Extending the OpenMP Tasking Model to Allow Dependent Tasks

Alejandro Duran, Josep M. Perez, Eduard Ayguadé,
Rosa M. Badia, and Jesus Labarta

Barcelona Supercomputing Center (BSC) - Technical University of Catalunya (UPC)

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 paper we propose an extension 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. Furthermore the paper describes our proof-of-concept implementation (SMP Superscalar) and shows preliminary performance results on an SGI Altix 4700.

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 specification released 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 [1] was an early attempt to add dynamic task generation to OpenMP. This proprietary extension to OpenMP allows hierarchical generation of tasks by nesting `taskq` constructs. Synchronization of

R. Eigenmann and B.R. de Supinski (Eds.): IWOMP 2008, LNCS 5004, pp. 111–122, 2008.

descendant tasks is 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 [2]. Direct nesting of `section` blocks is allowed, but hierarchical synchronization of tasks can only be attained by nesting parallel regions. The Nanos group also proposed the `pred` and `succ` constructs to specify precedence relations among statically named `sections` in OpenMP [3]. [4] 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 [5] gives programmers a way to express patterns of concurrency that do not match the worksharing constructs defined in the current OpenMP 2.5 specification. 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 propose in 3.0, may still be too rigid too express all parallelism available in some applications, specially when the scalability to a high number of cores is the target.

```
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      lu0(A[kk][kk]);
11      /* fwd phase */
12      for (jj=kk+1; jj<NB; jj++)
13         if (A[kk][jj] != NULL)
14               fwd(A[kk][kk], A[kk][jj]);
15      /* bdiv and bmod phases */
16      for (ii=kk+1; ii<NB; ii++)
17         if (A[ii][kk] != NULL) {
18            bdiv (A[kk][kk], A[ii][kk]);
19            for (jj=kk+1; jj<NB; jj++)
20               if (A[kk][jj] != NULL)
21                 {
22                    if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
23                    bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
24                 }
25            }
26   }
27 }
```

Fig. 1. Main code of the sequential SparseLU kernel

To motivate the proposal we use one of the examples that was used to test the appropriateness and performance of the tasking proposal in OpenMP 3.0: the sparseLU kernel shown in Figure 1. This kernel computes an LU matrix

```
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       lu0(A[kk][kk]);
11#pragma omp parallel
12{
13       /* fwd phase */
14#pragma omp for schedule(dynamic, 1) nowait
15       for (jj=kk+1; jj<NB; jj++)
16          if (A[kk][jj] != NULL)
17             fwd(A[kk][kk], A[kk][jj]);
18
19       /* bdiv phase */
20#pragma omp for schedule(dynamic, 1)
21       for (ii=kk+1; ii<NB; ii++)
22          if (A[ii][kk] != NULL)
23             bdiv (A[kk][kk], A[ii][kk]);
24
25       /* bmod phase */
26#pragma omp for schedule(dynamic, 1) private(jj)
27       for (ii=kk+1; ii<NB; ii++)
28          if (A[ii][kk] != NULL)
29             for (jj=kk+1; jj<NB; jj++)
30                if (A[kk][jj] != NULL)
31                {
32                   if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
33                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
34                }
35   }
36}
37}
```

Fig. 2. Main code of the OpenMP 2.5 SparseLU kernel

factorization. The matrix is organized in blocks that may not be allocated. In this kernel, once `lu0` is computed (line 10), all instances of `fwd` and `bdiv` can be executed in parallel (lines 14 and 18, respectively). Each pair of instances `fwd` and `bdiv` allow the execution of an instance of `bmod` (line 23). 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 the current worksharing directives in 2.5 to partially exploit the parallelism available in the kernel, for example using `for` to distribute the work in the loops on lines 15, 21 and 27 or 29 in Figure 2. Due to the sparseness of the matrix, a lot of imbalance exists, forcing the programmer to use dynamic scheduling of the iterations to have good load balance. For the `bmod` phase we have two options: parallelize the outer (line 27) or the inner loop (line 29). If the outer loop is parallelized, the overhead is lower but the imbalance is greater. On the other hand, if the inner loop is parallelized the iterations are smaller which allows a dynamic schedule to have better balance but the overhead of the worksharing is much higher.

Notice that it has been necessary to apply loop distribution to isolate the loop that executes the multiple instances of function `bdiv`. The `nowait` clause in the loop in line 14 allows the exploitation of the parallelism that exist among

```
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 #pragma omp parallel
9    for (kk=0; kk<NB; kk++) {
10 #pragma omp single
11       lu0(A[kk][kk]);
12       /* fwd phase */
13 #pragma omp for nowait
14       for (jj=kk+1; jj<NB; jj++)
15          if (A[kk][jj] != NULL)
16 #pragma omp task firstprivate(kk, jj)
17                fwd(A[kk][kk], A[kk][jj]);
18       /* bdiv phase */
19 #pragma omp for
20       for (ii=kk+1; ii<NB; ii++)
21          if (A[ii][kk] != NULL)
22 #pragma omp task firstprivate(kk, ii)
23                bdiv (A[kk][kk], A[ii][kk]);
24
25       /* bmod phase */
26 #pragma omp for private(jj)
27       for (ii=kk+1; ii<NB; ii++)
28          if (A[ii][kk] != NULL)
29             for (jj=kk+1; jj<NB; jj++)
30                if (A[kk][jj] != NULL)
31 #pragma omp task firstprivate(kk, jj, ii)
32                {
33                   if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
34                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
35                }
36    }
37 }
```

Fig. 3. Main code of SparseLU with OpenMP 3.0 tasks

the instances of functions `fwd` and `bdiv`. The implicit barrier at the end of worksharing in line 20 forces the dependences of `fwd` and `bdiv` with `bmod`.

Using the task proposed in 3.0, the code restructuring is quite similar, as shown in Figure 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. The `nowait` clause in line 13 allows the parallel execution of `fwd` and `bdiv` instances. The implicit barriers at the end of loops in lines 19 and 16 force the dependences between pairs of `fwd`/`bdiv` with `bmod` inside a single `kk` iteration and viceversa across consecutive iterations of loop `kk`.

Figure 4 shows an execution trace obtained from an instrumented run of the kernel and visualized with Paraver [6]. The window represents time in horizontal axis and per-thread activity in the vertical axis (in this case, each color identifies the function that is being executed). The visualization corresponds to the end of a `kk` iteration and the beginning of the next `kk+1` iteration. Yellow lines represent thread creation and thread execution points (in the window only for `fwd` and `bdiv`).

As we pointed at the beginning of this section, there exists more parallelism in this kernel that can not be exploited with the current task definitions: parallelism that exists between tasks created in lines 17 (fwd) and 23 (bdiv) and tasks

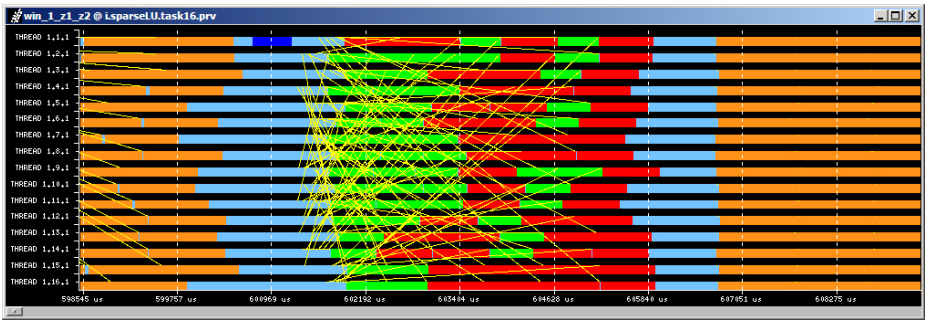

Fig. 4. Paraver window with a portion of SparseLU execution: lu0 (blue), fwd (green), bdiv (red) and bmod (orange) functions

created in line 34 (bmod) inside a single iteration . Also it would be interesting to express the parallelism that exists across consecutive iterations of the `kk` loop.

3 Proposed Extension

In this section we describe the extensions we propose to the OpenMP tasking model. We first describe them as part of the *StarSs* framework, a new programming paradigm for task-based programming that targets homogeneous symmetric multiprocessors (*SMPSs*) and the Cell/B.E. architecture [7] (*CellSs* [8]).

3.1 StarSs Pragmas and Execution Model

With *StarSs* the programmer identifies the functions that will be executed as tasks, using a pragma annotation right before the function definition. In addition the programmer specifies the directionality of each of the function parameters: input, output or input/output.

```
#pragma smpss task [clause[[,]clause] ...]
          {function-header|function-declaration}
```

where clauses can be:

- `input(argument-list)`
- `output(argument-list)`
- `inout(argument-list)`

Each element in `argument-list` is a block of contiguous memory locations whose number of elements is specified either in the function header or in the construct.

The following optional pragmas indicate a scope of the program where StarSs is used:

```
#pragma smpss start
#pragma smpss finish
```

When the `start` pragma is reached, the runtime initializes a worker thread in each processing element, who will wait for tasks to execute. Only a single thread (main thread) continues with the execution of the program, dynamically creating the tasks that are stored in a task graph. Both the main thread and the worker threads get tasks from the task graph once dependences are honored and execute the function associated. The `finish` pragma finishes all idle threads once the task graph is totally executed. Functions annotated with task have to be called between these two pragmas. If they are not present in the user code, the compiler will automatically insert the start pragma at the beginning of the application and the finish pragma at the end.

Figure 5 shows the SparseLU kernel programmed with the SMPSs extensions. The programmer identifies four tasks that correspond to the execution 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

```
1 #pragma smpss task input(diag[B][B]) inout(col[B][B])
2 void fwd(float *diag, float *col);
3
4 #pragma smpss task input(row[B][B],col[B][B]) inout(inner[B][B])
5 void bmod(float *row, float *col, float *inner);
6
7 #pragma smpss task input(diag[B][B]) inout(row[B][B])
8 void bdiv(float *diag, float *row);
9
10 #pragma smpss task inout(diag[B][B])
11 void lu0(float *diag);
12
13 int sparseLU() {
14    int ii, jj, kk;
15
16 #pragma smpss start
17    for (kk=0; kk<NB; kk++) {
18       lu0(A[kk][kk]);
19       /* fwd phase */
20       for (jj=kk+1; jj<NB; jj++)
21          if (A[kk][jj] != NULL)
22                fwd(A[kk][kk], A[kk][jj]);
23       /* bdiv and bmod phases */
24       for (ii=kk+1; ii<NB; ii++)
25          if (A[ii][kk] != NULL) {
26             bdiv (A[kk][kk], A[ii][kk]);
27             for (jj=kk+1; jj<NB; jj++)
28                if (A[kk][jj] != NULL)
29                  {
30                     if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
31                     bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
32                  }
33       }
34    }
35 #pragma smpss finish
36 }
```

Fig. 5. Main code of SparseLU with StarSs tasks

sequential version, with no transformations applied to allow the specification of the inherent parallelism available.

When a call to a function annotated with the `task` construct is found, the main thread creates a task for the associated function and adds information about data dependencies in a task graph. For each task, the runtime dynamically computes data dependencies by analyzing the direction (input, output or both), length and address of each parameter against those of previous tasks in sequential order. *True* data dependences (read-after-write) are honored by the runtime system by deferring the execution of the task until all `input` and `inout` arguments have been computed. The execution of the task can be done by any thread in the current parallel team. Once a task finishes its execution, the runtime updates the task graph to signal the modification of all `output` and `inout` arguments.

The runtime system automatically removes *false* dependencies (write-after-read and write-after-write) using memory renaming, a technique borrowed from the idea of register renaming in current out-of-order superscalar processors. For each variable that needs to be renamed, the runtime allocates temporary memory space for it. That is, if a task writes to an array, renaming can replace that array by a temporary one and redirect all following reads of that definition to the temporary array.

While the underlying runtime is capable of handling all inter-task related data dependencies, it cannot handle dependencies with the code executed by the master thread. To handle this, StarSs includes a data barrier:

```
#pragma smpss wait on (address-list)
```

At the `wait on` pragma, the master thread waits for all memory locations in the `address-list` to be updated. Once this happens, the main thread continues with the execution of the code.

3.2 StarSs and OpenMP

The StarSs pragmas and execution model fit well with the tasking definition in OpenMP 3.0

```
#pragma omp task [clause[[,]clause] ...]
        structured-block
```

In addition to the clauses supported in OpenMP 3.0:

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

our proposal is to include:

- `input(variable-list)`
- `output(variable-list)`
- `inout(variable-list)`

```
1 int sparseLU() {
2    int ii, jj, kk;
3
4 for (kk=0; kk<NB; kk++) {
5#pragma omp task inout(A[kk][kk])
6    lu0(A[kk][kk]);
7    for (jj=kk+1; jj<NB; jj++)?
8       if (A[kk][jj] != NULL)?
9#pragma omp task input(A[kk][kk]) inout(A[kk][jj])
10          fwd(A[kk][kk], A[kk][jj]);
11
12   for (ii=kk+1; ii<NB; ii++) {
13      if (A[ii][kk] != NULL)?
14#pragma omp task input(A[kk][kk]) inout(A[ii][kk])
15          bdiv (A[kk][kk], A[ii][kk]);
16          for (jj=kk+1; jj<NB; jj++)?
17             if (A[kk][jj] != NULL) {
18                if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
19#pragma omp task input(A[ii][kk], A[kk][jj]) inout(A[kk][kk])
20                bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
21             }
22      }
23 }
```

Fig. 6. Main code of SparseLU with the proposed dependent tasks, version 1

We also propose to include the

```
#pragma omp wait on (address-list)
```

in order to provide a more flexible version of `taskwait`.

The first difference wit StarSs is that our proposed clauses apply to an OpenMP task, which is a structured block of code and not a function declaration or definition. The main implication of this is that the `variable-list` does not indicate formal function arguments but variables used in the scope of the structured block of code. Figure 6 shows the SparseLU example with the proposed extension in OpenMP.

The second difference is that StarSs forces dependent tasks to be generated in sequential order (or at least in an order that guarantees that the source is generated before the target of the dependence). In addition, only the *main thread* can generate tasks for the *worker* threads. In OpenMP is it possible to have multiple task generators (by having `task` inside a worksharing or by nesting `task`). This needs to be considered in the implementation of the extensions in the prototype OpenMP implementation, but in any case, it is the programmer responsibility to ensure the appropriate order of task generation.

Clauses `Input`, `output` and `inout` provide additional information to the `shared` data clause. This information is used by the runtime to dynamically build and update the task graph and schedule tasks for execution as soon as all their input variables are generated. 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 another one). `Firstprivate` variables could also be affected with an `input` clause, meaning that the per-task private copy of the variable should be initialized with the value generated by another task (in its `output` clause) instead of the value at creation time.

```
1 int sparseLU() {
2    int ii, jj, kk;
3    int lu0done, fwddone[NB], bdivdone[NB], bmoddone[NB][NB];
4
5 for (kk=0; kk<NB; kk++) {
6 #pragma omp task input(bmoddone[kk][kk]) output(lu0done)
7    lu0(A[kk][kk]);
8    for (jj=kk+1; jj<NB; jj++)?
9       if (A[kk][jj] != NULL)?
10 #pragma omp task input(lu0done,bmoddone[kk][kk]) output(fwddone[jj])
11          fwd(A[kk][kk], A[kk][jj]);
12
13    for (ii=kk+1; ii<NB; ii++) {
14       if (A[ii][kk] != NULL)?
15 #pragma omp task input(lu0done,bmoddone[kk][kk]) output(bdivdone[ii])
16          bdiv (A[kk][kk], A[ii][kk]);
17          for (jj=kk+1; jj<NB; jj++)?
18             if (A[kk][jj] != NULL) {
19                if (A[ii][jj]==NULL) A[ii][jj]=allocate_clean_block();
20 #pragma omp task input(bdivdone[ii],fwddone[jj]) inout(bmoddone[kk][kk])
21                bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
22             }
23       }
24 }
```

Fig. 7. Main code of SparseLU with the proposed dependent tasks, version 2

Previous proposals based on providing a name to each `section` or `task` [3,4] can also be implemented using the proposed extensions in this paper, as shown in Figure 7. In this case, a dependence is encapsulated in a variable that should be declared by the programmer and used in an `output` clause (in the source task) and and in a `input` clause (in the target task). This synchronization variable can be subject of reuse and therefore, false dependences; the automatic renaming mechanism in the runtime avoids these false dependences and avoids its scalar (or vector) expansion.

4 Additional Runtime Features

The prototype task implementation for OpenMP 3.0 enqueues new created tasks in a team pool of tasks. Any thread of the team can access this pool a execute the tasks from there. Threads have also a local pool in which they place those tasks that have been suspended by them if those tasks are *tied tasks*. Other threads are not allowed to steal tasks from this pool. But the OpenMP specification allows for other forms of scheduling (with certain restrictions related to tied/untied tasks). For example, it would be possible to implement a work-first scheduler (like Cilk [9] does) where tasks are executed as soon as they are created and the parent task is suspended and stored in a per task pool of tasks. Dependence restrictions would need to be considered in this case. To avoid starvation (because all tasks go to the local pools) work-stealing is allowed.

In the implementation of SMPSs each thread has a local pool of ready tasks. The main thread is responsible of running the main program by going through the non task user code, analyzing the data dependencies and adding the tasks to the task graph. New tasks that have no input dependencies are added to the main thread task pool; any worker thread can steal from the pool of the main

thread. When the main thread stops task generation (because the task pool is full or he is waiting for tasks to finish) it also execute tasks from its own pool.

Worker threads look for ready tasks first in their own pool, then on the main thread pool and then on the other thread pools. When a thread finishes running a task, it puts all the task successors that have become ready into its task pool. While worker threads consume tasks from their pool in LIFO order, they steal them from other threads in FIFO order. That is, they consume the graph in a depth first order as long as they can can get ready tasks, and then steal tasks from other threads in a breadth first order when their task pools become empty.

The idea behind this design is that each thread will be executing tasks in a different region of the graph and have little interference with other threads as long as there are ready tasks in that region or there are unexplored zones in the graph. Otherwise they will steal work from other threads in a way that tries to minimize the effect on the cache locality of that thread.

5 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) [2]. For comparison purposes we also use the reference implementation [10] 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.

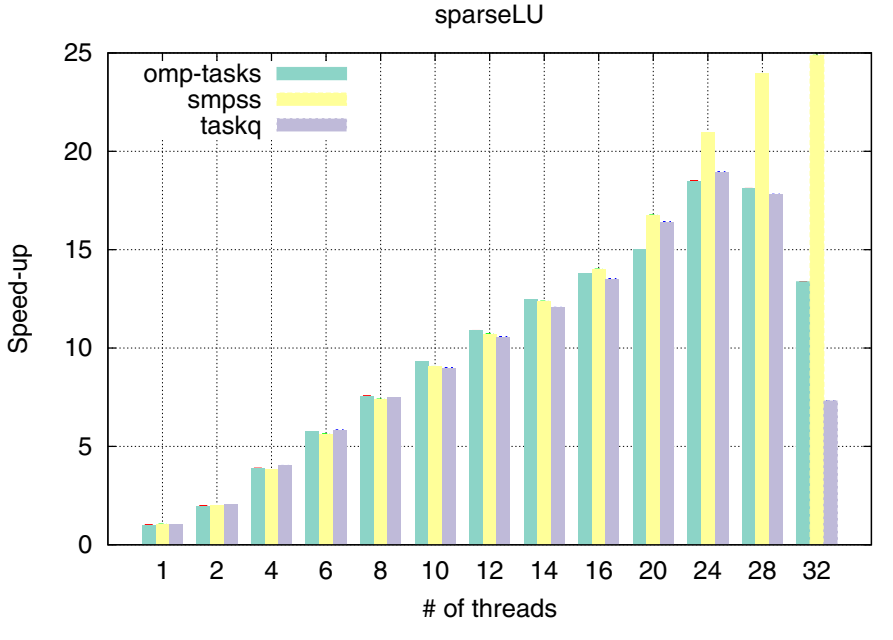

Fig. 8. Speed-up of *taskq*, *task* and *smpss* for SparseLU

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 8 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 *fwd*/*bdiv* and *bmod* phases (inside iteration kk) and between *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 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.

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 %

6 Conclusions

This paper proposed 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 used in a task. 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 (SMP Superscalar – SMPSs).

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 [7] (Cell Superscalar [8]). 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 IBM MareIncognito project, 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 Spanish Ministry of Education (contracts no. TIN2004-07739-C02-01 and TIN2007-60625).

References

1. Shah, S., Haab, G., Petersen, P., Throop, J.: Flexible control structures for parallellism in OpenMP. In: 1st European Workshop on OpenMP (September 1999)
2. 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)
3. Gonzàlez, M., Ayguadé, E., Martorell, X., Labarta, J.: Exploiting pipelined executions in OpenMP. In: 32nd Annual International Conference on Parallel Processing (ICPP 2003) (October 2003)
4. Sinnen, O., Pe, J., Kozlov, A.: Support for Fine Grained Dependent Tasks in OpenMP. In: 3rd International Workshop on OpenMP (IWOMP 2007) (2007)
5. 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 2007) (2007)
6. Labarta, J., Girona, S., Pillet, V., Cortes, T., Gregoris, L.: Dip: A parallel program development environment. In: 2nd International Euro-Par Conference on Parallel Processing (1996)
7. 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)
8. 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)
9. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: PLDI 1998: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, New York, pp. 212–223. ACM Press, New York (1998)
10. 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)