Precedence Relations in the OpenMP Programming Model

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

Abstract

In this paper, we propose an extension to the OpenMP programming model in order to express precedence relations among tasks originated from work–sharing constructs. Our proposal is based on the definition of a name space that identifies the work parceled out by the work–sharing construct. Then the programmer defines the precedence relations using this name space. The proposal is described with a set of synthetic examples and a `LU` kernel. Four different parallel strategies for this `LU` kernel are studied and compared. The paper shows that the amount of parallelism that can be exploited using the new directives and clauses is larger than the parallelism that is exploited using the current OpenMP specification.

1 Introduction

Parallel architectures are becoming affordable and common platforms for the development of computing-demanding applications. Users of such architectures require simple and powerful programming models to develop and tune their parallel applications with reasonable effort. These programming models are usually offered as library implementations or extensions to sequential languages that express the available parallelism in the application. Standard language extensions are defined by means of directives and language constructs (e.g. OpenMP [4], which is the emerging standard for shared–memory parallel programming).

In general, the parallelism that the user can express using these language extensions corresponds to simple task graphs, with loosely coupled tasks. Numerical codes in science and engineering usually present enough parallelism to consider their parallelization, but usually the available parallelism is organized in complex task graphs with tight relationships among the tasks. This parallelism is not directly expressable with the annotations in these language extensions. For example, pipelined execution schemes are very common in these applications, and represent an important example of the aforementioned lost of parallelism caused by the limitations of the parallel language extensions. The available parallelism in such codes depends on the precedence relations that may exist between the tasks involved in the pipelined structure in successive iterations of an iterative outer loop.

The programmer has to define complex synchronization data structures and statements in order to exploit this parallelism. The proposal in this paper tries to face this problem, proposing an extension to the OpenMP programming model in order to express precedence relations among tasks originated from work–sharing constructs. The proposal is based on the definition of a name space that identifies the work parceled out by the work–sharing construct. Then the programmer defines the precedence relations using this name space.

Other research projects targeting the exploitation of multiple levels of parallelism focus in integrating task and data parallelism using different programming models (MPI+OpenMP, HPF, etc.) [2, 5].

The paper is organized as follows. Section 2 describes the extensions to the OpenMP programming model proposed in this paper when dealing with a single level of parallelism. This section uses a number of synthetic examples to present the syntax and semantics of the proposed extensions. Section 3 adds the support for multiple levels of parallelism. Section 4 applies the proposed extensions to a `LU` kernel, and analyzes different parallelization strategies that are originated when using the extensions. Finally, conclusions are presented in Section 5.

2 Extension to OpenMP

In this section, 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.

In the fork/join execution model defined by OpenMP [4], 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 by the BARRIER directive). Exclusive execution mode is also possible through the definition of CRITICAL regions.

The SECTIONS directive is a non–iterative work–sharing construct that specifies that the enclosed sections of code (each one delimited by a SECTION directive) are to be divided among threads in the team. Each section becomes a task which is executed once by a thread in the team. The DO work–sharing construct is used to divide the iterations of a loop into a set of independent tasks, each one executing a chunk of consecutive iterations.

Our proposal can be divided into 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.

2.1 The NAME clause

The NAME directive 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 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 directive 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 iterations that each task is executing.

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 or end of a task (because they appear as part of the definition of the work–sharing itself). The same keywords can also be used as directives, in which case they specify where the precedence relation 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 two different syntax:

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

When a task_id is only composed by a name_ident identifier, the parallel task corresponds to a task coming out from a SECTIONS work–sharing construct. In that case, the name_ident corresponds to an identifier supplied in a clause NAME that is annotating a SECTION construct. When the name_ident is followed by an expression, the parallel task corresponds to a chunk of work of a parallelized loop. The evaluation of the supplied expres-

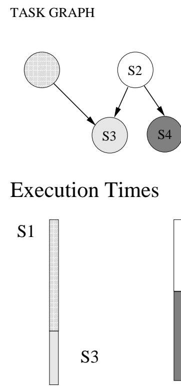

Figure 1: Task graph for Example 1.

sion must result on an integer value that is identifying a concrete iteration of the loop. The precedence relation is defined to the chunk of work containing the iteration pointed out by the expression evaluation. In order to specify precedences over all the chunks of the parallelized loop, the programmer is allowed to substitute the expression by the `ALL` clause. Any `(name_ident,ALL)` construct is indicating that one of the work–sharing constructs involved in the precedence relation is a `DO` construct, and that all of the loop chunks of work are defining/releasing a precedence.

Some simplifications are possible to ease the process of writing parallel code. First of all, when `NAME` is used as a clause of a `SECTION`, the programmer may avoid writing the `SECTION` keyword. Second, the precedences between chunks of the same parallel loop, might be defined without supplying any name to the `DO` construct. The `PRED` and `SUCC` clauses and directives simply express the precedences between the iterations of the parallel loop. Finally, in some cases the specification of both `PRED` and `SUCC` is redundant; in these cases, the programmer may omit one of them.

In the rest of this section we provide a set of examples that will help to understand the use of the proposed extensions.

Example 1:

Figure 1 shows a task graph composed of four tasks and an estimation of their execution time. The specification of this task graph using OpenMP incurs some loss of parallelism, due to the barrier synchronization required to enforce some precedences (which is more restrictive than the synchronization required to enforce it). Figure 2 shows the resulting OpenMP code and the estimated execution time.

Figure 3 shows the specification of the task graph using the extensions proposed in this paper. Notice that each precedence edge in the original task graph

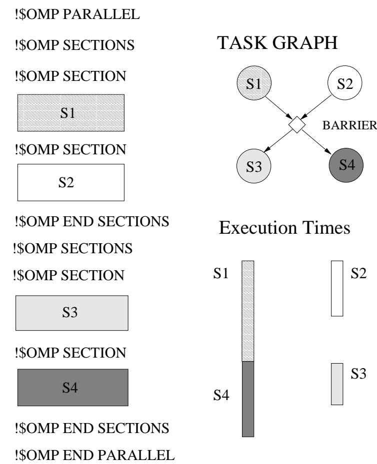

Figure 2: OpenMP annotations for Example 1.

has been translated into a `PRED` clause associated to a named section (`NAME` directive). The use of both `PRED` and `SUCC` is redundant in this case, so one of the two could be omitted.

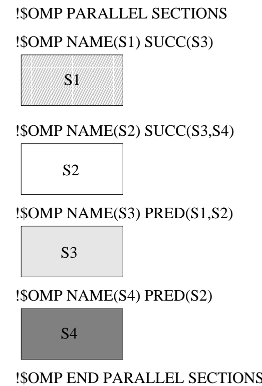

Figure 3: Extended OpenMP annotations for Example 1.

Example 2:

Figure 4 shows another task graph in which the precedence between tasks `S2` and `S4` is satisfied in the middle of the execution of task `S2`. Similarly, this precedence is defined in the middle of the execution of task `S4`. The use of the `PRED` and `SUCC` clauses incurs in some loss of parallelism.

In this case, the programmer could redefine the tasks by partitioning tasks `S2` and `S4` and specifying the apropiate precedences. The use of the `PRED` and `SUCC` directives allows a clean definition of the precedences while keeping the granularity of the initial tasks. Figure 4 shows the parallel annotations to exploit all the available parallelism.

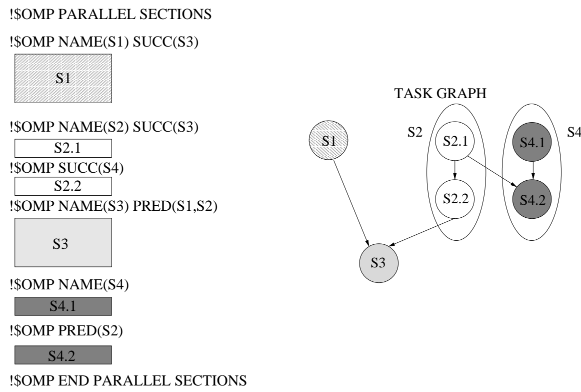

Figure 4: Task graph and extended OpenMP annotations for Example 2.

Example 3:

Figure 5 shows a task graph in which pairs of tasks synchronize at multiple points during their execution. In this case, the sections named `S1` and `S2` have more than one point where precedences are defined and released. The execution model for `PRED` and `SUCC` directives assumes that they match at runtime. In this example, the first point where a precedence is defined in `S2` matches the point where the first precedence is released in `S1`.

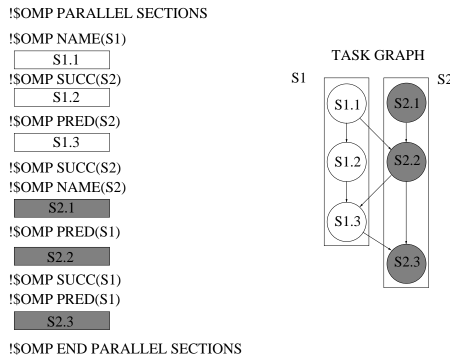

Figure 5: Task graph and extended OpenMP annotations for Example 3.

Example 4:

Consider the source code in figure 6. The first loop executes `N` iterations and the second loop executes `2*N` iterations. The two loops are sequential. However, some parallelism exists between the two loops. Assume that dependences imply that iteration `i` of the first loop has to precede the execution of iterations `2*i` and `2*i+1` of the second loop, as shown in the task graph in the same figure. The execution of these two loops can be easily pipelined with the use of the extensions proposed in this paper.

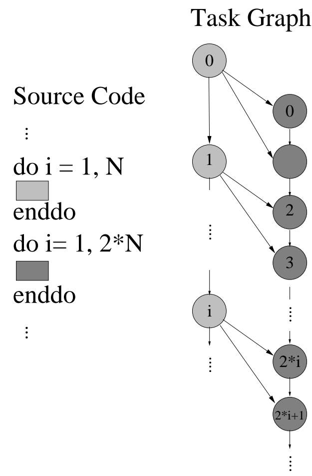

Figure 6: Source code and task graph for Example 4.

Figure 7 shows the parallel code annotated with the proposed extensions. In this case, `PRED` and `SUCC` clauses are included in an iterative `do` construct. As in the previous example, the effect of these directives is matched at runtime, ensuring that the precedence relations are correctly fulfilled. This requires the use of an `IF` clause to enforce the execution of the `SUCC` clause every two iterations of the second loop.

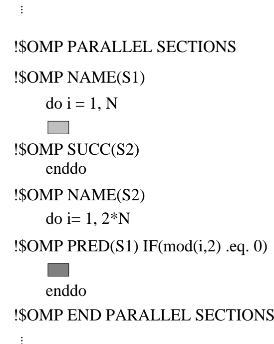

Figure 7: Extended OpenMP code for Example 4.

Example 5:

Figure 8 shows a loop with cross–iteration dependences. Although the loop is not completely parallel, it is possible to exploit some amout of parallelism if precedences are guaranteed.

Each iteration of the loop is identified with the value of the loop induction variable. To express a precedence, the clause `PRED` is provided with only one argument, an expression whose evaluation returns a value in the iteration space. This expression depends on the loop induction variable and determines the iteration that the current iteration

depends on. The precedences between iterations are translated to precedences between the chunks of work once the number of processors is known at runtime. This means that the programmer has not to be aware of how the work is going to be scheduled; only the precedences in the iteration space have to be supplied. Figure 9 shows the annotated OpenMP program using the extensions proposed in this paper. The PRED clause is used with one argument indicating, for each iteration i, which iteration precedes its execution.

3 Nested Parallel Constructs

In this section, we extend the previous proposal in order to handle situations in which nested parallelism is defined by the user. When the definition of precedences appear in the dynamic extend of a nested parallel region caused by an outer PARALLEL DO, multiple instances of the same name definition (given by a NAME clause/directive) exist. In order to differentiate them, the name_ident is extended with the induction variable of the parallel loop that causes the replication:

```
name_ident[:ind_var]+
```

Therefore, the task_id construct might take the following syntax:

```
task_id = (name_ident[:{expr|ALL}]*) |
          (name_ident[:{expr|ALL}]*,{expr|ALL})
```

The expression that is extending the name_ident construct will be evaluated at run–time. Its evaluation must result in an integer value contained in the iteration space of the parallelized loop that caused the multiple instances of the same name definition. The precedences apply to the chunk of work that will execute the iteration pointed out by the expression. As in the case of the precedence definition over chunks of work defined by a parallel loop,

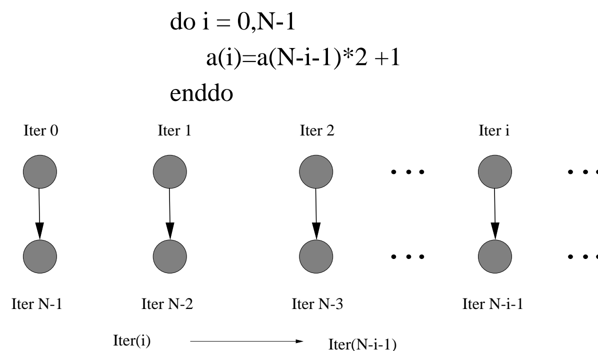

Figure 8: Precedences between loop iterations in Example 5.

```
!$OMP DO PRED(N-i) IF(i .gt. N/2)
    do i = 0,N-1
      a(i)=a(N-i-1)*2 +1
    enddo
!$OMP ENDDO
```

Figure 9: Extended OpenMP annotations for Example 5.

the ALL clause might be used to refer to all the parallel tasks generated by the multiple instances of the same named work–sharing construct. Example 8 will present this situation. Notice that the name_ident might be extended with more than one loop induction variable, depending on the loop nesting level of the NAME clause where the name_ident appeared.

Example 6:

Consider the code in figure 10 and its associated task graph. The body of the loop consists of two function calls (foo1 and foo2). As it is shown in the task graph, both function invocations can be executed in parallel. The precedence between the execution of foo1 in iteration k and the execution of the same function in iteration k+1 precludes the parallel execution of the loop. Figure 11 shows the possible annotations with the current OpenMP set of directives. Notice that only the inner level of parallelism is exploited, while the do statement remains not parallelized. The corresponding task graph to this parallelization is also shown in the same figure.

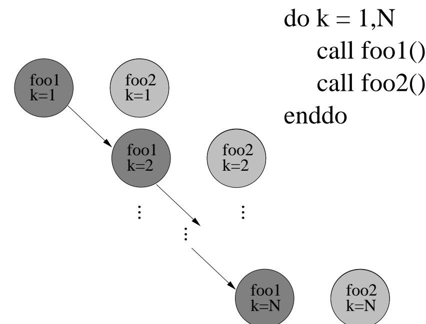

Figure 10: Source code and task graph for Example 6.

However, it is possible to pipeline the execution of the iterations of the loop if we provide the mechanisms to enforce this precedence. Figure 12 shows the annotated extended OpenMP code. Notice that the first task is identified with sect_foo1:k, being :k the replicator of the name sect_foo1. This name is used to specify the precedence relations through the PRED and SUCC clauses.

```
   do k = 1,N
!$OMP PARALLEL SECTIONS
!$OMP SECTION
     call foo1()
!$OMP SECTION
     call foo2()
!$OMP END PARALLEL SECTIONS
   enddo
```

Figure 11: OpenMP annotations for Example 6.

```
!$OMP PARALLEL DO
      do k = 1,N
!$OMP PARALLEL SECTIONS
!$OMP NAME(sect_foo1:k) PRED((sect_foo1:k-1)
!$OMP& SUCC((sect_foo1:k+1))
          call foo1()
!$OMP SECTION
          call foo2()
!$OMP END PARALLEL SECTIONS
      enddo
```

Figure 12: Extended OpenMP annotations for Example 6.

Example 7:

Figure 13 shows another example where the user can exploit multiple levels of parallelism. In this example, two execution paths exist. One of them works with array A and the other one works with array B; array elements computed in the first loop are used in the second loop. This originates a precedence relation between the chunks that appear in the parallelization of these two loops.

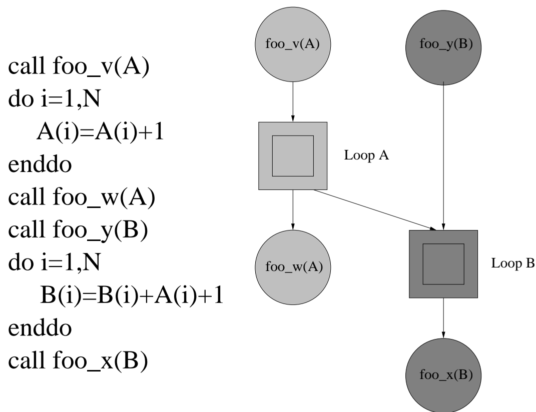

Figure 13: Source code and task graph for Example 7.

Figure 14 shows the annotated parallel code using the current OpenMP specification and the actual task graph that it represents. Notice that this parallel code exploits the parallelism between the calls to routines `foo_v` and `foo_y` working on matrices `A` and `B`. The same for the calls to routines `foo_w` and `foo_x`. The two parallel loops in the code have to be executed in sequence with the implicit barriers introduced by the parallel constructs.

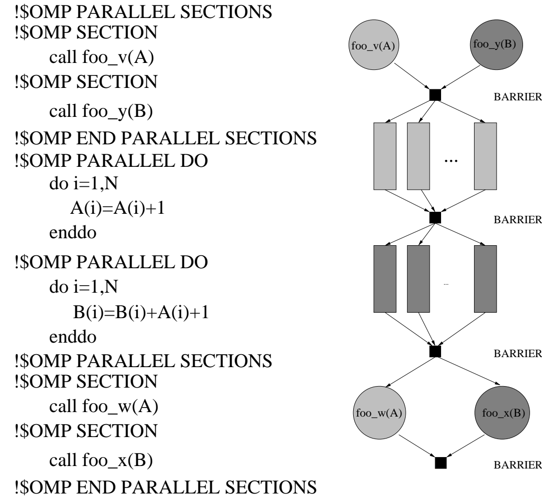

Figure 14: OpenMP parallelization for Example 7.

Figure 15 shows the parallel code using the extensions proposed in this paper. Two sections are defined, one for each execution path. The loops inside each section can also be parallelized with a `PARALLEL DO` construct. In order to specify the precedences between the loop chunks, the loops are annotated with the `NAME` clause. `PRED` and `SUCC` clauses are finally used to express the precedences between chunks. Notice that in this case it is not necessary to replicate the name given to the parallel loops because its definition is not included in an outer parallel loop.

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
        call foo_v(A)
!$OMP PARALLEL DO NAME(a) SUCC(b,i)
        do i=1,N
           A(i)=A(i)+1
        enddo
        call foo_w(A)
!$OMP SECTION
!$OMP PARALLEL DO NAME(b) PRED(a,i)
        call foo_y(B)
        do i=1,N
           B(i)=B(i)+A(i)+1
        enddo
        call foo_x(B)
!$OMP PARALLEL SECTIONS
```

Figure 15: Extended OpenMP parallelization for Example 7.

Example 8

Consider the source code and its correspondent task graph in figure 16. The outer loop is completely parallel and some parallelism is available between each loop iteration and the call to routine `foo1`. The different calls to routines `foo2` and `foo3` performed during the loop execution and the computations in the `foo1` routine can be overlapped. The calls to

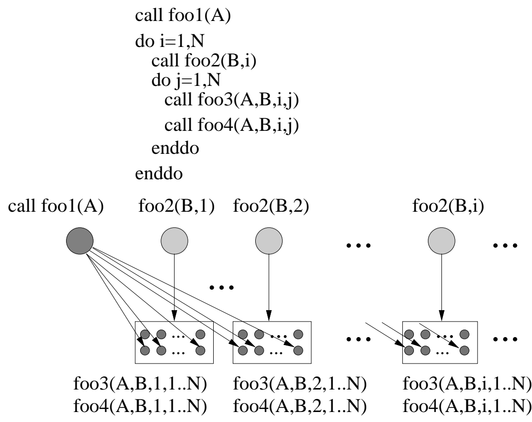

Figure 16: Source code and task graph for Example 8.

routine `foo4` can not be executed before the computations in the `foo1` routine have terminated.

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP NAME(sect_foo1) SUCC(loop_foo3:ALL,ALL)
    call foo1(A)
!$OMP SECTION
!$OMP PARALLEL DO
   do i=1,N
      call foo2(B,i)
!$OMP PARALLEL DO NAME(loop_foo3:i)
      do j=1,N
         call foo3(A,B,i,j)
!$OMP PRED(sect_foo1)
         call foo4(A,B,i,j)
      enddo
   enddo
!$OMP END SECTIONS
!$OMP END PARALLEL
```

Figure 17: Extended OpenMP parallelization for Example 8.

The current OpenMP definition would only allow the exploitation of the loop parallelism. The parallelism between the calls to routines `foo1`, `foo2` and `foo3` could not be exploited. Figure 17 shows the source code with the new directives and clauses to express all the amount of parallelism in the original task graph. Notice the use of the `ALL` clause to name all the tasks coming out from the nest of the `PARALLEL DO` directives. Notice the use of the `ALL` clause as an argument of the `SUCC` clause in the definition of the named section `sect_foo1`. This is necessary because the precedence that is releasing is defined for all the chunks of each of the different instances of the inner loop.

4 Putting it all together: blocked LU

In this section, we present a final example in which the proposals presented in this paper are used. The program computes a blocked LU decomposition [3] on a two dimensional matrix structured in a number of blocks. Different parallelization strategies are analyzed.

The original source code of the application is shown in Figure 18. Four different procedures are executed on each block of the matrix: `lu0`, `fwd`, `bdiv` and `bmod`.

```
do kk=1,NB
   call lu0(A,kk)
   do jj=kk+1,NB
      call fwd(A,kk,jj)
   enddo
   do ii=kk+1,NB
      call bdiv (A,ii,kk)
      do jj=kk+1,NB
         call bmod(A,ii,jj,kk)
      enddo
   enddo
enddo
```

Figure 18: Source code for blocked LU computation.

Depending on the value of the loop induction variable `kk` of the outer loop, a call to one of the mentioned routines is made for each block in the matrix. Figure 19 shows the computation for each iteration with an input matrix composed of 16 blocks. In iteration `kk`, the block `(kk,kk)` is processed by `lu0`. Procedure `fwd` works on all the blocks `(kk,kk+1...N)`, being `N` the number of blocks in one row of the matrix. Procedure `bdiv` works on all the blocks `(kk+1...N,kk)`. Finally, procedure `bmod` works on the blocks `(kk+1...N,kk+1...N)`.

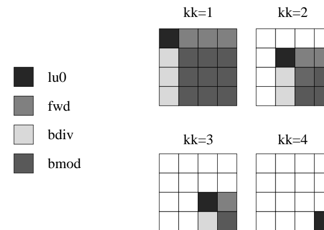

Figure 19: Execution of four iterations of the LU program.

Figure 20 shows the task graph for one iteration of the outer `kk` loop. The computation in routine `lu0` always precedes any other computation. After

lu0, all the bdiv and fwd computations can be executed in parallel. This means that both loops implementing these computations are completely parallel. Finally, routine bmod has to wait for the completion of the fwd computation working on the block in the same column where the bmod routine is invoked.

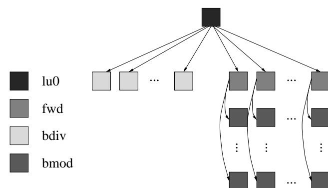

Figure 20: Precedence relations in one iteration of the LU computation.

```
    do kk=1,NB
!$OMP PARALLEL SECTIONS
!$OMP NAME(lu0) SUCC(fwd,bdiv+bmod)   (2) (1)
       call lu0(A,kk)
!$OMP NAME(fwd) PRED(lu0)   (2)
       do jj=kk+1,NB
          call fwd(A,kk,jj)
!$OMP SUCC(bdiv+bmod)   (3)
       enddo
!$OMP NAME(bdiv+bmod) PRED(lu0)   (1)
       do ii=kk+1,NB
          call bdiv (A,ii,kk)
          do jj=kk+1,NB
!$OMP PRED(fwd) IF(ii.eq.kk+1)   (3)
             call bmod(A,ii,jj,kk)
          enddo
       enddo
!$OMP END PARALLEL SECTIONS
    enddo
```

Figure 21: Extended OpenMP code for the `sections` version of LU.

The application can also exploit some parallelism across successive iterations of the outer `kk` loop. Once the `bmod` computation on a block `(k+1,k+1)` in the `k` iteration is finished, the `lu0` computation in iteration `k+1` can start. Once the `bmod` computations over blocks `(k+1...N,k+1)` in iteration `k` finish, the `bdiv` computation over the same blocks in iteration `k+1` can also start. The same for the `bmod` computations on blocks `(k+1,k+1...N)` in iteration `k` and the `fwd` computations on the same blocks in iteration `kk+1`.

In this section we describe four parallelization strategies for the code. The first two will focus on the exploitation of the parallelism inside one iteration of the outer loop. The additional parallelism between iterations of the outer loop is exploited in the last two versions. These four versions are called `sections`, `sections+do`, `do+sections`, and `do+sections+do`.

4.1 `sections` version

This version defines a parallel region inside the body of the outer loop. Three named sections are defined by the use of the `NAME` directive inside a `PARALLEL SECTIONS` construct. Figure 21 shows the extended OpenMP code for this version and Figure 22 the associated task graph. The named section `lu0` is defined with two successors: the named sections `bdiv+bmod` and `fwd`. These precedences appear in the task graph annotated with the `(1)` and `(2)` labels, respectively. Finally, the section `fwd` is releasing a precedence defined with section `bdiv+bmod`. This corresponds to precedence relation between the computations performed in the `fwd` stage and the computations performed in the `bmod` stage. These precedences appear in the task graph as the precedence edges labeled with `(3)`.

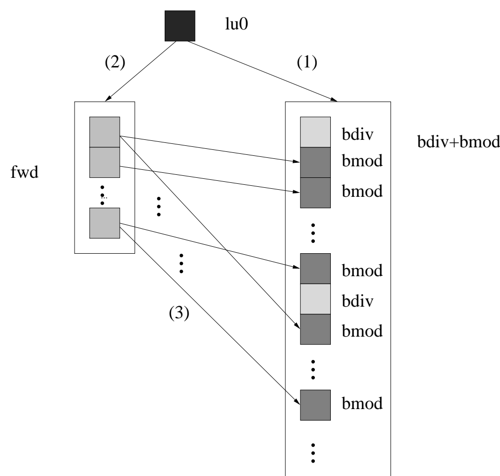

Figure 22: Task graph for the `sections` version of LU.

4.2 `sections+do` version

Compared to the previous version, this version also exploits the loop–level parallelism inside each named section. Figure 23 shows the extended OpenMP annotations for this version. The same precedences in the `sections` version are expressed between the named sections. The precedences that were nested to the loops inside the sections, are now expressed in terms of the chunks of iterations in the parallelized loops. The parallel loops have been named in order to express the appropriate precedences. Notice that the `ALL` construct is used in the definition of the precedences between the `fwd` and the `bmod` computations performed in the same `k` iteration. This is necessary due to the fact that the name definition for the `bmod_loop` identifier is nested to a parallelized loop, causing many instances of the same parallelized loop. As each chunk of the `fwd_loop` loop is releasing a precedence to one chunk of a concrete instance of the `bmod_loop` loop,

```
      do kk=1,NB
!$OMP PARALLEL SECTIONS
!$OMP NAME(lu0) SUCC(fwd,bdiv+bmod)   (1) (2)
         call lu0(A,kk)
!$OMP NAME(fwd) PRED(lu0)   (2)
!$OMP PARALLEL DO NAME (fwd_loop)
         do jj=kk+1,NB
            call fwd(A,kk,jj)
!$OMP SUCC(bmod_loop:ALL,jj)   (3)
         enddo
!$OMP NAME(bdiv+bmod) PRED(lu0)   (1)
!$OMP PARALLEL DO
         do ii=kk+1,NB
            call bdiv (A,ii,kk)
!$OMP PARALLEL DO NAME(bmod_loop:ii)
            do jj=kk+1,NB
!$OMP PRED(fwd_loop,jj)   (3)
               call bmod(A,ii,jj,kk)
            enddo
         enddo
!$OMP END PARALLEL SECTIONS
      enddo
```

Figure 23: Extended OpenMP code for the `sections+do` version of LU.

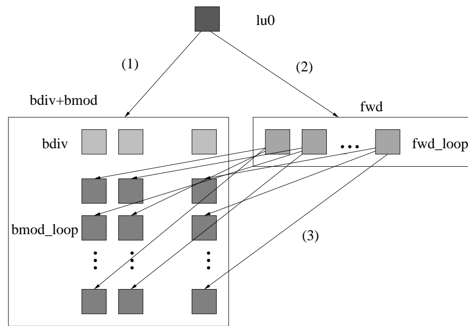

Figure 24: Task graph for the `sections+do` version of LU.

the use of the `ALL` clause allows the specification of such synchronizations. Figure 24 shows the corresponding task graph for this parallelization. The precedence edges corresponding to the precedences between the named sections are labeled with numbers `(1)` and `(2)`. The precedences between the loop chunks are labeled with the number `(3)`.

4.3 `do+sections` version

This version exploits the same amount of parallelism than the `sections` version inside each iteration of the outer loop. In addition, the outer loop is now parallelized using a `PARALLEL DO` construct and inserting new precedence clauses to ensure that the precedences between the computations in successive iterations of the outer loop are guaranteed, as shown in Figure 25. Notice that the names defined in each named section have been replicated using the induction variable of the outer loop. Precedences between work–sharing constructs

```
!$OMP PARALLEL DO
      do kk=1,NB
!$OMP PARALLEL SECTIONS
!$OMP NAME(lu0:kk) SUCC((fwd:kk),(bdiv+bmod:kk))   (1) (2)
!$OMP PRED(bdiv+bmod:kk-1)   (7)
         call lu0(A,kk)
!$OMP NAME(fwd,kk) PRED(lu0:kk)   (2)
         do jj=kk+1,NB
!$OMP PRED(bdiv+bmod:kk-1) IF(kk.gt.1)   (5)
            call fwd(A,kk,jj)
!$OMP SUCC(bdiv+bmod:kk)   (3)
          enddo
!$OMP NAME(bdiv+bmod:kk) PRED(lu0:kk)   (1)
         do ii=kk+1,NB
!$OMP PRED(bdiv+bmod:kk-1) IF(kk.gt.1)   (4)
            call bdiv (A,ii,kk)
            do jj=kk+1,NB
!$OMP PRED(fwd:kk) IF(ii.EQ.kk+1)   (3)
!$OMP PRED(bdiv+bmod:kk-1) IF(kk.gt.1)   (6)
               call bmod(A,ii,jj,kk)
!$OMP SUCC(fwd:kk+1) IF(ii.eq.kk+1 .and. kk.lt.NB)   (5)
!$OMP SUCC(bdiv+bmod:kk+1) IF(jj.eq.kk+1 .and. kk.lt.NB)   (4)
!$OMP SUCC(bdiv+bmod:kk+1) IF(ii.gt.kk+1 .and. jj.gt.kk+1 .and. kk.lt.NB)   (6)
!$OMP SUCC(lu0:kk+1) IF(kk.lt.NB) (7)
            enddo
         enddo
!$OMP END PARALLEL SECTIONS
      enddo
```

Figure 25: Extended OpenMP code for the `do+sections` version of LU.

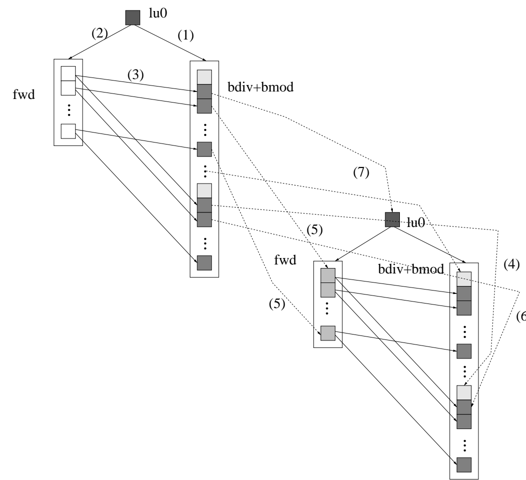

Figure 26: Task graph for the `do+sections` version of LU.

in the same iteration are represented with solid lines in the task graph shown in Figure 26. Precedences between work–sharing constructs in different iterations are represented with dashed lines. The first group of precedences are labeled with numbers `(1)`, `(2)`, `(3)` and the second group of precedences are labeled with numbers `(4)`,`(5)`, `(6)` and `(7)`.

4.4 `do+sections+do` version

Finally, this version exploits all the available parallelism in the application: inside each iteration of the outer loop and across successive iterations. The parallelism inside each iteration is specified with the use of the `PARALLEL SECTIONS` and the `PARALLEL DO` constructs. The precedences between `lu0`, `fwd`, `bdiv` and `bmod` are expressed with the definition of three named sections `lu0`, `fwd` and `bdiv+bmod`.

```
!$OMP PARALLEL DO
      do kk=1,NB
!$OMP PARALLEL SECTIONS
!$OMP NAME(lu0:kk) SUCC((fwd:kk),(bdiv+bmod:kk))   (2) (1)
!$OMP PRED((bmod_loop:kk-1:kk),kk) IF(kk.gt.1)   (7)
         call lu0(A,kk)
!$OMP NAME(fwd:kk) PRED(lu0:kk)   (2)
!$OMP PARALLEL DO NAME(fwd_loop:kk)
         do jj=kk+1,NB
!$OMP PRED((bmod_loop:kk-1:kk),jj) IF(kk.gt.1)   (5)
            call fwd(A,kk,jj)
!$OMP SUCC((bmod_loop:kk:kk),jj)   (3)
         enddo
!$OMP NAME(bdiv+bmod:kk) PRED(lu0:kk)   (1)
!$OMP PARALLEL DO NAME(bdiv+bmod_loop:kk)
         do ii=kk+1,NB
!$OMP PRED((bmod_loop:kk-1:kk),ii) IF(kk.gt.1)   (4)
            call bdiv (A,ii,kk)
!$OMP PARALLEL DO NAME(bmod_loop:kk:ii)
            do jj=kk+1,NB
!$OMP PRED((fwd_loop:kk),jj) IF(jj.eq.kk+1)(3)
!$OMP PRED((bmod_loop:kk-1),jj) IF(kk.gt.1)   (6)
               call bmod(A,ii,jj,kk)
!$OMP SUCC((fwd_loop:kk+1),jj) IF(ii.eq.kk+1 .and. kk.lt.NB)   (5)
!$OMP SUCC((bmod_loop:kk+1:ii),jj) IF(jj.gt.kk+1 .and. ii.gt.kk+1 .and. kk.lt.NB)   (4)
!$OMP SUCC(bdiv+bmod:kk+1) IF(jj.eq.kk+1 .and. kk.lt.NB)   (6)
!$OMP SUCC(lu0:kk+1) IF(jj.eq.kk+1 .and. ii.eq.kk+1 .and. kk.lt.NB)   (7)
            enddo
         enddo
!$OMP END PARALLEL SECTIONS
      enddo
```

Figure 27: Extended OpenMP code for the `do+sections+do` version of LU.

Figure 28: Task graph for the `do+sections+do` version of LU.

Inside sections `fwd` and `bdiv+bmod`, the loop–level parallelism is exploited. These loops have been also named because their chunks of work are affected by precedences. The parallelism between consecutive iterations of the outer loop is specified with the `PARALLEL DO` construct in the outer `do` statement. Figure 27 shows the extended OpenMP annotations for this version.

Notice that the `PRED` and `SUCC` clauses refer to named work–sharing constructs where their names have been replicated using the loop induction variable of the outer loop, as in the previous version. Figure 28 shows the task graph associated to this parallelization. Solid lines correspond to precedences between work–sharing constructs in the same iteration. They are labeled as `(1)`, `(2)` and `(3)`. Dashed lines correspond to precedences between work–sharings in different iterations, labeled as `(4)`, `(5)`, `(6)` and `(7)`.

5 Conclusions

In this paper, we have presented a set of new directives and clauses for the OpenMP programming model towards the definition of precedence relations between the work–sharing constructs in a parallel region. Although our proposal is not closed, many interesting examples have been described in which complex task graphs and pipelined structures appear. The current specification of OpenMP does not provide direct support to this kind of parallelization strategies. The programmer has to define its own synchronization structures and statements in order to exploit this parallelism. The extensions are applied to a blocked `LU` decomposition kernel. The usefulness of the proposed extensions is shown with this example. Four different parallel strategies are presented for the `LU` code, each one exploiting different portions of all the available parallelism.

The full proposal is currently being implemented on top of the OpenMP NanosCompiler [1].

Acknowledgments

This research has been supported by the Ministry of Education of Spain under contracts TIC98-511 and TIC97-1445CE, the ESPRIT project 21907 NANOS and the CEPBA (European Center for Parallelism of Barcelona).

References

[1] E. Ayguadé, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro and J. Oliver. NanosCompiler: A Research Platform for OpenMP Extensions. In *First European Workshop on OpenMP*, Lund (Sweden), October 1999

[2] 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.

[3] G. von Laszewski, M. Parashar, A. Gaber Mohamed and G.C. Fox. On the Parallelization of Blocked LU Factorization Algorithms on Distributed Memory Architectures. In Proc. of the *1992 Conference on Supercomputing*, Washington (USA), July 1992.

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

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