An Experimental Evaluation of the New OpenMP Tasking Model

Eduard Ayguadé[1], Alejandro Duran[1], Jay Hoeflinger[2], Federico Massaioli[3], Xavier Teruel[1]

[1] BSC-UPC
[2] Intel
[3] CASPUR

Abstract. The OpenMP standard was conceived to parallelize dense array-based applications, and it has achieved much success with that. Recently, a novel tasking proposal to handle unstructured parallelism in OpenMP has been submitted to the OpenMP 3.0 Language Committee. We tested its expressiveness and flexibility, using it to parallelize a number of examples from a variety of different application areas. Furthermore, we checked whether the model can be implemented efficiently, evaluating the performance of an experimental implementation of the tasking proposal on an SGI Altix 4700, and comparing it to the performance achieved with Intel's Workqueueing model and other worksharing alternatives currently available in OpenMP 2.5. We conclude that the new OpenMP tasks allow the expression of parallelism for a broad range of applications and that they will not hamper application performance.

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. However, the sophistication of parallel programmers has grown in the last 10 years since OpenMP was introduced, and the complexity of their applications is increasing. Therefore, OpenMP is in the process of adding a tasking model to address this new programming landscape. The new directives allow the user to identify units of independent work, leaving the decisions of how and when to execute them to the runtime system.

In this paper, we have attempted to evaluate this new tasking model. We wanted to know how the new tasking model compared to traditional OpenMP worksharing and the existing Intel workqueueing model, both in terms of expressivity and performance. In order to evaluate expressivity, we have parallelized a number of problems across a wide range of application domains, using the tasking proposal. Performance evaluation has been done on a prototype implementation

```
1  #pragma omp parallel private (p)
2  {
3    #pragma omp for
4    for (i=0; i< n_lists; i++) {
5      p = listheads[i];
6      while(p) {
7        #pragma omp task
8          process(p)
9        p=next(p);
10     }
11   }
12 }
```

Fig. 1. Parallel pointer chasing on multiple lists using `task`

```
1 void traverse(node *p, bool post)
2 {
3   if (p->left)
4     #pragma omp task
5       traverse(p->left, post);
6   if (p->right)
7     #pragma omp task
8       traverse(p->right, post);
9   if (post) { /* postorder! */
10    #pragma omp taskwait
11  }
12  process(p);
13 }
```

Fig. 2. Parallel depth-first tree traversal

of the tasking model. Performance results must be treated as preliminary, although we have validated the performance of our implementation against the performance of the commercial Intel workqueueing model implementation[1].

2 Motivation and related work

The task parallelism proposal under consideration by the OpenMP Language committee [2] 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 proposal addresses common operations like complex, possibly recursive, data structure traversal, and situations which could easily cause load imbalance. The efficient parallelization of these algorithms using the 2.5 OpenMP standard is not impossible, but requires extensive program changes, such as run-time data structure transformations. This implies significant hand coding and run-time overhead, reducing the productivity that is typical of OpenMP programming[3].

Figure 1 illustrates the use of the new `omp task`[1] construct from the proposal. It creates a new flow of execution, corresponding to the construct's structured block. This flow of execution is concurrent to the rest of the work in the parallel region, but its execution can be performed only by a thread from the current team. Notice that this behavior is different from that of worksharing constructs, which are cooperatively executed by the existing team of threads. Execution of the task region does not necessarily start immediately, but can be deferred until the runtime schedules it.

The `p` pointer variable used inside the tasks in Figure 1 is implicitly determined *firstprivate*, i.e. copy constructed at task creation from the original copies used by each thread to iterate through the lists. This default was adopted in the proposal to balance performance, safety of use, and convenience for the programmer. It can be altered using the standard OpenMP data scoping clauses.

[1] This paper will express all code in C/C++, but the tasking proposal includes the equivalent directives in Fortran.

The new `#pragma omp taskwait` construct used in Figure 1 suspends the current execution flow until all tasks it generated have been completed. The semantics of the existing `barrier` construct is extended to synchronize for completion of all generated tasks in the team.

For a programming language extension to be successful, it has to be useful, and must be checked for expressiveness and productivity. Are the directives able to describe explicit concurrency in the problem? Do data scoping rules, defaults and clauses match the real programmers' needs? Do common use cases exist that the extension does not fulfill, forcing the programmer to add lines of code to fill the gap? The two examples above, while illustrative, involve very basic algorithms. They cannot be considered representative of a real application kernel.

In principle, the more concurrency that can be expressed in the source code, the more the compiler is able to deliver parallelism. However, factors like subtle side effects of data scoping, or even missing features, could hamper the actual level of parallelism which can be achieved at run-time. Moreover, parallelism *per se* does not automatically imply good performance. The semantics of a directive or clause can have unforeseen impact on object code or runtime overheads. In a language extension process, this aspect should also be checked thoroughly, with respect to the existing standard and to competing models.

The suitability of the current OpenMP standard to express irregular forms of parallelism was already investigated in the fields of dense linear algebra [4, 5], adaptive mesh refinement [6], and agent-based models [7].

The Intel *workqueueing* model [8] was the first attempt to add dynamic task generation to OpenMP. The model, available as a proprietary extension in Intel compilers, allows hierarchical generation of tasks by the nesting of `taskq` constructs. Synchronization of descendant tasks is controlled by means of the default barrier at the end of `taskq` constructs. The implementation exhibits some overhead problems [7] and other performance issues [9].

In our choice of the application kernels to test drive the OpenMP tasking proposal, we were also inspired by the classification of different application domains proposed in [10], which addresses a much broader range of computations than traditional in the HPC field.

3 Programming with OpenMP Tasks

In this section we describe all the problems we have parallelized with the new task proposal. We have worked on applications across a wide range of domains (linear algebra, sparse algebra, servers, branch and bound, etc) to test the expressiveness of the proposal. Some of the applications (*multisort*, *fft* and *queens*) are originally from the Cilk project[11], some others (*pairwise alignment*, *connected components* and *floorplan*) come from the Application Kernel Matrix project from Cray[12] and two (*sparseLU* and *user interface*) have been developed by us. These kernels were not chosen because they were the best representatives of their class but because they represented a challenge for the current 2.5 OpenMP standard and were publicly available.

We have divided them into three categories. First were those applications that could already be easily parallelized with current OpenMP worksharing but where the use of tasks allows the expression of additional parallelism. Second were those applications which require the use of nested parallelism to be parallelized by the current standard. Nested parallelism is an optional feature and it is not always well supported. Third were those applications which would require a great amount of effort by the programmer to parallelize with OpenMP 2.5 (e.g. by programming their own tasks).

3.1 Worksharing versus tasking

SparseLU The sparseLU kernel computes an LU matrix factorization. The matrix is organized in blocks that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists. This is particularly true for the the *bmod phase* (see Figure 3). SparseLU can be parallelized with the current worksharing directives (using an OpenMP *for* with dynamic scheduling for loops on lines 10, 15 and 21 or 23). For the *bmod phase* we have two options: parallelize the outer (line 21) or the inner loop (line 23). 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.

Using tasks, first we 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.

It is interesting to note that, if the proposed extension included mechanisms to express dependencies among tasks, it would be possible to express additional parallelism that exists between tasks created in lines 12 and 17 and tasks created in line 25. Also it would be possible to express the parallelism that exists across consecutive iterations of the *kk* loop.

Protein pairwise alignment This application aligns all protein sequences from an input file against every other sequence. The alignments are scored and the best score for each pair is output as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. It uses the recursive Myers and Miller algorithm to align sequences.

The outermost loop can be parallelized, but the loop is heavily unbalanced, although this can be partially mitigated with dynamic scheduling. Another problem is that the number of iterations is too small to generate enough work when the number of threads is large. Also, the loops of the different passes (forward pass, reverse pass, diff and tracepath) can also be parallelized but this parallelization is much finer so it has higher overhead.

We used OpenMP tasks to exploit the inner loop in conjunction with the outer loop. Note that the tasks are nested inside an OpenMP *for* worksharing construct. This breaks iterations into smaller pieces, thus increasing the amount of parallel work but at lower cost than an inner loop parallelization because they can be excuted inmediately.

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

Fig. 3. Main code of SparseLU with OpenMP tasks

3.2 Nested parallelism versus tasking

Floorplan The Floorplan kernel computes the optimal floorplan distribution of a number of cells. The algorithm is a recursive branch and bound algorithm. The parallelization is straight forward (see figure 5). We hierarchically generate tasks for each branch of the solution space. But this parallelization has one caveat. In these kind of algorithms (and others as well) the programmer needs to copy the partial solution up to the moment to the new parallel branches (i.e. tasks). Due to the nature of C arrays and pointers, the size of it becomes unknown across function calls and the data scoping clauses are unable to perform a copy on their own. To ensure that the original state does not disappear before it is copied, a task barrier is added at the end of the function. Other possible solutions would be to copy the array into the parent task stack and then capture its value or allocate it in heap memory and free it at the end of the child task. In all these solutions, the programmer must take special care.

Multisort, FFT and Strassen Multisort is a variation of the ordinary mergesort. It sorts a random permutation of n 32-bit numbers with a fast parallel sorting algorithm by dividing an array of elements in half, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. When the array is too small, a serial quicksort is used so the task granularity is not too small. To avoid the

```
1 #pragma omp for
2 for (si = 0; si < nseqs; si++) {
3    len1 = compute_sequence_length(si+1);
4
5    /* compare to the other sequences */
6    for (sj = si + 1; sj < nseqs; sj++) {
7       #pragma omp task
8       {
9          len2 = compute_sequence_length(sj+1);
10         compute_score_penalties(...);
11         forward_pass(...);
12         reverse_pass(...);
13         diff(...);
14         mm_score = tracepath(...);
15         if (len1 == 0 || len2 == 0) mm_score  = 0.0;
16         else                        mm_score /= (double) MIN(len1,len2);
17
18         #pragma omp critical
19        print_score();
20      }
21   }
22 }
```

Fig. 4. Main code of the pairwise aligment with tasks

overhead of quicksort, an insertion sort is used for arrays below a threshold of 20 elements.

The parallelization with tasks is straight forward and makes use of a few `task` and `taskgroup` directives (see figure 6), the latter being the structured form of the `taskwait` construct introduced in section 2.

FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm. Strassen's algorithm for multiplication of large dense matrices uses hierarchical decomposition of a matrix. The structure of the parallelization of these two kernels is almost identical to the one used in multisort, so we will omit it.

N Queens problem This program, which uses a backtracking search algorithm, computes all solutions of the n-queens problem, whose objective is to find a placement for n queens on an n x n chessboard such that none of the queens attacks any other.

In this application, tasks are nested dynamically inside each other. As in the case of floorplan, the state needs to be copied into the newly created tasks so we need to introduce additional synchronizations (in the form of `taskgroup`) in order for the original state to be alive when the tasks start so they can copy it.

Another issue is the need to count all the solutions found by different tasks. One approach is to surround the accumulation with a critical directive but this would cause a lot of contention. To avoid it, we used `threadprivate` variables that are reduced within a `critical` directive to the global variable at the end of the parallel region.

Concom (Connected Components) The concom program finds all the connected components of a graph. It uses a depth first search starting from all the nodes of the graph. Every node visited is marked and not visited again.

```
1 void add_cell(int id, coor FOOTPRINT, ibrd BOARD, struct cell *CELLS) {
2   int  i, j, nn, area;  ibrd board;  coor footprint, NWS[DMAX];
3
4   for (i = 0; i < CELLS[id].n; i++) {
5       nn = compute_possible_locations(id, i, NWS, CELLS);
6 /* for all possible locations */
7       for (j = 0; j < nn; j++) {
8 #pragma omp task private(board, footprint,area) \
9         shared(FOOTPRINT,BOARD,CELLS)
10 {          /* copy parent state*/
11           struct cell cells[N+1];
12           memcpy(cells,CELLS,sizeof(struct cell)*(N+1));
13       memcpy(board, BOARD, sizeof(ibrd));
14
15       compute_cell_extent(cells,id,NWS,j);
16
17           /* if the cell cannot be layed down, prune search */
18           if (! lay_down(id, board, cells)) {
19              goto _end;
20           }
21       area = compute_new_footprint(footprint,FOOTPRINT,cells[id]);
22
23       /* if last cell */
24           if (cells[id].next == 0) {
25               if (area < MIN_AREA)
26               #pragma omp critical
27                   if (area < MIN_AREA)  save_best_solution();
28           } else if (area < MIN_AREA)
29       /* only continue if area is smaller to best area, otherwise prune */
30               add_cell(cells[id].next, footprint, board,cells);
31 _end:;
32 }
33     }
34   }
35   #pragma omp taskwait
36 }
```

Fig. 5. C code for the Floorplan kernel with OpenMP tasks

The parallelization with tasks involves just four directives: a parallel directive, a single directive, a task directive and a critical directive. This is a clear example of how well tasks map into tree-like traversals.

3.3 Almost impossible in OpenMP 2.5

Web server We used tasks to parallelize a small web server called Boa. In this application, there is a lot of parallelism, as each client request to the server can be processed in parallel with minimal synchronizations (only update of log files and statistical counters). The unstructured nature of the requests makes it very difficult to parallelize without using tasks.

On the other hand, obtaining a parallel version with tasks requires just a handful of directives, as shown in figure 8. Basically, each time a request is ready, a new task is created for it.

The important performance metric for this application is response time. In the proposed OpenMP tasking model, threads are allowed to switch from the current task to a different one. This task switching is needed to avoid starvation,

```
1 void sort(ELM *low, ELM *tmp, long size) {
2      if (size < quick_size) {
3       /* quicksort when reach size threshold */
4       quicksort(low, low + size - 1);
5       return;
6      }
7      quarter = size / 4;
8
9      A = low; tmpA = tmp;
10     B = A + quarter; tmpB = tmpA + quarter;
11     C = B + quarter; tmpC = tmpB + quarter;
12     D = C + quarter; tmpD = tmpC + quarter;
13
14     #pragma omp taskgroup {
15        #pragma omp task
16       sort(A, tmpA, quarter);
17        #pragma omp task
18       sort(B, tmpB, quarter);
19        #pragma omp task
20       sort(C, tmpC, quarter);
21        #pragma omp task
22       sort(D, tmpD, size - 3 * quarter);
23     }
24     #pragma omp taskgroup {
25        #pragma omp task
26       merge(A, A+quarter-1, B, B+quarter-1, tmpA);
27        #pragma omp task
28       merge(C, C+quarter-1, D, low+size-1, tmpC);
29     }
30     merge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
31 }
```

Fig. 6. Sort function using OpenMP tasks

and prevent overload of internal runtime data structures when the number of generated tasks overwhelms the number of threads in the current team. The implementation is allowed to insert implicit switching points in a task region, wherever it finds appropriate. The `taskyield` construct inserts an explicit switching point, giving programmers full control. The experimental implementation we used in our tests is not aggressive in inserting implicit switching points. To improve the performance of the Web server, we inserted a `taskyield` construct inside the serve_request function so that no request is starved.

User Interface We developed a small kernel that simulates the behavior of user interfaces (UI). In this application, the objective of using parallelism is to obtain a lower response time rather than higher performance (although, of course, higher performance never hurts). Our UI has three possible operations, which are common to most user interfaces: start some work unit, list current ongoing work units and their status, and cancel an existing work unit.

The work units map directly into tasks (as can be seen in Figure 9). The thread executing the `single` construct will keep executing it indefinitely. To be able to communicate between the interface and the work units, the programmer needs to add new data structures. We found it difficult to free these structures from within the task because it could easily lead to race conditions (e.g. free the structure while listing current work units). We decided to just mark them to be

```
1 void CC (int i, int cc) {
2     int j, n;
3     /* if node has not been visited */
4     if (!visited[i]) {
5         /* add node to current component */
6         add_to_component(i, cc); /* omp critical inside */
7
8         /* add each neighbor's subtree to the current component */
9         for (j = 0; j < nodes[i].n; j++) {
10        n = nodes[i].neighbor[j];
11        #pragma omp task
12            CC(n, cc);
13        }
14    }
15 }
16
17 void main () {
18     init_graph();
19     cc = 0;
20    /* for all nodes ... unvisited nodes start a new component */
21    for (i = 0; i < NN; i++)
22       if (!visited[i]) {
23     #pragma omp parallel
24         #pragma omp single
25             CC(i, cc);
26           cc++;
27       }
28
29 }
```

Fig. 7. Connected components code with OpenMP tasks

freed by the main thread when it knows that no tasks are using it. In practice, this might not always be possible and complex synchronizations may be needed.

We also used the `taskyield` directive to avoid starvation.

4 Evaluation

4.1 The prototype implementation

In order to test the proposal in terms of expressiveness and performance, we have developed our own implementation of the proposed tasking model. We developed the prototype on top of a research OpenMP compiler (source-to-source restructuring tool) and runtime infrastructure [13].

The implementation uses execution units, that are managed through different execution queues (usually one *global queue* and one *local queue* for each thread used by the application). The library offers different services (fork/join, synchronize, dependence control, environment queries, ...) that can provide the worksharing and structured parallelism expressed by the OpenMP 2.5 standard. We added several services to the library to give support to the task scheme. The most important change in the library was the offering of a new scope of execution that allows the execution of independent units of work that can be deferred, but still bound to the thread team (the concept of *task*, see section 2).

```
1 #pragma omp parallel
2 #pragma omp single nowait
3 while (!end) {
4     process signals (if any)
5     foreach request from the blocked queue {
6         if ( request dependences are met ) {
7             extract from the blocked queue
8             #pragma omp task
9                 serve_request(request);
10        }
11    }
12    if ( new connection ) {
13        accept_it();
14        #pragma omp task
15            serve_request(new connection);
16    }
17    select();
18 }
```

Fig. 8. Boa webserver main loop with OpenMp tasks

When the library finds a task directive, it is able to decide (according to internal parameters: *maximum depth level* in task hierarchy, *maximum number of tasks* or *maximum number of tasks by thread*) whether to execute it immediately or create a work unit that will be queued and managed through the runtime scheduler. This new feature is provided by adding a new set of queues: *team queues*. The scheduler algorithm is modified in order to look for new work in the *local*, *team* and *global* queues respectively.

Once the task is first executed by a thread, and if the task has *suspend/resume* points, we can expect two different behaviors. First, the task could be bound to that thread (so, it can only be executed by that thread) and second, the task is not attached to any thread and can be executed by any other thread of the team. The library offers the possibility to move a task from the *team* queues to the *local* queues. This ability covers the requirements of the `untied` clause of the `task` construct, which allows a task suspended by one thread to be resumed by a different one.

The synchronization construct is provided through *task counters* that keep track of the number of tasks which were created in the current scope (the current scope can be a `task` or `taskgroup` construct). Each task has in its own structure with a *successor* field that points to the counter it must decrement.

4.2 Evaluation methodology

We have already shown the flexibility of the new tasking proposal, but what about its performance? To determine this, we have evaluated the performance of the runtime prototype against other options.

We have run all the previous benchmarks but we do not include the results for the webserver (due to a lack of the proper network environment) and the simple-ui (because it has an interactive behavior). For each application we have tried each possible OpenMP version: a single level of parallelism (labeled OpenMP

```
1 void Work::exec ( ) {
2    while (!end) {
3       //do some amount of work
4       #pragma omp taskyield
5    }
6 }
7
8 void start_work (...) {
9    Work *work = new Work(...);
10   list_of_works.push_back(work);
11   #pragma omp task
12   {
13      work->exec();
14      work->die();
15   }
16   gc();
17 }
18
19 void ui () {
20    ...
21    if ( user_input == START_WORK ) start_work(...);
22 }
23
24 void main ( int argc, char **argv ) {
25    #pragma omp parallel
26    #pragma omp single nowait
27       ui();
28 }
```

Fig. 9. Simplified code for a user interface with OpenMP tasks

worksharing), multiple levels of parallelism (labeled OpenMP nested) and with OpenMP tasks. For those applications that could be parallelized with Intel's taskqueues, we also evaluated them with taskqueues.

Table 1 summarizes the different input parameters and the experiments run for each application.

We compiled the codes with taskqueues and nested parellelism with Intel's icc compiler version 9.1 at the default optimization level. The versions using tasks use our OpenMP source-to-source compiler and runtime prototype implementation, using icc as the backend compiler. The speedup of all versions is computed, using as a baseline the serial version of each kernel. We used Intel's icc compiler to compile the serial version.

All the benchmarks have been evaluated on an SGI Altix 4700 with 128 processors, although they were run on a cpuset comprising a subset of the machine to avoid interference with other running applications.

4.3 Results

In figure 10 we show the speedup for all the kernels (except the *concom*) with the different evaluated versions: OpenMP worksharing, OpenMP nested, OpenMP tasks and Intel's taskqueues. We do not show the results of the *concom* kernel because the slowdowns prevented us from running the experiments due to time constraints. These slowdowns were not only affecting the OpenMP task version

Application	Input parameters	Experiments
strassen	Matrix size of 1280x1280	nested, tasks, taskqueues
multisort	Array of 32M of integers	nested, tasks, taskqueues
fft	Array of 32M of complex numbers	nested, tasks, taskqueues
queens	Size of the board is 14x14.	nested, tasks, taskqueues
alignment	100 sequences	worksharing, nested, tasks
floorplan	20 cells	nested, tasks, taskqueues
concom	500000 graph nodes, 100000 edges	nested, tasks, taskqueues
sparseLU	Sparse matrix of 50 blocks of 100x100	worksharing, nested, tasks, taskqueues

Table 1. Input parameters for each application

but also the OpenMP nested and Intel's taskqueues. The main reason behind the slowdown is granularity. The tasks (or parallel regions in the nested case) are so fine grained that it is impossible to scale without aggregrating them. That is something that currently none of the models supports.

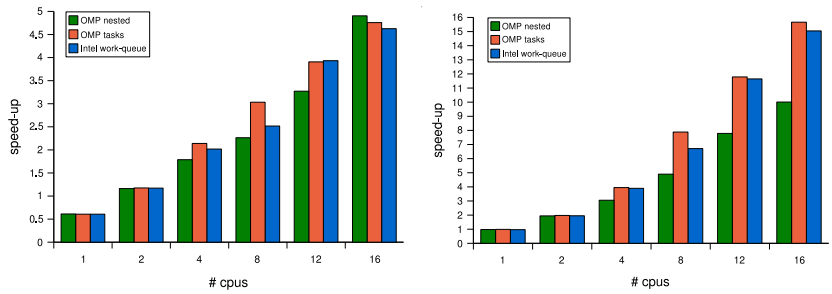
For a small number of threads (up to 4) we see that the versions using the new OpenMP tasks perform about the same as those using current OpenMP (worksharing and nested versions). But, as we increase the number of processors the task version scales much better, always improving over the other versions except for the *multisort* kernel, which has the same performance. These improvements are due to different factors, depending on the kernel: better load balance (*sparseLU*, *alignment*, *queens*, *fft*, *strassen* and *floorplan*), greater amount of parallel work (*alignment* and *sparseLU*) and less overhead (*alignment*). Overall, we can see that the new task proposal has the potential to benefit a wide range of application domains.

When we compare how the current prototype performs against a well established implementation of tasking, Intel's taskqueue, we can see that in most of the kernels the obtained speedup is almost the same and in a few cases (*sparseLU* and *floorplan*), even better. Only in two of them (*fft* and *strassen*) does taskqueue perform better, and even then, not by a large amount.

Taking into account that the prototype implementation has not been well tuned, we think that the results show that the new model will allow codes to obtain at least the performance of Intel's taskqueue and is even more flexible.

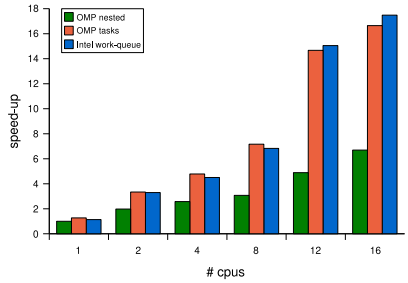
5 Suggestions for future work

While the performance and flexibility of the new OpenMP tasking model seem good, there is still room for improvement. We offer these suggestions for ways to improve the usability and performance of the model, based on our experience with the applications described in this paper.

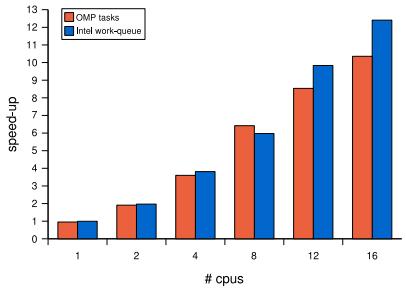


(a) Multisort evaluation

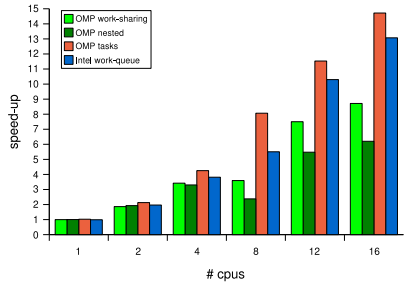
(b) N Queens evaluation



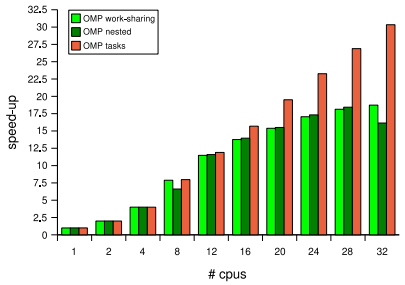
(c) FFT evaluation



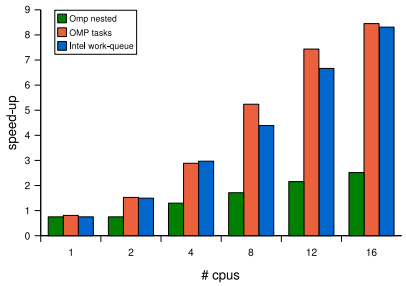
(d) Strassen evaluation



(e) SparseLU evaluation



(f) Alignment evaluation



(g) Floorplan evaluation

Fig. 10. Evaluation results for all the kernels. Speedups use serial version as baseline

One problem we encountered consistently in our programming was the need to capture the value of a data structure when all we had was a pointer to it. If a pointer is used in a `firstprivate` directive, only the pointer is captured. In order to capture the data structure pointed-at, the user must program it by hand inside the task, including proper synchronization, to make sure that the data is not freed or popped off the stack before it is copied. Support for this in the language would improve the usability of the tasking model.

In the N Queens problem, we could have used a reduction operation for tasks. In other words, we could have used a way to automatically make tasks contribute values to a shared variable. It can be programmed explicitly using threadprivate variables, but a reduction clause would save programming effort.

The `taskgroup` and `taskwait` constructions provide useful task synchronization, but are cumbersome for programming some types of applications, such as a multi-stage pipeline. A pipeline could be implemented by giving names to tasks, and waiting for other tasks by name.

We anticipate much research in the area of improving the runtime library. One research direction that would surely yield improvements is working on the task scheduler, as it can significantly affect application performance. Another interesting idea would be to find the impact of granularity on application performance and develop ways, either explicitly or implicitly, to increase the granularity of the tasks (for example by aggregating them) so they could be applied to applications with finer parallelism (e.g. the connected components problem) or reduce the overhead in other applications.

Of course, we have not explored all possible application domains, so other issues may remain to be found. Therefore, it is important to continue the assesment of the proposal by looking at new applications and particularly at Fortran codes, where optimizations could be affected differently by the tasking model. Another interesting dimension to assess in the future is the point of view of novice programmers and their learning curve with the model.

6 Conclusions

This paper had two objectives: first, test the expressiveness of the new OpenMP tasks proposal. Second, verify that the model does not introduce hidden factors that hamper the actual level of parallelism which can be achieved at runtime.

We have shown that the new proposal allows the programmer to express the parallelism of a wide range of applications from very different domains (linear algebra, server applications, backtracking, etc). Furthermore, we have found different issues that OpenMP language designers may want to consider in the future to further improve the expressiveness of the language and simplify the programming effort in some scenarios.

Using these applications we have seen the new proposal matches other tasking proposals in terms of performance and that it surpasses alternative implementations with the current 2.5 OpenMP elements. While these results are not conclusive, as they certainly have not explored exhaustively all possibilities, they

provide a strong indication that the model can be implemented without incurring significant overheads. We have also detected two areas where runtime improvements would benefit the applications (i.e. task scheduling and granularity).

In summary, we think that while the new OpenMP task proposal can be improved, it provides a solid basis for the development of applications containing irregular parallelism.

Acknowledgments

The Nanos group at BSC-UPC has been supported by the Ministry of Education of Spain under contract TIN2007-60625, and the European Commission in the context of the SARC integrated project #27648 (FP6).

References

1. Intel Corporation. *Intel(R) C++ Compiler Documentation*, May 2006.
2. E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, P. Unnikrishnan, and G. Zhang. A Proposal for Task Parallelism in OpenMP. In *3rd International Workshop on OpenMP (IWOMP'07)*, 2007.
3. L. Hochstein et al. Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers. In *SuperComputing '05*, November 2005.
4. S. Salvini. Unlocking the Power of OpenMP. Invited lecture at 5th European Workshop on OpenMP (EWOMP '03), September 2003.
5. J. Kurzak and J. Dongarra. Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead. LAPACK Working Note 178, Dept. of Computer Science, University of Tennessee, September 2006.
6. R. Blikberg and T. Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31(10-12):984–998, 2005.
7. F. Massaioli, F. Castiglione, and M. Bernaschi. OpenMP parallelization of agent-based models. *Parallel Computing*, 31(10-12):1066–1081, 2005.
8. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallellism in OpenMP. In *1st European Workshop on OpenMP*, September 1999.
9. F. G. Van Zee, P. Bientinesi, T. M. Low, and R. A. van de Geijn. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM Trans. Math. Soft., submitted*, 2006.
10. K. Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Science Depts., University of California at Berkeley, December 2006.
11. Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, New York, NY, USA, 1998. ACM Press.
12. B. Chamberlain, J. Feo, J. Lewis, and D. Mizell. An application kernel matrix for studying the productivity of parallel programming languages. In *W3S Workshop - 26th International Conference on Software Engineering*, pages 37–41, May 2004.
13. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, and E. Ayguadé a nd J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP 2004*, October 2004.