Int J Parallel Prog (2009) 37:292–305
DOI 10.1007/s10766-009-0101-1

A Proposal to Extend the OpenMP Tasking Model with Dependent Tasks

Alejandro Duran · Roger Ferrer · Eduard Ayguadé · Rosa M. Badia · Jesus Labarta

Received: 4 March 2009 / Accepted: 7 April 2009 / Published online: 25 April 2009

Abstract Tasking in OpenMP 3.0 has been conceived to handle the dynamic generation of unstructured parallelism. New directives have been added allowing the user to identify units of independent work (tasks) and to define points to wait for the completion of tasks (task barriers). In this document we propose extensions to allow the runtime detection of dependencies between generated tasks, broading the range of applications that can benefit from tasking or improving the performance when load balancing or locality are critical issues for performance. The proposed extensions are evaluated on a SGI Altix multiprocessor architecture using a couple of small applications and a prototype runtime system implementation.

Keywords OpenMP · Task parallelism · Programming models · Tasks synchronization

A. Duran (✉) · R. Ferrer · E. Ayguadé · R. M. Badia · J. Labarta
Computer Sciences Department, Barcelona Supercomputing Center, Jordi Girona, 31, Barcelona, Spain
e-mail: alex.duran@bsc.es

R. Ferrer
e-mail: roger.ferrer@bsc.es

E. Ayguadé
e-mail: eduard.ayguade@bsc.es

R. M. Badia
e-mail: rosa.m.badia@bsc.es

J. Labarta
e-mail: jesus.labarta@bsc.es

E. Ayguadé · J. Labarta
Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya,
Jordi Girona, 1–3, Barcelona, Spain

R. M. Badia
Consejo Superior de Investigaciones Cientificas, Barcelona, Spain

1 Introduction

OpenMP grew out of the need to standardize the directive languages of several vendors in the 1990s. It was structured around parallel loops and was meant to handle dense numerical applications. The simplicity of its original interface, the use of a shared memory model, and the fact that the parallelism of a program is expressed in directives that are loosely-coupled to the code, all have helped OpenMP become well-accepted today.

The latest OpenMP 3.0 specification released [1] includes tasking, which has been conceived to handle the dynamic generation of unstructured parallelism. This allows programmers to parallelize program structures like `while` loops and recursive functions more easily and efficiently. When a thread in a parallel team encounters a `task` directive, the data environment is captured. That environment, together with the code represented by the structured block, constitutes the generated task. The data-sharing attribute clauses `private`, `firstprivate`, and `shared` determine whether variables are private to the data environment, copied to the data environment and made private, or shared with the thread generating the task, respectively. The task may be executed immediately or may be queued for execution. All tasks created by a team in a parallel region are completed at the next `barrier`. It is also possible to wait for all tasks generated by a given task (whether implicit or explicit) using the `taskwait` directive.

The Intel *work-queueing* model [2] was an early attempt to add dynamic generation of `tasks` to OpenMP. This proprietary extension to OpenMP allowed hierarchical generation of tasks by nesting `taskq` constructs. Synchronization of descendant tasks was controlled by means of implicit barriers at the end of `taskq` constructs. Tasks have to be defined in the lexical extent of a `taskq` construct. The Nanos group at UPC proposed *dynamic sections* as an extension to the standard `sections` construct to allow dynamic generation of tasks [3]. Direct nesting of `section` blocks was allowed, but hierarchical synchronization of tasks was only possible by nesting parallel regions.

An early approach to include dependences in programming models that exploit task-level parallelism is Jade [4]. Programs in Jade are written standard serial, imperative language, and Jade constructs are used to declare how parts of the program access data. Jade programmers must specify three things: how data is decomposed into the atomic units that the program will access, how to decompose a sequential program into tasks, and a description of how each task will access data when it runs, indicating if a task reads or writes a given data. Given this information, the implementation automatically extracts and exploits the task-level concurrency present in the computation.

SMP superscalar (*SMPSs*) [5] has recently been proposed as a task based parallel programming model. Similarly to OpenMP 3.0 the code is annotated with pragmas, although in this case the pragmas annotate when a function is a `task` and inline annotation is not allowed. Another difference is that the pragmas indicate the direction (`input`, `output`, or `inout`) of the functions' parameters with the objective of giving hints to the runtime in order to discover the actual tasks' data dependencies. *SMPSs* runtime also implements data renaming, allowing to eliminate false dependencies in

the task dependency graph. The SuperMatrix [6] approach is similar in motivation and in technique to *SMPSs*. However, SuperMatrix is exclusively focused on linear algebra algorithms. No pragmas or specific programming model is defined, since the runtime directly considers for parallelization a set of linear algebra routines. Supermatrix also implements a task dependency analysis, but in this case data renaming is not considered.

There have been several attempts to include dependences in the scope of sections in OpenMP. The Nanos group proposed the `pred` and `succ` constructs to specify precedence relations among statically named `sections` in OpenMP [7]. Authors in [8] also proposed an extension to define a name for `section` and to specify that a section `dependson` another named section.

2 Motivation

Task parallelism in OpenMP 3.0 gives programmers a way to express patterns of concurrency that do not match the worksharing constructs defined in OpenMP 2.5. The extension in 3.0 addresses common operations like complex, possibly recursive, data structure traversal, and situations which could easily cause load imbalance. However tasking, as currently proposed in 3.0, may still be too rigid too express all the parallelism available in some applications, specially when we want to scale to systems with large number of processors.

To motivate the proposal in this paper we use one of the examples [9] that was used to test the appropriateness and performance of the tasking proposal in OpenMP 3.0: the sparseLU kernel shown in Fig. 1. This kernel computes an LU matrix factorization on a blocked matrix. The matrix is organized as a hypermatrix with pointers to the actual blocks of data (which may not be allocated due to the sparsity of the original matrix). In this kernel, once `lu0` is computed (line 14), all instances of `fwd` and `bdiv` can be executed in parallel (lines 18 and 22, respectively). Each pair of instances `fwd` and `bdiv` allow the execution of an instance of `bmod` (line 27). Across consecutive iterations of the `kk` loop there are dependences between each instance of `bmod` and instances of `lu0`, `fwd`, `bdiv` and `bmod` in the next iteration.

With these data dependences in mind, the programmer could use traditional worksharing directives in OpenMP to partially exploit the parallelism available in the kernel, for example using `for` to distribute the work in the loops on lines 16 and 20. It would be necessary to apply loop distribution to isolate the loop that executes the multiple instances of function `bdiv` and exploit the parallelism that exist among the instances of functions `fwd` and `bdiv`. Due to the sparseness of the matrix, a lot of imbalance may exist, forcing the programmer to use dynamic scheduling of the iterations to have good load balance. The resulting code is shown in Fig. 2.

Using the tasking execution model proposed in 3.0, the code restructuring is quite similar, as shown in Fig. 3; however tasks allow to only create work for non-empty matrix blocks. We also create smaller units of work in the `bmod` phase with an overhead similar to the outer loop parallelization. This reduces the load imbalance problems. Notice that all threads are involved in task generation due to the combination of the `for` work distributor and the `task` generator. The `nowait` clause in line 9 allows

```
1 void fwd(float *diag, float *col);
2 void bmod(float *row, float *col, float *inner);
3 void bdiv(float *diag, float *row);
4 void lu0(float *diag);
5
6 #define NB 100
7 #define B 50
8 float *A[NB][NB];
9
10 int sparseLU() {
11    int ii, jj, kk;
12
13    /* allocation of blocks with
14       A[ii][jj] = (float *)malloc(B*B*sizeof(float)) */
15
16    for (kk=0; kk<NB; kk++) {
17       lu0(A[kk][kk]);
18       /* fwd phase */
19       for (jj=kk+1; jj<NB; jj++)
20          if (A[kk][jj] != NULL)
21             fwd(A[kk][kk], A[kk][jj]);
22       /* bdiv and bmod phases */
23       for (ii=kk+1; ii<NB; ii++)
24          if (A[ii][kk] != NULL) {
25             bdiv (A[kk][kk], A[ii][kk]);
26             for (jj=kk+1; jj<NB; jj++)
27                if (A[kk][jj] != NULL)
28                  {
29                     if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
30                     bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
31                  }
32             }
33    }
34 }
```

Fig. 1 Main code of the sequential SparseLU kernel

```
1 int sparseLU() {
2    int ii, jj, kk;
3
4    for (kk=0; kk<NB; kk++) {
5       lu0(A[kk][kk]);
6 #pragma omp parallel
7 {
8       /* fwd phase */
9 #pragma omp for schedule(dynamic, 1) nowait
10       for (jj=kk+1; jj<NB; jj++)
11          if (A[kk][jj] != NULL)
12             fwd(A[kk][kk], A[kk][jj]);
13
14       /* bdiv phase */
15 #pragma omp for schedule(dynamic, 1)
16       for (ii=kk+1; ii<NB; ii++)
17          if (A[ii][kk] != NULL)
18             bdiv (A[kk][kk], A[ii][kk]);
19
20       /* bmod phase */
21 #pragma omp for schedule(dynamic, 1) private(jj)
22       for (ii=kk+1; ii<NB; ii++)
23          if (A[ii][kk] != NULL)
24             for (jj=kk+1; jj<NB; jj++)
25                if (A[kk][jj] != NULL)
26                  {
27                     if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
28                     bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
29                  }
30    }
31 }
32 }
```

Fig. 2 Main code of the OpenMP 2.5 SparseLU kernel

```
1 int sparseLU() {
2    int ii, jj, kk;
3
4#pragma omp parallel
5    for (kk=0; kk<NB; kk++) {
6#pragma omp single
7       lu0(A[kk][kk]);
8       /* fwd phase */
9#pragma omp for nowait
10      for (jj=kk+1; jj<NB; jj++)
11         if (A[kk][jj] != NULL)
12#pragma omp task firstprivate(kk, jj)
13            fwd(A[kk][kk], A[kk][jj]);
14      /* bdiv phase */
15#pragma omp for
16      for (ii=kk+1; ii<NB; ii++)
17         if (A[ii][kk] != NULL)
18#pragma omp task firstprivate(kk, ii)
19            bdiv (A[kk][kk], A[ii][kk]);
20
21      /* bmod phase */
22#pragma omp for private(jj)
23      for (ii=kk+1; ii<NB; ii++)
24         if (A[ii][kk] != NULL)
25            for (jj=kk+1; jj<NB; jj++)
26               if (A[kk][jj] != NULL)
27#pragma omp task firstprivate(kk, jj, ii)
28                  {
29                     if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
30                     bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
31                  }
32   }
33 }
```

Fig. 3 Main code of SparseLU with OpenMP 3.0 tasks

the parallel execution of `fwd` and `bdiv` task instances. The implicit barrier at the end of loop in line 15 forces the dependencies between pairs of `fwd`/`bdiv` with `bmod` inside a single `kk` iteration. Similarly, the implicit barrier at the end of loop in line 22 forces the dependencies across consecutive iterations of loop `kk`.

As we previously pointed, there exists more parallelism in this kernel that cannot be exploited with the current task definitions: (1) parallelism that exists between tasks created in lines 13 (fwd) and 19 (bdiv) and tasks created in line 30 (bmod) inside the same `kk` iteration; and (2) parallelism that exists across consecutive iterations of the `kk` loop. Figure 4 shows the dependences among nodes in the task graph. The extensions proposed in this paper intend to exploit this parallelism.

3 Proposed OpenMP Extensions

In this section we propose a set of architecture independent extensions to the OpenMP 3.0 tasking model to express dependencies between tasks. In Sect. 3.6 we comment how the basic proposal could be further extended to handle other target architectures, such as heterogeneous multicores with local memories (e.g. the Cell/B.E. architecture [10]).

3.1 Extension to the Task Construct

Tasks are the most important new feature of OpenMP 3.0. A programmer can define deferrable units of work, called tasks, and later ensure that all the tasks defined up

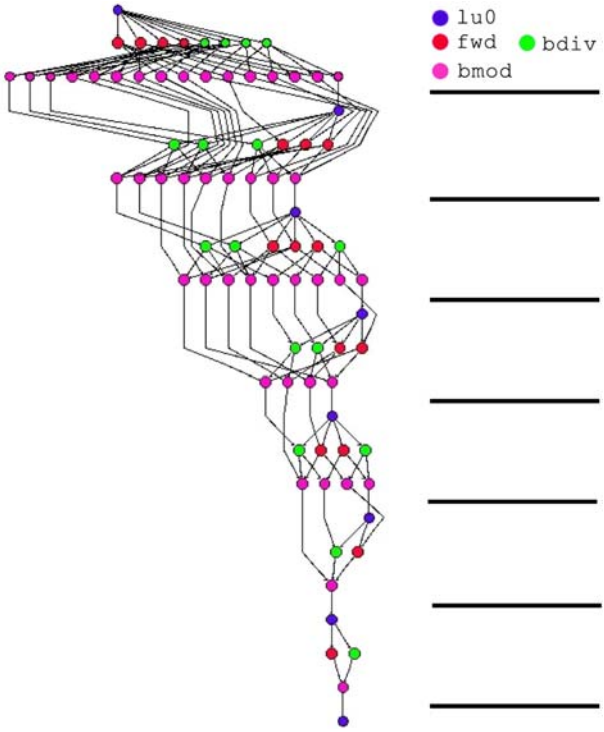

Fig. 4 Dependence graph among tasks in SparseLU

to some point have finished. Tasks are created in the context of a team of threads. In OpenMP such team of threads is created with the `parallel` construct (an implicit team with just one thread is created at the beginning of the parallel application). A task is created when the code reaches the *task construct*, defined as follows:

```
#pragma omp task [clause-list]
  structured-block
```

OpenMP defines several *clauses* that can be used in the task construct. They are `shared`, `private`, `firstprivate` and `untied`. The first three are used for setting data sharing attributes of variables in the task body; the last one specifies that a the task can be resumed by a different thread after a possible *task switching point*. Data sharing clauses have the following syntax:

- `shared (variable-list)`
- `firstprivate (variable-list)`
- `private (variable-list)`

where *variable-list* is a list of identifiers. Naming a variable inside a data sharing clause explicitly sets the data sharing attribute for the variable inside the task construct. References in the task construct to variables whose data sharing attribute is private or firstprivate will not refer to the original variable but to a private storage of the task. Firstprivate ones, in addition, will have such storage initialized with the value of the

original variable when the program execution reaches the `task` construct. References to variables whose data sharing attribute is shared will refer to the original variable.

Our proposal extends the `task` construct with some additional clauses that are used to derive dependence relationships among tasks. When a task requires a previously computed variable we say that it has an *input* dependence and we will express it with the `input` clause. Similarly, when a task is generating a variable that might be required later by another task we say that it has an *output* dependence and we use an `output` clause to express such dependence. A clause `inout` exists to express the case when a task requires a variable and generates a new value for it, meaning an *input–output* dependence. The syntax of these clauses is shown below:

- `input(data-reference-list)`
- `output(data-reference-list)`
- `inout(data-reference-list)`

Dependences are expressed by means of *data-reference-lists*, which are a superset of a *variable-list*. A *data-reference* in such a list can contain a variable identifier but also references to subobjects. References to subobjects include array element references (like `a[4]`), array sections (like `a[3:6]`, defined below), field references (like `a.b`) and shaping expressions (like `[10][20] p`, defined below).

Since C does not have any way to express ranges of an array, we have borrowed the *array-section* syntax from Fortran 90. These array sections, with syntax `a[e1:e2]`, designate all elements from `a[e1]` to `a[e2]` (both ends are included and `e1` shall yield a lower or equal value than `e2`). Multidimensional arrays are eligible for multidimensional array sections (like `a[1:2][3:4]`). While not technically naming a subobject, non-multidimensional array section syntax can also be applied to pointers (i.e.: `pA[1:2]` is valid for `int *pA`, but note that `pB[1:2][3:4]` is invalid for `int **pB`, also note that `pC[1:2][3:4]` is valid for `int (*pC)[N]` and so it is `pD[1:2][3:4][5:6]` for `int (*pD)[N][M]`). For syntactic economy `a[:x]` is the same as `a[0:x]` and, only for arrays of `N` elements, `a[x:]` and `a[:]` mean respectively `a[x:N-1]` and `a[0:N-1]`. Designating an array (i.e.: `a`) in a data reference list, with no array section nor array subscript, is equivalent to the whole array-section (i.e.: `a[:]`).

Shaping expressions are a sequence of dimensions, enclosed in square brackets, and a data reference, that should refer to a pointer type (like `[10][20] p`). These shaping expressions are aimed at those scenarios where an array-like structure has been allocated but only a pointer to its initial element is available. Shaping expressions goal is returning back such unavailable structural information to the compiler.

3.2 Extension to the Tasking Execution Model

Explicit tasks are dynamically created and the memory region specifiers in *data-reference-list* are used to build a dependence task graph:

- *Data references* specified in `input` or `inout` clauses are checked against the *data references* specified in `output` or `inout` clauses of all tasks, in the scope of the

```
1 int i;
2 int a[100];
3 int y;
4 ...
5 #pragma omp task output(a) // taskA
6 {
7     for (i = 0; i < 100; i++)
8       a[i] = i + 1;
9 }
10 #pragma omp task input(a[0:49]) output(y) // taskB
11 {
12     y = 0;
13     for (i = 0; i < 50; i++)
14     y += a[i];
15 }
16 #pragma omp task input(y) // taskC
17 {
18 printf("%d\n", y);
19 }
20 ...
```

Fig. 5 Example with dependent tasks

same parent task, in execution or pending to be executed. If there is a match, a true dependence is added between both tasks.

- *Data references* specified in the `output` or `inout` clauses are checked against *data references* specified in `input`, `output` or `inout` clauses of all tasks, in the scope of the same parent task, in execution or pending to be executed. If there is a match, a false dependence appears. This dependence could be eliminated by dynamically renaming the memory region specified in *data reference*. Renaming is an optional feature of the runtime which can be selectively activated to increase the potential parallelism in the task graph.
- A variable in a `shared` data clause, but not in a `input`, `output` or `inout` clause, indicates that the variable is accessed inside the task but it is not affected by any data dependence in the current scope of execution (or is protected by using another mechanism).
- A task is ready for execution once all its dependencies in the task graph are honored.

Figure 5 shows an example with three tasks. Task `taskA` generates values for all the elements of array `a`. Task `taskB` has an input dependence due to this generated array and generates the value of the scalar `y`. This scalar is an input dependence to task `taskC`. A valid execution order for these three tasks involves running first `taskA`, then `taskB` and later `taskC`. The result printed in task `taskC` should be 1275.

Figure 6 shows another simple example of dependent tasks in which scalar variables are the subject of dependence detection. In this example, `taskB` and `taskC` can be executed in parallel if the false dependence created by `input(x)` in `taskB` and `inout(x)` in `taskC` is removed. In this case, even if `taskC` completes execution before `taskB` starts, renaming should ensure that `taskB` receives the value of `x` produced by `taskA`. If the runtime decides not to do renaming (or it is unable to do it), `taskC` has to be executed once `taskB` finishes. `TaskD` should always print 4.

As previously specified, dependencies are detected inside the scope of the same parent task. Assume the code excerpt shown in Fig. 7. In this case the programmer

```
1 ...
2#pramga omp task output(x) // taskA
3 {
4 x = 1;
5 }
6#pragma omp task input(x) output(y) // taskB
7 {
8 y = x+1;
9 }
10#pragma omp task inout(x) // taskC
11 {
12 x++;
13 }
14#pragma omp task input(x,y) // taskD
15 {
16 printf("%d\n",x+y);
17 }
18 ...
```

Fig. 6 Another example with dependent tasks

```
1 ...
2#pragma omp task output(x) // taskA
3 {
4 ...
5 }
6#pragma omp task // taskB
7 {
8     ... // codeB.1
9
10    #pragma omp task input(x) // taskC
11    {
12    ...
13    }
14    #pragma omp task output(y) // taskD
15    {
16    ...
17    }
18
19    ... // code B.2
20 }
21#pragma omp task input(y) // taskE
22 {
23 ...
24 }
```

Fig. 7 Incorrect example with nested tasks

cannot expect that `taskC` waits for the termination of `taskA`, which would allow the overlapped execution of `taskA` with `codeB.1` inside `taskB`. And the same with `taskD` and `taskE`. To ensure this the programmer needs to insert `input(x)` and `output(y)` in `taskB`, which precludes the exploitation of any parallelism of `taskA` and `codeB.1`, and `codeB.2` and `taskE`.

Implicit tasks created in `parallel` regions are assumed to be totally independent. It is not allowed to use `input`, `output` and `inout` in a parallel construct.

3.3 Additional Clause and Execution Model For `taskwait`

The execution of a `taskwait` has to force the write-back to memory of any possible data renaming that has been dynamically introduced by the runtime system to eliminate false dependencies.

```
1 ...
2 #pragma omp task output(x)
3 {
4 /* task A */
5 }
6 #pragma omp task input(x)
7 {
8 /* task B */
9 }
10 ...
11 #pragma omp taskwait on(x)
12 total = total + x;
13 ...
```

Fig. 8 Example using the extended `taskwait` pragma

In this paper we also propose to extend the `taskwait` construct as follows

```
#pragma omp taskwait on(data-reference-list)
```

in order to wait for the termination of those tasks whose `output` or `inout` match with `data-reference-list`.

For example, in code shown in Fig. 8, the programmer needs to insert the `taskwait` pragma in order to ensure that the next statement reads the appropriate value for variable `x`, which is generated by `task A`. However, `task B` can run in parallel with the code after the `taskwait` pragma.

3.4 SparseLU Example

Figure 9 shows the SparseLU kernel (shown in Fig. 1) programmed with the extensions proposed in this paper. The programmer identifies four tasks that correspond to the invocation of functions `lu0`, `fwd`, `bdiv` and `bmod`. For example, for function `bmod` the programmer is specifying that the first and second arguments (`row` and `col`) are `input` parameters (they are only read during the execution of the function) and that the third argument (`inner`) is `inout` since it is read and written during the execution of the function. Notice that the annotations are placed on the original sequential version, with no transformations applied to allow the specification of the inherent parallelism available.

3.5 Multisort Example

Figure 10 shows the parallelization of function `cilksort` in Multisort. Dependencies are expressed between the four instances of function `cilksort` and the two first instances of `cilkmerge`. A final dependence between these two instances of `cilkmerge` and the final instance of `cilkmerge` is also required.

3.6 Data Movement in Architectures with Local Memories

The information provided in the `input`, `output` and `inout` clauses can be used by the runtime to move data physical address spaces of the processors executing the

```
1 void fwd(float *diag, float *col);
2 void bmod(float *row, float *col, float *inner);
3 void bdiv(float *diag, float *row);
4 void lu0(float *diag);
5
6 int sparseLU() {
7    int ii, jj, kk;
8
9 for (kk=0; kk<NB; kk++) {
10#pragma omp task inout(A[kk][kk][:B-1][:B-1])
11    lu0(A[kk][kk]);
12    for (jj=kk+1; jj<NB; jj++)
13       if (A[kk][jj] != NULL)
14#pragma omp task input(A[kk][kk][:B-1][:B-1]) inout(A[kk][jj][:B-1][:B-1])
15          fwd(A[kk][kk], A[kk][jj]);
16
17    for (ii=kk+1; ii<NB; ii++) {
18       if (A[ii][kk] != NULL)
19#pragma omp task input(A[kk][kk][:B-1][:B-1]) inout(A[ii][kk][:B-1][:B-1])
20          bdiv (A[kk][kk], A[ii][kk]);
21          for (jj=kk+1; jj<NB; jj++)
22             if (A[kk][jj] != NULL) {
23                if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
24#pragma omp task input(A[ii][kk][:B-1][:B-1], A[kk][jj][:B-1][:B-1])
25                inout(A[ii][jj][:B-1][:B-1])
26                bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
27             }
28       }
29 }
```

Fig. 9 Main code of SparseLU with the proposed dependent tasks

threads (tasks) in the team. For example, in a Cell-like architecture, the memory region specifiers can be used to move data between main memory and the local memories in the *SPE*. The implementation of the *SMPSs* programming model for Cell (named *CellSs* [11]) does it. In addition, the information provided in these clauses can be used to design locality-aware task scheduling policies. For this kind of architectures two additional clauses can be used to partially unspecify dependencies or movement:

- `nodep(data-reference-list)`
- `nomove(data-reference-list)`

Clause `nodep` specifies that the memory *data reference* specified should not be used to build the task graph, so that the programmer is responsible to enforce any data dependence or to avoid any data race in the access to memory in that region. Clause `nomove` specifies that the runtime should not take care of moving data between physical address spaces of the processors executing the threads in the team.

4 Preliminary Evaluation

In order to test the proposal in terms of expressiveness and performance, we have developed the *StarSs* runtime for SMP (named *SMPSs*) and used the Mercurium compiler (source-to-source restructuring tool) [3]. For comparison purposes we also use the reference implementation [12] of the tasking proposal in OpenMP 3.0 based on the *Nanos* runtime and the same source-to-source restructuring tool, and the workqueueing implementation available in the Intel compiler. In the final version of the paper, if accepted for publication in the special issue, authors will also include an evaluation

```
1 void cilksort(ELM *low, ELM *tmp, long size)
2 {
3       long quarter;
4       ELM *A, *B, *C, *D, *tmpA, *tmpB, *tmpC, *tmpD;
5
6       if (size < quick_size) {
7   seqquick(low, low + size - 1);
8   return;
9       }
10      quarter = size / 4;
11
12      A = low; tmpA = tmp;
13      B = A + quarter; tmpB = tmpA + quarter;
14      C = B + quarter; tmpC = tmpB + quarter;
15      D = C + quarter; tmpD = tmpC + quarter;
16
17  #pragma omp task untied output([quarter] A)
18      cilksort(A, tmpA, quarter);
19  #pragma omp task untied output([quarter] B)
20      cilksort(B, tmpB, quarter);
21  #pragma omp task untied output([quarter] C)
22      cilksort(C, tmpC, quarter);
23  #pragma omp task untied output([quarter] D)
24      cilksort(D, tmpD, size - 3 * quarter);
25
26 #pragma omp task untied input([quarter] A, [quarter] B) \
27                         output([size/2] tmpA)
28      cilkmerge(A, A+quarter-1, B, B+quarter-1, tmpA);
29 #pragma omp task untied input([quarter] C, [quarter] D) \
30                         output([size/2] tmpC)
31      cilkmerge(C, C+quarter-1, D, low+size-1, tmpC);
32
33      #pragma omp task untied input([size/2] tmpA, [size/2] tmpC)
34          cilkmerge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
35
36 #pragma omp taskwait
37 }
```

Fig. 10 Cilksort example with the proposed dependent tasks

of the proposal implemented on the same prototype tasking implementation for OpenMP 3.0.

We evaluate how the proposed extension improves the scalability of the SparseLU benchmark that has been used to motivate the proposal. All the executions have been done on an SGI Altix 4700 using up to 32 processors in a cpuset (to avoid interference with other running applications).

Figure 11 shows the speed-up with respect to the sequential execution time. Notice that up to 16 threads the three versions (*taskq*, *task* and *SMPSs*) behave similarly. When more threads are used, load unbalancing starts to be more noticeable and the overheads of tasking are not compensated with the parallel execution. Task barriers between instances of `fwd`/`bdiv` and `bmod` phases (inside iteration `kk`) and between instances of `bmod` and `fwd`/`bdiv` phases (in consecutive iterations of `kk` introduce this load unbalance and overheads. However, *SMPSs* is able to overcome these two limitations by overlapping tasks in these computational phases inside and across iterations of the `kk` loop.

The implementation of *SMPSs* has overheads. Table 1 shows a breakdown of the execution time of the *SMPSs* version of SparseLU. The table shows the percentage of time that each thread is in each phase (*worker threads*' information has been summarized due to space limitations). For this example, the *main thread* invests around the

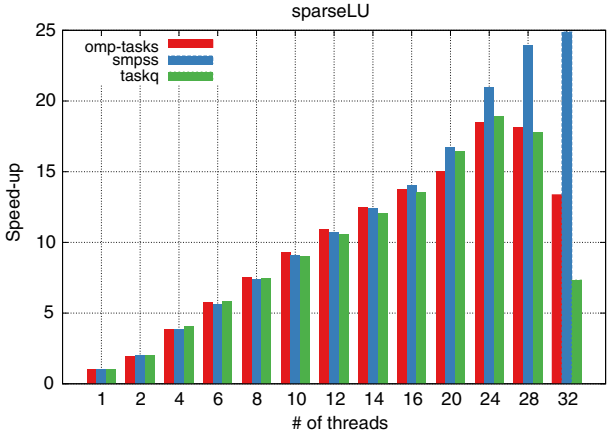

Fig. 11 Speed-up of *taskq*, *task* and *SMPSs* for SparseLU

Table 1 Breakdown of *SMPSs* overheads for the SparseLU with 16 threads

Thread phase	Main thread (%)	Max worker th. (%)	Min worker th. (%)	Avg. worker th. (%)
User code	5.12			
Initialisation	0.13			
Adding task	10.51			
Remove tasks	19.67	2.41	0.86	1.46
Waiting for tasks	0.46	1.95	1.04	1.47
Getting task descr.	0.36	1.28	0.56	1.10
Tasks' execution	63.76	97.43	94.97	95.97

30% of its time in the maintenance of the task graph, and around 65% of its time is left for execution of tasks. The worker threads also suffer of some overheads (around 5%), not only due to the maintenance of the task graph but also to the time the threads are waiting for tasks ready to be executed and the time invested in getting the tasks description. Depending on the application and on the number of threads, these overheads will have more or less impact in the performance, but we are looking for more efficient implementations of the task graph to reduce them.

5 Conclusions

This paper proposes an extension to the OpenMP 3.0 tasking model: data dependent tasks. Data dependencies among tasks are indirectly expressed by specifying the input and output direction of the arguments as well as the memory range used. This is a key difference with respect to previous proposals that were based on the specification of named tasks and `dependson` relationships.

The paper uses one of the application kernels used to demonstrate the expressiveness of tasking in OpenMP 3.0: SparseLU. We motivate the proposal with this kernel and show how its scalability improves with a prototype implementation of the proposal (*SMPSs*). We also use a second example (Cilksort) to illustrate the use of the proposed extensions.

The possibility of expressing input and output direction for the data used by the task provides extra benefits for other multicore architectures, such as for example the Cell/B.E. processor [10] (Cell Superscalar [11]). In this case, the information provided by the programmer allows the runtime system to transparently inject data movement (DMA transfers) between SPEs or between SPEs and main memory.

Acknowledgments The Programming Models group at BSC-UPC is supported by the Spanish Ministry of Science and Innovation (contract no. TIN2007-60625), the European Commission in the context of the SARC project (contract no. 27648) and the HiPEAC Network of Excellence (contract no. IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement. We are thankful to Josep M. Perez from BSC-CNS for his comments to initial versions of this paper.

References

1. OpenMP Architecture Review Board. OpenMP 3.0 Specification. http://www.openmp.org, May 2008
2. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallelism in OpenMP. In 1st European Workshop on OpenMP, September 1999
3. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E. Labarta, J.: Nanos mercurium: a research compiler for openMP. In Proceedings of the European Workshop on OpenMP 2004, October 2004
4. Rinard, M.C., Scales, D.J., Lam, M.S.: Jade: A high-level, machine-independent language for parallel programming. Computer **26**(6), 28–38 (1993)
5. Perez, J.M., Badia, R.M., Labarta, J.: A dependency-aware task-based programming environment for multi-core architectures. In IEEE Cluster 2008 (2008)
6. Chan, E., van Zee, F.G., Quintana-Ortí, E.S., Quintana-Ortí, G., van de Geijn, R.: Satisfying your dependencies with supermatrix. In IEEE International Conference on Cluster 2007 (2007)
7. Gonzàlez, M., Ayguadé, E., Martorell, X., Labarta, J.: Exploiting pipelined executions in OpenMP. In 32nd Annual International Conference on Parallel Processing (ICPP'03), October 2003
8. Sinnen, O., Pe, J., Kozlov, A.: Support for fine grained dependent tasks in OpenMP. In 3rd International Workshop on OpenMP (IWOMP'07) (2007)
9. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in OpenMP. In 3rd International Workshop on OpenMP (IWOMP'07) (2007)
10. Pham, D., Asano, S., Bolliger, M., Day, M.N., Hofstee, H.P., Johns, C., Kahle, J., et al.: The design and implementation of a first-generation cell processor. In IEEE International Solid-State Circuits Conference (ISSCC 2005) (2005)
11. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In Proceedings of the ACM/IEEE SC 2006 Conference, November 2006
12. Ayguadé, E., Duran, A., Hoeflinger, J., Massaioli, F., Teruel, X.: An experimental evaluation of the new openMP tasking model. In Proceedings of the 20th International Workshop on Languages and Compilers for Parallel Computing, October 2007