Exploiting Pipelined Executions in OpenMP

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
{marc, eduard, xavim, jesus}@ac.upc.es

Abstract

This paper proposes a set of extensions to the OpenMP programming model to express point–to–point synchronization schemes. This is accomplished by defining, in the form of directives, precedence relations among the tasks that are originated from OpenMP 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. The paper briefly describes the main aspects of the runtime implementation required to support precedences relations in OpenMP. The paper focuses on the evaluation of the proposal through its use two benchmarks: NAS LU and ASCI Seep3d.

1 Introduction

OpenMP [8] has emerged as the standard programming model for shared–memory parallel programming. One of the features available in the current definition of OpenMP is the possibility of expressing multiple–levels of parallelism [3, 6, 7, 9, 10]. When applying multi–level parallel strategies, it is common to face with the need of expressing pipelined computations in order to exploit the available parallelism [11, 13, 5]. These 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 this parallelism, the programmer has to define complex synchronization data structures and use synchronization primitives along the program, sacrificing readability and maintainability.

In this paper we propose and evaluate an extension to the OpenMP programming model with a set of new directives and clauses to specify generic task graphs and an associated assignement of work to threads.

2 Extensions to OpenMP

In this section we summarize the 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. An initial definition of the extensions to specify precedences was described in [4].

2.1 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.

2.1.1 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$OMP SINGLE NAME(name_ident)
...
C$OMP END SINGLE

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.

 http://dx.doi.org/10.1109/ICPP.2003.1240576

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 appropiate thread synchronizations, depending on the SCHEDULE strategy specified to distribute iterations.

2.1.2 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:

```
task_id = name_ident | name_ident,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. Section 3 describes the runtime that performs this translation.

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.

```
name_ident[:task_id]+
```

Therefore, the task_id construct might take the following syntax:

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

Figure 1 shows a multilevel example. Two nested loops have been parallelized although they are not completely parallel. Some parallelism might be exploited according to the data depedences caused by the use of A(k-1,j-1) in iteration (k,j). Both parallel loops have been named and the appropiate precedences have been defined to ensure that data dependendes 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.

```
C$OMP PARALLEL DO NAME (outerloop)
  do k = 1, N
C$OMP PARALLEL DO NAME (innerloop)
    do j = 1, N
C$OMP PRED((outer_loop,k-1):(inner_loop, j-1))
      ...
      A(k,j)=A(k-1,j-1)*A(k,j)
      ...
C$OMP SUCC((outer_loop,k+1):(inner_loop, j+1))
    enddo
  enddo
```

Figure 1. Example of multilevel code with precedences.

3 Runtime Support

In this section we describe the support required from the runtime system to efficiently implement the language extensions that specify precedence relations. The runtime system usually offers mechanisms to create/resume parallelism plus some basic synchronization mechanisms such as mutual exclusive execution (critical regions), ordered executions (ticketing) and global synchronizations (barriers). All these functionalities, including support for multiple levels of parallelism, are provided by the NthLib library [7] supporting the code generated by the NanosCompiler [3]. The proposal in this paper requires explicit point–to–point synchronization mechanisms, so the following subsections describe the most important implementation aspects of the precedences module in the NthLib library.

3.1 Thread Identification

Any runtime system supporting the standard OpenMP specification has to be able to identify each thread in a parallel region. The standard defines that this identifier has to be an integer number in the range `[0...numthreads-1]`. Obviously, our runtime system is compliant with this requirement. We call this identifier the `Local Thread Identifier`.

When dealing with nested parallelism it is possible to define a parent/son relationship between threads. We say that a thread is `son` of a `parent` thread when the first one has been spawned by the second one. Due to the nesting of several `PARALLEL` constructs, the `Local Thread Identifier` is not unique anymore. To avoid this, the runtime system supports what we call the `Global Thread Identifier`. This identifier is composed of all the `Local Thread Identifier` of its ancestor threads. The `Global Thread Identifier` can be understood as a coordinates that locate each thread in the parallelism tree.

3.2 Work–Sharing Descriptor

For each named work–sharing, a work–sharing descriptor (`ws_desc`) is allocated by the runtime. Depending on the type of the work–sharing construct, the `ws_desc` contains different information. In case of a `DO` work–sharing construct it contains the lower and upper bounds of the induction variable, the iteration step, the scheduling applied, and the number of threads currently executing the loop. Once a thread participating in the execution of the loop starts executing, the `ws_desc` descriptor is initialized with all the information mentioned above. In case of a `SECTION` or `SINGLE` construct, similar information is stored in the `ws_desc`.

As it has been showed in section 2, nested parallelism might cause that several instances of the same work–sharing construct execute concurrently. In the case where the work–sharing construct is named by the programmer, the runtime has to be able of distinguish from all its instances. This is achieved by defining an unique identifier for work–sharings similary to what has been defined for threads: a work–sharing construct can be identified with the identifier supplied in the `NAME` clause plus the `Global Thread Identifier` of any of the threads executing it, but removing their `Local Thread Identifier` in the `Global Thread Identifier`.

Nested parallelism also causes problems related to the amount of memory used by the runtime to represent all instances of the same work–sharing construct. Suppose the source code in figure 1. Following the parallelism definition, each thread in the outermost level of parallelism will create several instances of the same `innerloop` work–sharing. The number of instances depends on the number of iterations assigned to each thread in the outermost level. The runtime should allocate enough `ws_desc` for representing them, what would lead to the fact that the amount of memory depends on the number of iterations of the `outerloop` loop. This is something that our runtime implementation wants to avoid. As the instances of the `innerloop` caused by a thread executing in the `outerloop` will be executed one ofter another, the runtime can allocate only one `ws_desc` for all of them and reuse it. This memory reuse suppose an appropiate optimization to bound the amount of allocated memory.

As it was mentioned before the necessary information for the translation mechanism is contained in the `ws_desc`. This information summarizes the per thread work–sharing state and allows the runtime to know whether the work-sharing has been initialized or not, which threads have started executing it or have finished their execution. For a `SECTION` or `SINGLE` construct, the `Local Thread Identifier` of the thread executing the construct is stored in the `ws_desc`. In case a `DO` work–sharing con-

struct, the ws_desc contains the current iteration being executed per each participating thread.

3.3 Thread Synchronization.

For each pair of named work–sharings that generate parallel tasks related with precedence relations, the runtime allocates a precedence descriptor. The precedence descriptor includes information describing the work–sharings constructs plus some amount of memory to be used for the synchronizations. When a thread executing on a named work–sharing construct encounters a SUCC/PRED directive, access to the precedence descriptor and obtains a memory location to synchronize. The memory location is used as a dependence counter. Waiting for a precedence release at runtime means an increment of this counter and spinning until the counter becomes less than one. The release of a precedence means a decrement of the same counter.

Routines nthf_def_prec and nthf_free_prec are provided to define/release precedences at runtime. The main arguments for these routines are a precedence descriptor plus the Global Thread Identifier of the thread to synchronize with. 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.

A thread encountering a PRED or SUCC directive is forced to synchronize with the thread executing the task_id in the directive. The runtime is in charge of supplying a memory address to make possible the synchronization. The runtime uses the Global Parallel Identifier of the target thread of the synchronization and combines it with the Global Parallel Identifier of the thread invoking the runtime to synchronize. Both Global Parallel Identifier are used as a sort of coordinates to establish the memory location. In order to get the Global Parallel Identifier of the target thread, the runtime translates each task_id in the directive to a Local Parallel Identifier of the thread executing that task. The way the runtime produces this translation is explained in the next section. Therefore, during a synchronization, two different phases might be distinguished: a Translation Phase where all the task_id are translated to Local Parallel Identifier in order to build the Global Parallel Identifier of the target thread; and a Synchronization Phase where the memory location is determined and the synchronization takes place (increment or decrement of the dependence counter).

When a precedence is defined between a pair of SECTION constructs, one counter has to be allocated in the precedence descriptor. When the precedence involves a DO and a SECTION constructs, the runtime has to allocate as many counters as threads execute the parallel loop, each one to support the comunication between the thread executing the SECTION construct and any of the threads executing the loop. When a precedence relation involves two DO work–sharing constructs, the runtime has to allocate a matrix of counters (with as many rows and columns as the number of threads executing in each loop). In particular, as many counters as pairs consumer/producer need to be allocated.

3.4 TaskId–Thread Translation.

In this section we describe the basic data structures and services available to perform the translation between the task_id supplied by the programmer and the thread executing the task. Routine nthf_task_to_thread implements the translation mechanism in the runtime library. It receives a ws_desc as argument and returns the Local Parallel Identifier of the thread executing the target task_id.

Three main problems have to be faced by the runtime:

a) When the translation mechanism is invoked over a ws_desc that has not been initialized yet, the runtime has to deal with this situation because the translation will not be possible until the ws_desc is totally defined.

b) When invoking the translation mechanism over a DO construct, it is possible that the ws_desc is totally defined, but the iteration required has not started its execution yet. The translation can be performed, but in case another translation is required to traverse the next level of parallelism, it has to be ensured that this translation is performed over the correct instantation of the work–sharing construct required in the task_id supplied by the programmer.

c) In the same context of b), it is possible that the iteration that was required in the first translation was already executing. Then another problem might appear. In case this iteration ends before the second translation finishes, the second translation has to be invalidated because it is possible that has been performed over a ws_desc corresponding to the instantation of a task not involved in the synchronization in course. The runtime has to be able to detect this situation and offer correct actions to deal with it.

Our implementation faces a) and b) by applying blocking mechanisms, but achieves the fact of never blocking a thread that is releasing a precedence. The c) situation is solved by a binding mechanism that binds a thread that has performed a translation to the ws_desc where the translation has been done.

For more details on this subject, a complete description of this mechanism can be found in [5].

```
!$omp parallel default(shared)
!$omp& private(k,iam)
!$omp master
    mthreadnum=omp_get_num_threads()-1
    if (mthreadnum.gt.jend-jst)
  1    mthreadnum=jend-jst
!$omp end master
    iam = omp_get_thread_num()
    isync(iam) = 0
!$omp barrier
    do k = 2, nz -1
       call jacld(k)
       call blts( isiz1, isiz2,
  1               isiz3,
  2                nx, ny, nz, k,
  3                omega,
  4                rsd, tv,
  5                a, b, c, d,
  6                ist, iend, jst,
  7                jend,nx0, ny0)
    end do
!$omp end parallel
```

Figure 2. Source code for NAS LU application.

```
 subroutine blts(...)
 ...
 iam = omp_get_thread_num()
 if (iam.gt.0 .and. iam.le.mthreadnum)
     neigh=iam-1
     do while (isync(neigh).eq.0)
!$omp flush(isync)
     end do
     isync(neigh)=0
!$omp flush(isync)
 endif
!$omp do
 do j=jst,jend
   ...
 enddo
!$omp end do nowait
 if (iam .lt. mthreadnum) then
      do while (isync(iam) .eq. 1)
!$omp flush(isync)
      end do
      isync(iam) = 1
!$omp flush(isync)
 endif
 ...
 end
```

Figure 3. Source code for NAS LU application.

4 Evaluation

We have tested our run–time implementation on a SGI Origin2000 with 64 R10000 processors (250 Mhz) and Irix 6.5. The parallel code is automatically generated using the NanosCompiler [3] to transform the source code annotated with the new precedence directives to run on NthLib [7].

4.1 NAS LU

LU is a simulated CFD application that comes with the NAS benchmarks. 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.

4.1.1 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.

Figures 2 and 3 show the structure of the source code for this version. Notice that the programmer has to introduce the FLUSH construct to ensure memory consistency for the integer vector `isync` used for synchronization. The vector is not padded, so false sharing problems may appear in the synchronization execution degradating performance. The lefmost bar in figure 6 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.

```
      ...
!$omp parallel default(shared)
!$omp& private(k1,k2,bk,a,b,c,d)
!$omp do name (l_bk)
    do bk = 1, nblocksk
     do bj=1,nblocksj
!$omp pred (l_bk, bk-1)
      do bi=1,nblocksi
       call jacld(bk,bj,bi,a,b,c,d)
       call blts( isiz1, isiz2, isiz3,
 >              nx, ny, nz, bk,bj,bi,
 >              omega,rsd, tv,
 >              a, b, c, d,
 >              ist, iend, jst, jend,
 >              nx0, ny0)
      enddo !bi
!$omp succ (l_bk, bk+1)
     enddo !bj
    end do !bk
!$omp end do nowait
!$omp end parallel
      ...
```

Figure 4. NAS LU application single level parallelism and precedences.

4.1.2 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 2 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 4 shows the new source code with precedence directives. The middle bar in figure 6 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.

```
!$omp parallel default(shared)
!$omp& private(k1,k2,bk,tv,a,b,c,d)
!$omp do name (l_bk)
   do bk = 1, nblocksk
!$omp parallel
!$omp do name (l_bj) private(bi,k)
     do bj=1,nblocksj
       do bi=1,nblocksi
!$omp pred((l_bk,bk-1):(l_bj,bj))
!$omp pred(l_bj,bj-1)
         call jacld(bk,bj,bi,a,b,c,d)
         call blts(isiz1, isiz2, isiz3,
     >           nx, ny, nz, bk,bj,bi,
     >           omega,
     >           rsd, tv,
     >           a, b, c, d,
     >           ist, iend, jst, jend,
     >           nx0, ny0)
!$omp succ(l_bj,bj+1)
!$omp succ((l_bk,bk+1):(l_bj,bj))
       enddo !bi
     enddo !bj
!$omp enddo nowait
!$omp end parallel
   enddo !bk
!$omp enddo nowait
!$omp end parallel
```

Figure 5. NAS LU application with two levels of parallelism and precedences.

4.1.3 Two levels with precedences

This version exploits near all the parallelism present in the computation. Figure 5 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 exploting 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 figure 6 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.

4.2 US DOE ASCI Sweep3D

The Sweep3D benchmark uses a multidimensional wavefront algorithm for discrete ordinates deterministic par-

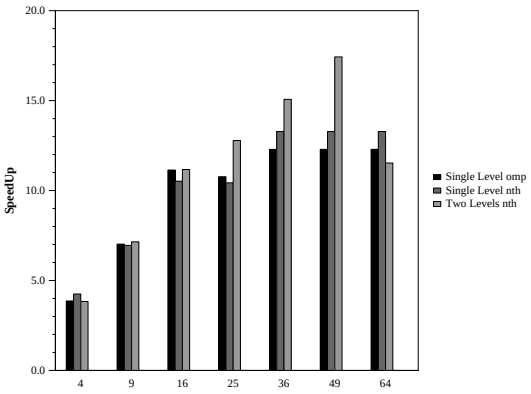

Figure 6. Performance for LU NAS application.

```
      do k=1,nk
      do j=1,nj
      ...
      do m=1,6
      do i=1,ni
        phijk(j,k,m)=phijk(j,k,m)+...
        phiik(i,k,m)=phiik(i,k,m)+...
        phiij(i,j,m)=phiij(i,j,m)+...
      enddo
      do i=1,ni
        face(i,j,k)=face(i,j,k)+
1        phijk(j,k,m)+phiik(i,k,m)+
2        phiij(i,j,m)
      enddo
      enddo
      enddo
      ...
      enddo
      enddo
```

Figure 7. Main core for the source code of SWEEP3D application.

ticle transport simulation. The main input data is a 3 dimensional array named `face`. The core computation presents six reductions in all dimensions, what forbits the parallelization of the computation in any of the dimensions. Figure 7 shows the structure of the source code. Three nested loops implement the computation along the 3 dimensional input matrix. Each (i,j,k) element in the 3 dimensional array has to be computed after elements `(i-1,j,k)`, `(i,j-1,k)` and `(i,j,k-1)` because of the reductions in each dimension. We have followed the same strategy as in the LU benchmarck. The loops `k`, `j` and `i` have been blocked. Thus, a block `(bi,bj,bk)` now has to be computed after blocks `(bi-1,bj,bk)`, `(bi,bj-1,bk)` and `(bi,bj,bk-1)`. Loops `bk` and `bj` have been parallelized and named. The computation for each `(bi,bj,bk)` block is enclosed by the precedence directives that ensures a correct execution. Notice that the proposed parallelization forces us to introduce some changes in the access to the reduction planes. Because of the blocking algorithm, now the reduction planes are accessed in a way that the dimension that is consecutively allocated in memory has been cutted and distributed among different threads. This leads to false sharing problems that completely sinks performance. To avoid this problem, we have been forced to change the memory allocation of the reductions planes.

As in the case for the LU application, two different parallel strategies can be implemented: a parallel version just exploiting one level of parallelism (`1l sweep`), where only the `bk` loop is parallelized, and a parallel version that exploits two levels of parallelism by parallelizing loops `bk` and `bj` (`2l sweep`). We have tested the two versions of the benchmark with an input matrix of 50x50x50 elements. Figure 8 shows the performance numbers for the SWEEP3D application. Version `1l sweep` works with a blocking factor of 2, so 25 blocks are defined. When running on 25 processors the application reaches a speedup around 13. The `2l sweep` version can be executed with more threads, as a second level of parallelism is exploited. This version reaches its maximum speed–up when executing with 40 threads: 17.6, about a 25% more of performance. As in the case of the NAS LU benchmark, the SWEEP3D code was a completely sequential code from the point of view of the current OpenMP definition. With the new proposed directives this code can be parallelized.

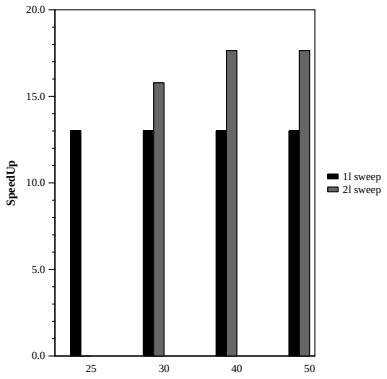

Figure 8. Performance for SWEEP3D application.

The NAS LU and SWEEP3D benchmarks allow us to demonstrate the need to enlarge the synchronization schemes in the OpenMP programming model. We have

seen that the kind of requiered synchronizations can be included in the programming model without critical changes in it. The experiences with those benchmarks also showed that nested parallelism can be combined with the new synchronization schemes without introducing unacceptable runtime overheads.

5 Conclusions

In this paper we have presented a set of extensions to the OpenMP programming model oriented towards the specification of explicit point–to–point synchronizations.
We have detected that current OpenMP standard includes very simple synchronization constructs. When the available parallelism in an application is not expressable with current OpenMP constructs, the programmer is forced to explicitly program the thread synchronizations in the application code. We show that it is necessary to enlarge the OpenMP standard to allow the programmer the specification of the necessary thread synchronizations due to data dependences. We have tested our proposal with two applications, all of them parallelized with OpenMP. The evaluation has showed the lack of generic synchronization mechanisms, powerful enough to support the needs of the tested applications. In the paper we have demonstrated that our proposal covers the mentioned needs without introducing deep changes in the OpenMP programming model. The evaluation showed that with the proposed extensions, programmers can get from 10 to 30 percent of performance improvement.
The evaluation shows that our proposal and runtime implementation offers enough expressiveness and does not introduce large overheads due to runtime implementation.

Acknowledgments

This research has been supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIC2001-0995-C02-01.

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] 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.

[4] M. Gonzalez, E. Ayguadé, X. Martorell, J. Labarta, N. Navarro and J. Oliver. Precedence Relations in the OpenMP Programming Model. Second European Workshop on OpenMP, EWOMP 2000 (September 2000).

[5] M. Gonzalez, E. Ayguadé, X. Martorell and J. Labarta. Defining and Supporting Pipelined Executions in OpenMP. Workshop on OpenMP Applications and Tools (WOMPAT'01) August 2001.

[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] T. Rauber and G. Runge. Compiler support for task scheduling in hierarchical execution models. Journal of Systems Architecture, 45:483-503, 1998

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

[13] 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