1 Nested Parallelism and Pipelining in OpenMP

M. GONZALEZ, E. AYGUIADE, X. MARTORELL, J. LABARTA

Computer Architecture Department, Technical University of Catalonia
cr. Jordi Girona 1-3, Modul D6, 08034 - Barcelona, Spain
{marc, eduard, xavim, jesus}@ac.upc.es

This chapter describes two proposals for the OpenMP programming model arising from the detected lacks in two topics: nested parallelism and pipelined computations. The contributions in this chapter point out the situations where nested parallelism appears as the way for increasing performance and the heuristics for an efficient parallel execution under a multi-level strategy. Some applications, although containing some amount of parallelism, this one can not be expressed through OpenMP directives due to data dependences. Usually the available parallelism is organized in the form of a pipeline, requiring complex thread synchronizations not available in the current OpenMP definition. The chapter presents a proposal introducing the precedence relations in the OpenMP programming model. The proposal allows the programmers to specify explicit point-to-point thread synchronizations, being general enough to manage with any pipelined computation.

1.1 INTRODUCTION

OpenMP [7] has emerged as the standard programming model for shared-memory parallel programming. The OpenMP organization has joined the main parallel vendors in an effort of standardizing parallel directives. The resulting definition for the OpenMP programming model contains the main features included in the parallel environments of each main vendor. It follows an SPMD programming model, where parallel tasks are specified by work-sharing constructs that define the work distribution among the available threads. Current OpenMP definition offers support for loop and sections parallelism, plus some thread synchronization schemes, like barrier synchronizations, mutual exclusions or atomic operations. Its simplicity and portability across a range of parallel platforms are achieved without sacrificing the performance.
The task of parallelizing a sequential code is not an easy work to do. Program-
ners have to spend significant efforts in the detection of the available parallelism.
Much harder is the work dedicated to the tuning of the application. In many cases,
programmers see how the available parallelism is restricted because the parallel en-
vironment does not offer enough support for its efficient exploitation, or even for
its exploitation itself. Since the OpenMP birth, there have been several research
works around the most common topics in any parallel programming model. A lot
of previous ideas in the literature on HPF and MPI have been now developed for
OpenMP. Typically, OpenMP research works present the evaluation of the current
programming model through the parallelization of several real applications that point
out the benefits and lacks of the model. The evaluation of the model is done by mea-
suring the impact of several factors like the data placement, the work distribution,
the thread synchronizations and the overheads of the parallel runtime system. All
of these measurements are always faced to the available functionalities in the model
that allow the programmers to tune their applications to get the best performance. In
that context, most of the OpenMP research works end up with the definition of new
constructs in the language to cover the detected lacks.

One of the most significant OpenMP functionalities is the possibility of express-
ing multiple levels of parallelism [2, 4, 6, 8, 9, 17]. This is achieved by nesting
parallel constructs, which includes parallel loops and sections. The possibility of
nesting parallel constructs is the natural way to express, using a single programming
paradigm, the integration of task and data parallelism or even the exploitation of mul-
tiple levels of data parallelism. Exploiting just a single level of parallelism may incur
in low performance returns when the number of processors to run the application is
increased [2]. The solution to that is the exploitation of the multi-level parallelism
in the application. Typically, scientific applications contain multilevel parallelism.
This parallelism is organized in several computations operating over independent
data structures, where the work to be done can be performed in parallel too (task and
data parallelism). In some situations, the computations present some dependences
that restrict the available parallelism, but not to the point of eliminating the possibility
of having a multilevel strategy.

It has been very recently, that OpenMP commercial systems (compiler and run-
time) support the nested parallelism exploitation [13, 14]. Because of that, one
can find several research works that combine two programming models in order to
achieve a multi-level execution. Typically, MPI is used in the outermost level, and
then OpenMP is used in the innermost level. Examples of this kind of works are
[15, 16]. From our point of view, these works just demonstrate the usefulness of
the nested parallelism. We are going to show that there is no need of mixing two
programming models, OpenMP is enough but with several modifications.

This chapter summarizes our research activity in the nested parallelism exploitation
and thread synchronizations. We have carried out several experiments that have
helped us to point out other issues related to the nested parallelism exploitation
[2]. Work allocation has appeared to be an important issue to achieve performance.
Current work allocation schemes in OpenMP seem to be not enough to efficiently
exploit the parallelism in the innermost level. The lacks in the model lead to a loss
of performance due to several factors. Applications might suffer from a poor locality exploitation caused by a wrong work placement. Applications might not be suitable for the grain size obtained in a multilevel parallel strategy. The resulting grain size might be too small to make worth the parallelism exploitation with large number of threads. Our experience has showed that it is necessary to include in the model new mechanisms to avoid the default work allocation. The programmer has to be able to define the appropriate work distribution schemes that fit well in the application.

We have detected that when looking for nested parallelism, data dependences might appear through the different levels of parallelism. Although the data dependences, some parallelism is still available, but can not be specified through the existing directives in the programming model. Typically, this situations appear in codes containing irregular access patterns or an available parallelism in the form of a pipeline [3, 5, 10, 12]. The parallel computations are characterized by a data dependent flow of computation that implies serialization. In this direction, the specification of generic task graphs as well as complex pipelined structures is not an easy task in the framework of OpenMP. In order to exploit the available parallelism, the programmer has to define complex synchronization data structures and use synchronization primitives along the program, sacrificing readability and maintainability.

1.2 OPENMP EXTENSIONS FOR NESTED PARALLELISM

OpenMP extends the Fortran sequential programming model with single-program multiple data (SPMD) constructs, work-sharing constructs and synchronization constructs, and provides support for the sharing and privatization of data. The parallelism is specified through the definition of a parallel region. A parallel region is a portion of code that is going to be executed by several threads. The code in the parallel region is somehow replicated and executed by the threads, but several mechanisms allow different work distribution schemes. The OpenMP execution model follows the fork/join model. A program written with OpenMP begins executing as a single process, which is called the master thread. The program executes sequentially until a parallel region is encountered. The master thread creates a team of threads and becomes the master of the team. The statements in the parallel region are executed in parallel by all threads. Upon completion of the parallel region, the threads in the team synchronize and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution.

1.2.1 Parallelism Definition

OpenMP provides two directives for the definition of the parallelism: PARALLEL and END PARALLEL. The programmer defines the parallel region by enclosing the parallel code between these two directives. The syntax of the construct is:

!$OMP PARALLEL [ clauses ]
c Parallel code
!$OMP END PARALLEL

When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The master thread belongs to the team. The number of threads in the team is controlled by environment variables, the NUM_THREADS clause, and/or library calls. Only the use of the NUM_THREADS clause is going to be described. Following the current OpenMP definition the behavior of the program is implementation-dependent when more threads are requested than can be successfully created. We will see in section 1.2.2 that our proposal supports this situation and does not introduce any restriction on the number of threads to be used in a parallel region. Once created, the number of threads in the team remains constant for the duration of that parallel region. It can be changed either explicitly by the user or automatically by the runtime system from one parallel region to another.

Within the dynamic extent of a parallel region, threads are numbered consecutively ranging from zero (for the master thread) up to one less than the number of threads within the team. The intrinsic omp_get_num_threads returns the number of executing threads in the team. It is possible to obtain the value of a thread identifier by invoking the library routine omp_get_thread_num. Between two consecutive parallel regions, the thread numbers for the two regions are consistent. This means that a thread identified with a given thread number in the first parallel region will be identified with the same thread number in the second region. This allows the programmer to maintain data locality for the two parallel regions. The PARALLEL construct accepts several clauses. The most common are PRIVATE, FIRSTPRIVATE, and REDUCTION.

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 which specifies the enclosed sections of code (each one delimited by a SECTION directive) are 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. Finally, the SINGLE work-sharing construct informs that only one thread in the team is going to execute the work.

The END PARALLEL directive denotes the end of the parallel region. There is an implied barrier at this point, so that only when all threads in the team have reached the END PARALLEL directive, the master thread continues the execution past the end of the parallel region.

1.2.1.1 Nested Parallelism OpenMP considers nested parallelism by allowing the programmer to nest PARALLEL constructs. The nested parallelism appears because a thread already executing inside an outer parallel region, might encounter
a PARALLEL construct, defining an inner parallel region. If a thread in a team
executing a parallel region encounters another parallel region, it creates a new team,
and it becomes the master of that new team. The thread name space is defined in the
same manner independently of the level where a parallel region is spawned: threads
get numbered from zero to the number of available threads minus one. The work
allocation schemes associated to the work-sharing constructs are equally defined,
whether or not they appear in a nested parallel construct.

1.2.2 Thread Groups

In the fork/join execution model defined by OpenMP, 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).

In this study, 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 group.
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 the 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 and the allocation strategy or scenario for the inner levels of parallelism
that might be spawned. When a member of this new team encounters another parallel
construct (nested to the one that caused the group definition), it creates a new team
and deploys its parallelism to the threads that compose its group. The GROUPS
clause allows the user to specify 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 [2]. In this paper we
only comment on the two more relevant. For additional details concerning alternative
formats as well as implementation issues, please refer to this publication.

Weighted Group Definition In this case, the user specifies the number of groups
(ngroups) and an integer vector (weight) indicating the relative amount of computa-
tion that each group has to perform. The syntax is:

GROUPS(ngroups,weight)

Vector weight is allocated by the user in the application address space and it has
to be computed from information available within the application itself (for instance
iv NESTED PARALLELISM AND PIPELINING IN OPENMP

howmany(1:ngrroups) = 1
do while (sum(howmany(1:ngrroups)) .lt. nthreads)
   pos = maxloc(weight(1:ngrroups)/howmany(1:ngrroups))
   howmany(pos(1)) = howmany(pos(1)) + 1
end do
masters(1) = 0
do i = 1, ngrroups-1
   masters(i+1) = masters(i) + howmany(i)
end do

Fig. 1.1 Skeleton of the algorithm used to compute the composition of groups.

iteration space, computational complexity or even information collected at runtime). The runtime library determines, from this information, the composition of the groups. The algorithm assigns all the available threads to the groups and ensures that each group at least receives one thread. The main body of the algorithm is shown in figure 1.1 (using Fortran90 syntax). The library generates two internal vectors (masters and howmany). In this algorithm, nthreads is the number of threads that are available to spawn the parallelism in the parallel construct containing the group definition.

Most General Group Definition This syntax corresponds to the most general format for the groups definition. Its syntax is

GROUPS(ngrroups, masters, howmany);

The first argument (ngrroups) 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. Notice that this format must be used when the default mapping explained before does not provide the expected performance.

1.2.3 Evaluation of the proposal

In this section we present some evaluation of the proposal for the thread groups. We want to show that exploiting nested parallelism is worthy but has to be done through certain scheduling decisions, achieved by the thread grouping. The presented results also show that there is no need for mixing OpenMP programming with any other programming paradigm. By including the new constructs in the OpenMP language, it is possible to avoid hybrid models for the nested parallelism exploitation. We think this is quite important, faced to the results appeared in recent research works around OpenMP nested parallelism.

1.2.3.1 MGPO应用 MGPO is a standard Fortran77 multi-block grid code. A parallel version of MGPO uses MPI asynchronous sends and receives
SUBROUTINE barosp(drhox, drhoy, drx2d, dry2d, nb)

C$OMP PARALLEL DO PRIVATE (i, j, k)
   DO 200 j = 1, nb
   DO 200 k = 1, nb
       rho (i, 1, k, nb) = rho (i, 1, k, nb) - rmean (i, 1, k, nb)
   200 CONTINUE
C$OMP END

PROGRAM main
...

C$OMP PARALLEL DO PRIVATE (m) GROUPS (maxb, work)
   DO 414 m = 1, maxb
       IF (node .ne. 2) THEN
           CALL barosp (drhox, drhoy, drx2d, dry2d, m)
       ENDIF
   414 CONTINUE
END

Fig. 1.2 Excerpt of the OpenMP implementation.

to exchange data between adjacent blocks at the interfaces. OpenMP has been
used as a second level of parallelization within each MPI process to improve the
load balance in the simulation of the Arabian Gulf. This area is extended from 48°
East to 58° East in longitude and from 23.5° North to 30.5° North in latitude. The
computation is defined over a grid structure formed by 20 blocks, covering the gulf
area. OpenMP with NanosCompiler [1] extensions is used to parallelize the serial
version of the MGPOm code at the outer levels (block to block) as well as at the
inner levels (within each block). The number of threads used to exploit the inner level
of parallelism depends on the size of each grid block. Figure 1.2 shows the use of
OpenMP directives and GROUPS construct implemented into the main program and
a subroutine of the serial MGPOm code version. This figure shows a version in which
the runtime library, using the default algorithm described in section 1.2.2, determines
the composition of the groups. The GROUPS clause has as input arguments the
number of blocks (maxb) and a vector with the number of grid points in each block
(work). Notice that the GROUPS clause is the only non-standard use of OpenMP.
The exploitation of multiple levels of parallelism is achieved through the nesting of
PARALLEL DO constructs. Table 1.1 shows the number of grid points in each block
for the 20-block grid. This is the composition of the work vector. The runtime library
supporting the code generated by the NanosCompiler would generate the allocation
of threads to groups shown in tables 1.2 and 1.3 assuming 20 and 30 processors,
respectively. Who and Howmany are the two internal vectors generated by the library
with the master of each group (who) and the number of threads to be used in each
group (howmany). With 20 processors, the critical path is determined by the largest
block (i.e. block number 8), yields a theoretical speedup of 12.5. With 30 processors,
the critical path is determined by the block with the largest ratio size/howmany (i.e. block number 17), with a theoretical speed-up of 19.4. According to the allocation of

<table>
<thead>
<tr>
<th>Block</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Howmany</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Who</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Howmany</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Who</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 1.2 Default allocation of threads to groups with 20 threads.

<table>
<thead>
<tr>
<th>Block</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Howmany</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Who</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Howmany</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Who</td>
<td>15</td>
<td>16</td>
<td>18</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>78</td>
</tr>
</tbody>
</table>

Table 1.3 Default allocation of threads to groups with 30 threads.
OPENMP EXTENSIONS FOR NESTED PARALLELISM

```
PROGRAM main
  CALL compute_groups(work, maxb, who, howmany)
  ...
C$OMP PARALLEL DO PRIVATE(n) GROUPS(maxb, who, howmany)
  DO 414 n = 1, maxb
       ...
       IF (node .ne. 2) THEN
          CALL baropg (drbox, drhoq, drx2d, dry2d, n,
          ...
       ENDIF
414 CONTINUE
  ...
END
```

Fig. 1.3 Excerpt of the OpenMP implementation with user-defined groups

of the threads to groups as well as the identities of blocks in a cluster. A cluster of
two blocks with 20 processors is shown in this case. In this case, the work imbalance
is noticeably reduced.

<table>
<thead>
<tr>
<th>Cluster</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks</td>
<td>1.9</td>
<td>2.15</td>
<td>3.12</td>
<td>4.11</td>
<td>5.8</td>
<td>6.20</td>
<td>7.19</td>
<td>10.1</td>
<td>14.15</td>
<td>16.18</td>
</tr>
<tr>
<td>Howmany</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Who</td>
<td>0</td>
<td>7</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>17</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>

Table 1.4 Allocation of threads to groups with 20 threads after clustering.

1.2.3.2 Performance In this section we evaluate the behavior of two parallel
versions of MGPOM. The MPI-OpenMP version exploits two levels of parallelism
by combining the two programming paradigms: MPI to exploit the inter-block
parallelism and OpenMP to exploit the intra-block parallelism. The OpenMP-Only
version exploits the two levels of parallelism using OpenMP and the extensions
offered by the NanosCompiler and supporting OpenMP runtime system. For the
compilation of the multilevel OpenMP version 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 an SGI
Origin2000 system [11]. The flags are set to -mips4 -O2. The experiments have
been performed on system with 64 R10k processors, running at 250 MHz with 4 Mb of
secondary cache each. Table 1.5 shows the speed-up achieved by the two versions of
the program. Notice that the performance of the nested OpenMP version is equivalent
to the original mixed MPI-OpenMP. In order to conclude this section, table 1.6 shows
the relative performance (w.r.t the execution with 20 processors of the OpenMP-
only version) for three OpenMP versions of the application: no_groups exploits t\w
two levels of parallelism but does not define groups (similar to using the NUM\_THRE\_ADS
clause), groups performs a homogeneous distribution of the available threads among
1.3 OPENMP EXTENSIONS FOR THREAD SYNCHRONIZATIONS

In this section we describe the OpenMP extensions proposed to support the specification of explicit point-to-point thread synchronizations. Although the proposal is general enough to cover any kind of synchronization scheme, the extensions are in the framework of nested parallelism and target the definition pipelined executions. The proposal is based on the introduction of the precedence relations in the OpenMP programming model. An initial definition of the extensions to specify precedences was described in [3].

1.3.1 Precedence Relations

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.
1.3.1.1 The NAME clause  The NAME clause is used to provide a name to a task that comes out of a work-sharing construct. Here follows its syntax of use for the OpenMP work-sharing constructs:

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

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 SECTIONS construct, the NAME clause is used to identify each SECTION. In a SINGLE construct the NAME clause is used in the same manner. In a DO work-sharing construct, the NAME clause only provides a name to the whole loop. We propose to define each iteration of the loop as a parallel task. This means that the name space for a parallel loop has to be large enough to identify each loop iteration. This is done by identifying each iteration of the parallelized loop with the identifier supplied in the NAME clause plus the value of the loop induction variable for that iteration. Notice that the number of tasks associated to a DO work-sharing construct is not determined until the associated do statement is going to be executed. This is because the number of loop iterations is not known until the loop is executed. Depending on the loop scheduling, the parallel tasks (iterations) are mapped to the threads. The programmer simply defines the precedences at the iteration level. These precedences are translated at runtime to task precedences that will cause the appropriate thread synchronizations, depending on the SCHEDULE strategy specified to distribute iterations.

1.3.1.2 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.

```
[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 task names that must release a precedence to allow the thread encountering the PRED directive to continue its execution. The SUCC directive is used to define all those tasks that, at this point, may continue their execution. The IF clause is used to guard the execution of the synchronization action. Expression exp is evaluated at runtime to determine if the associated PRED or SUCC directive 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 enclosing 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 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 presents two different formats:

\[
\text{task_id} = \text{name}_\text{ident} | \text{name}_\text{ident}, \text{expr}
\]

When the task_id is only composed of a name_ident identifier, the parallel task corresponds to a task coming out from a SECTIONS or SINGLE work-sharing construct. In this case, the name_ident corresponds to an identifier supplied in a NAME clause that annotates a SECTION/SINGLE construct. When the name_ident is followed by one expression, the parallel task corresponds to an iteration coming from a parallelized loop. The expression evaluation must result in an integer value identifying a specific iteration of the loop. The precedence relation is defined between the task being executed and the parallel task (iteration) coming out from the parallelized loop with the name supplied in the precedence directive. Notice that once the precedence has been defined, the synchronization that is going to ensure it will take place between the threads executing the two parallel tasks involved in the precedence relation. Therefore, implicit to the precedence definition, there is a translation of task identifiers to the threads executing the tasks, depending on the scheduling that maps tasks to threads.

In order to handle nested parallelism, we extend the previous proposal. When the definition of precedences appear in the dynamic extend of a nested parallel region caused by an outer PARALLEL directives, multiple instances of the same name definition (given by a NAME clause/directive) exist. In order to differentiate them, the name_ident needs to be extended with as many task_id as outer levels of parallelism.

\[
\text{name} \text{ident}[:\text{task id}] +
\]

Therefore, the task_id construct might take the following syntax:

\[
\text{task_id} = \text{name}_\text{ident} | \text{name}_\text{ident}, \text{expr} | [(\text{task_id}):]*\text{task_id}
\]

Figure 1.4 shows a multi-level example. Two nested loops have been parallelized although they are not completely parallel. Some parallelism might be exploited according to the data dependencies caused by the use of $A(k-1,j-1)$ in iteration $(k,j)$. Both parallel loops have been named and the appropriate precedences have been defined to ensure that data dependents are not violated. Notice that the task name space in the innermost loop (innerloop) is replicated for each iteration of the outermost loop (outerloop). To distinguish between different instances of the same name space, a task identifier is extended with the list of all task identifiers in the immediate upper levels of parallelism.
OPENMP EXTENSIONS FOR THREAD SYNCHRONIZATIONS

\begin{verbatim}
CETMP PARALLEL DO NAME (outerloop)
  do k = 1, N
CETMP PARALLEL DO NAME (innerloop)
  do j = 1, N
  CETMP PRED(outerloop,k-1):(inner_loop, j-1);
  \ldots
  A(k,j)=A(k-1,j-1)*A(k,j)
  \ldots
CETMP SUCC(outerloop,k+1):(inner_loop, j+1);
\end{verbatim}

\textbf{Fig. 1.4} Example of multi-level code with precedences.

1.3.2 Evaluation of the proposal

This section shows two parallelization strategies for one of the NAS benchmarks that make use of the proposed precedence directives. The objective is to point out the simplicity of the directives faced to the transformations that programmers usually do to adapt their codes to OpenMP programming style. As before, the evaluation has been done on a SGI Origin2000 with 64 R10000 processors (250 Mhz) and Irix 6.5. The parallel code is automatically generated using the NanosCompiler [1] to transform the source code annotated with the new precedence directives to run on NthLib [18, 17].

1.3.2.1 NAS LU LU is a simulated CFD application that comes with the NAS benchmarks [19]. It uses a symmetric successive over-relaxation (SSOR) method to solve a diagonal system resulting from a finite-difference discretization of the Navier-Stokes equations. Two parallel regions are defined for the solver computation. Both have the same structure in terms of data dependences, so only one will be described. The computation is performed over a three dimensional matrix, by the nest of three do loops, one per dimension. The matrix size is 31 * 31 * 31 elements. The computation defines that there is a dependence from the element \((k, j, i)\) to elements \((k+1, j, i)\), \((k, j+1, i)\) and \((k, j, i+1)\). We have evaluated three different versions of the LU benchmark for class W. Two versions using a single level parallel strategy, and a third version exploiting two levels of parallelism.

Single level omp This version corresponds to the one distributed in the NAS benchmarks. It exploits loop level parallelism in the outermost dimension \((k)\). As this loop is not completely parallel, the benchmark contains the necessary thread synchronizations to preserve the dependences in the \(k\) dimension. These synchronizations are coded by the programmer in the source code using vectors allocated in the application address space. Once a thread working on a \(k\) iteration has performed some iterations on the \(j\) loop, signals the thread working on \(k+1\) iteration for the same set of \(j\) iterations and allows its execution. Thus, a pipeline is created.

Figure 1.5 shows the structure of the source code for this version. Notice that the programmer has to introduce the FLUSH construct to ensure memory consistency
!iv  NESTED PARALLELISM AND PIPELINING IN OPENMP

```fortran
$OMP PARALLEL DEFAULT(SHARED) PRIVATE(k,iam)  subroutine bits(...)  $OMP MASTER  .  .  .  .  .  ian = omp_get_thread_num()  if (nthreadnum_gr (end-ist)) then  if (iam_gr 0 .and. iam.le.nthreadnum)  nthreadnum=end-jst  neigh=iam-1  endif  do while (isync(neigh).eq.0)  $OMP END MASTER  !$OMP FLUSH(isync)  !$OMP end do  ian = omp_get_thread_num()  isync(iam) = 0  !$OMP FLUSH(isync)  endt  !$OMP BARRIER  do k = 2, nz -1  !$OMP DO  isize1, isize2,  isize3.  ...  1  2  3  4  5  6  7  n, m, nz, k, rmd, tv, a, b, c, d, ist, iend, (st, jend, mx0, my0)  isize(iam) = 1  !$OMP FLUSH(isync)  !$OMP end do  endif  . .  .
```

Fig. 1.5  Source code for NAS LU application.

for the integer vector isync used for synchronization. The vector is not padded, so false sharing problems may appear in the synchronization execution degrading performance. The leftmost bar in figure 1.7 shows the performance numbers for this version in terms of speed-up. Notice that for this version only up to 31 processors might be used, as the k loop only contains 31 iterations.

**Single level with precedences**  This version follows a similar parallel strategy as the Single level omp version. To design this version, the extensions described in Section 1.3 have been introduced in the source code replacing the original synchronization code. False sharing problems disappear and the programmer has not to be aware about memory consistency issues as both things are handled by the runtime system. A blocking scheduling to the k,j,i do loops has been done and only the blocked k loop has been parallelized. The blocking allows the programmer to control the amount of work performed between two thread synchronizations. Figure 1.6 shows the new source code with precedence directives. The middle bar in figure 1.7 shows the performance numbers for this version. Notice that it behaves very similar to the Single level omp version, so no performance is lost due to possible runtime overheads. Both versions Single level omp and Single level nth are not exploiting all the available parallelism in the computation. After computing an element (k,j,i), the computation can continue on elements (k+1,j,i), (k,j+1,i) and (k,j,i+1) in parallel. Those versions only exploit the parallelism between the (k+1,j,i) and (k,j+1,i) elements.
Two levels with precedences  This version exploits near all the parallelism present in the computation. Figure 1.6 shows the new source code with precedence directives. In this version, once a thread ends its computation on a block (bk, bj, bi) composed by a set of k, j and i iterations, signals two threads: the ones that are going to work on the blocks (bk+1, bj, bi) and (bk, bj+1, bi). Notice that this version, as it is exploiting more parallelism, is able to take advantage of more than 31 processors, and even more than that, it is able to fill the pipeline faster than the Single level omp and Single level precedences versions. The performance numbers in rightmost bar in table 1.7 show that the Two levels precedences reaches the maximum speed-up with 49 threads, 20% more than the best performance in the Single level precedences versions.

1.4 SUMMARY

In this chapter we present two proposals for extending the OpenMP programming model. The first one is oriented toward one of the most studied OpenMP topics: the nested parallelism exploitation. Recent OpenMP research works showed that some OpenMP applications can not be speeded up while the number of available threads increases. This behavior has been seen when more than 32 threads are available and the parallel strategy is based on the exploitation of a single level of
parallelism. In front this situation, multi-level strategies appear as a good solution for increasing performance. It has been also showed that with the current OpenMP support, those applications can not benefit from nested parallelism. The reason for this is that OpenMP does not provide the appropriate mechanisms to control the thread distribution among the different levels of parallelism. Other recent works tried to solve the problem by mixing two programming paradigms: MPI for the outermost level and OpenMP in the inner levels. Those works showed the usefulness of the nested parallelism, but their solution needs the mixing of two programming models very different in their basis: MPI targets distributed memory architectures, while OpenMP was initially defined for shared memory architectures. The proposal in this chapter solves the OpenMP lacks in a pure OpenMP manner. The proposal introduces the thread grouping inside the OpenMP programming model. New constructs are defined that allow the programmers to explicitly set the number of threads to be used in each parallel level and a thread distribution for the execution of the nested parallel levels.

The second proposal in the chapter focuses on the OpenMP support for thread synchronizations. Current OpenMP definition covers simple synchronization schemes like the barrier synchronization, mutual exclusion execution and ordered execution. It is quite common that the applications contain some amount of parallelism but not in the form as what the OpenMP directives allow to specify. In such situations it is necessary to code explicit thread synchronizations that usually define a computational wavefront in the form of a pipeline. Current synchronization constructs seem not enough to deal with these situations, thus the proposal in the chapter tries to cover the OpenMP lacks. The proposal is based on the introduction of the precedence relations between the parallel tasks arising from the worksharing constructs. Programmers are allowed to define explicit point-to-point thread synchronizations, general enough to organize the computations under a pipelined structure.
Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain and the European Union (FEEER funds) under contract TIC2001-0995-C02-01, and by the ESPRIT POP project under contract IST-2001-33071.
References


16. P. Kloos, F. Mathey, P. Blaise ”OpenMP and MPI programming with a CG algorithm” Proceedings on the 3rd European Workshop on OpenMP (EWOMP2001)

