero. Both routines receive as an argument the precedence descriptor that contains the counter.

When the precedence relation involves threads executing tasks (chunks) of a `DO` work–sharing construct, the number of counters is determined at runtime. In particular, as many counters as pairs consumer/producer need to be allocated. For instance, the code shown in Figure 3 establishes a precedence between a `SECTION` and a `DO` work–sharing construct; in this case, as many counters as the number of threads participating in the loop execution are allocated in the precedence descriptor. These counters are used to synchronize the thread executing the section with each possible thread executing a chunk of the `DO` loop. In addition to that, the compiler needs to insert code to perform a translation from the iteration space to the thread space. This translation (explained later in this section) determines which thread executes a particular iteration of the loop.

In the general case, the precedence relation may involve tasks (chunks) generated from two `DO` work–sharing constructs. In this case, a matrix of counters is allocated (with as many rows and columns as the number of threads) in the precedence descriptor.

A routine `nthf_init_prec` is offered by the runtime to allocate and define the precedence descriptor for each of the above mentioned situations.

4.2 Iteration–Thread Translation

In this section we describe the basic data structure and service available to perform the translation between iteration number and thread executing the iteration. We will focus on the information that is needed at run–time and how this information is used to compute the translation. For each parallelized loop involved in a precedence relation, a loop descriptor is created with the following information: lower and upper bound for the induction variable, iteration step, the scheduling applied and the number of threads currently executing the loop. This loop descriptor is allocated in the application address space. Once the master thread of the group reaches a particular loop, its loop descriptor is initialized with all the information mentioned above.

Routine `nthf_index_to_thread` is in charge of the translation. The main arguments of this routine are a loop

```
subroutine name_A(prec_A_loop, ..., loop_desc, ...)
CODE_1
nth_id = nthf_index_to_thread(loop_desc, iter, ...)
call nthf_free_prec(prec_A_loop, nth_id, ...)
CODE_2
end
                      a)

subroutine loop(prec_A_loop, iter, ...)
nth_whoami = ...
nth_min = ...
nth_max = ...
do i = nth_min, nth_max
  CODE_4
  if (i.eq.iter) then
     call nthf_def_prec(prec_A_loop, nth_whoami,...)
  end if
  CODE_5
enddo
end
                      b)
```

Figure 4. Code generated for the example in Figure 3

descriptor and an iteration index. The output of the routine is the identifier of the thread executing the iteration, which is computed according to the information contained in the loop descriptor.

For example, Figure 4 shows the code generated by the compiler for the example in Figure 3. In the code generated for section `name_A`, the compiler injects a call to routine `nthf_index_to_thread` in order to know the identifier of the thread executing iteration `iter`. After that, the thread invokes `nthf_free_prec` over the corresponding element of the precedence descriptor. For the threads executing the loop, notice that the call to routine `nth_def_prec` will be done by the thread executing iteration `iter`. The invocation to `nthf_free_prec` is done with the identifier `nth_whoami` of the thread executing that iteration.

5 Experimental evaluation

In this section we evaluate the behavior of four parallel versions of the motivating example in Section 2. The experiments have been performed on a Silicon Graphics Origin2000 system [11] with 64 R10k processors, running at 250 MHz with 4 Mb of secondary cache each. For all compilations we use the NanosCompiler to translate from extended OpenMP Fortran77 to plain Fortran77 with calls to the supporting runtime library NthLib. We use the native `f77` compiler to generate code for the Origin system. The flags are set to `-64 -Ofast=ip27 -LNO:prefetch_ahead=1:auto_dist=on`.

The four parallel versions analyzed are: *1L–inner*, *2–levels*, *2L–groups* and *2L–precedences*. The *1L–inner* version exploits a single level parallelization. In this version the intra–block parallelism is exploited (i.e. the computation of the different blocks is serialized). The *2–levels* version exploits both levels of parallelism. This version is fully compatible with the current definition of OpenMP (i.e. does not use any of the extensions proposed in Section 3). The *2L–groups* version also exploits the two levels of parallelism but includes the definition of thread groups. These two versions assume that no precedence relations exist among blocks. They are useful to observe the improvement due to the use of the thread groups extension. Finally, the *2L–precedences* version is equivalent to *2L–groups* but taking into account the precedence relations among blocks. Eight blocks compose the input of the different versions. Two large blocks (large blocks are 8 times larger than small blocks) and six small blocks. When precedences are specified, two independent branches, each composed of the serial execution of one large and three small blocks, are defined.

Figure 5 shows the speed–up of the four parallel versions with respect to the original sequential version. The following conclusions are drawn from this figure:

- *1L–inner* is not able to efficiently use more than 16 processors. The parallelization overheads are not compensated for the smallest groups, causing a degradation in the parallelization efficiency.

- *2–levels* improves the behavior with respect to *1L–inner* but the efficiency of the parallelization also falls down above 32 processors. Notice that this version suffers from the same problem than *1L–inner*: large parallelization overheads compared to the amount of work per processor. The improvement is mainly due to an effective reduction of the time threads spent waiting on barrier synchronizations. In the *1L–inner* version, threads must wait until all of them finish the computation of a block (implicit barrier at the end of the `PARALLEL` construct). After that, the master thread generates the parallelism for the next block. However, in the *2–levels* the master thread initially generates the parallelism for the outer level. This means that threads (8 in this case) start executing the computation in a block and find the inner level of parallelism. Then they simultaneously generate the work for all the available threads. So notice that at this point (before starting the computation inside the main loop in function `solve`), all the work has been generated. Therefore, when a thread finishes with the computation of a chunk of iterations from a block, it enters the barrier but immediately finds more work to execute and delays the execution of the barrier.

- *2L–groups* performs better when more than 8 processors are available. In this version the work in the inner level of parallelism is distributed following the groups specification. Eight processors are devoted to the exploitation of the outer level of parallelism. The rest of processors are evenly distributed to exploit the inner level of parallelism. Notice that the number of processors devoted to execute a block is proportional to the

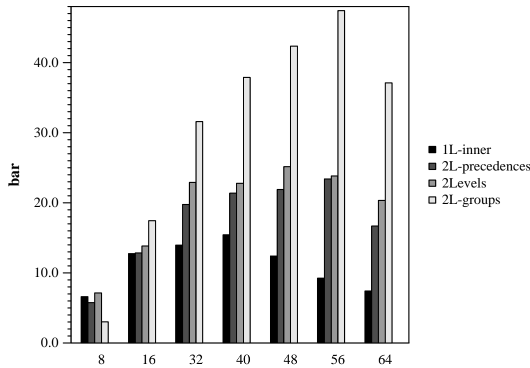

Figure 5. Four multi-block kernel versions (speed–up).

size of block, so that small blocks are executed with less processors. This improves the efficiency of the parallelization and results in noticeable reductions in the total execution time.

- The *2L–precedences* version should be only compared with the *1L–inner* version; the *2Levels* and *2L–groups* versions the outer level of parallelism is assumed to be free of dependences. Notice that this version outperforms *1L–inner* when more that 8 processors are used. In this case, the combination of thread groups and the execution of blocks following the precedence relations expressed by the user results in a noticeable reduction in the execution time. This version is always worse than *2L–groups* because of the limited parallelism inherently available at the outer level.

6 Conclusions

In this paper we have presented a set of extensions to the OpenMP programming model oriented towards the specification of complex pipelined computations in the context of multilevel parallelism exploitation. The proposal relieves the programmer from the burden of defining complex synchronization data structures and the insertion of explicit synchronization actions in the program that make the program difficult to understand and maintain. This work is transparently done by the compiler with the support of the OpenMP runtime library. Although the majority of the current systems only support the exploitation of single–level parallelism around loops, we believe that multi–level parallelism will play an important role in future systems. In order to exploit multiple levels of parallelism, several programming models can be combined (e.g. message passing and OpenMP). We believe that a single programming paradigm should be used and should provide similar performance.

The extensions have been implemented in the NanosCompiler and runtime library NthLib. We have coded a multi–block application using these extensions and analyzed the performance on a Origin2000 platform. The results show that when the number of processors is high, exploiting multiple levels of parallelism with thread groups results in better work distribution strategies and thus higher speed–up than both the single level version and the multilevel version without groups. When precedences are taken into consideration, the mechanism proposed at the language level and its implementation in the runtime library are powerful enough to express a variety of scientific applications.

Acknowledgments

This research has been supported by the Ministry of Education of Spain under contract TIC98-511 and the CEPBA (European Center for Parallelism of Barcelona).

References

[1] B. Chamberlain, C. Lewis and L. Snyder. Array Language Support for Wavefront and Pipelined Computations In *Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[2] I. Foster, B. Avalani, A. Choudhary and M. Xu. A Compilation System that Integrates High Performance Fortran and Fortran M. In *Scalable High Performance Computing Conference*, Knoxville (TN), May 1994.

[3] I. Foster, D.R. Kohr, R. Krishnaiyer and A. Choudhary. Double Standards: Bringing Task Parallelism to HPF Via the Message Passing Interface. In *Supercomputing'96*, November 1996.

[4] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. Concurrency: Practice and Experience. To appear vol.12, no. 9, August 2000.

[5] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta and N. Navarro. OpenMP Extensions for Thread Groups and Their Runtime Support. In *Workshop on Languages and Compilers for Parallel Computing*, August 2000.

[6] T. Gross, D. O'Halloran and J. Subhlok. Task Parallelism in a High Performance Fortran Framework. In IEEE Parallel and Distributed Technology, vol.2, no.3, Fall 1994.

[7] X. Martorell, E. Ayguadé, J.I. Navarro, J. Corbalán, M. González and J. Labarta. Thread Fork/join Techniques for Multi–level Parallelism Exploitation in NUMA Multiprocessors. In *13th Int. Conference on Supercomputing ICS'99*, Rhodes (Greece), June 1999.

[8] OpenMP Organization. OpenMP Fortran Application Interface, v. 2.0, www.openmp.org, June 2000.

[9] A. Radulescu, C. Nicolescu, A.J.C. van Gemund, and P.P Jonker CPR: Mixed task and data parallel scheduling for distributed systems. 15th International Parallel and Distributed Processing Symposium (IPDPS'2001), Apr. 2001

[10] S. Ramaswamy. Simultaneous Exploitation of Task and Data Parallelism in Regular Scientific Computations. Ph.D. Thesis, University of Illinois at Urbana–Champaign, 1996.

[11] Silicon Graphics Computer Systems SGI. Origin 200 and Origin 2000 Technical Report, 1996.

[12] J.Subhlok, J.M. Stichnoth, D.R O'Hallaron, and T. Gross. Optimal use of mixed task and data parallelism for pipelined computations. Journal of Parallel and Distributed Computing, 60:297-319, 2000