The Design of OpenMP Tasks

Eduard Ayguadé, Nawal Copty, *Member*, *IEEE Computer Society*, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, *Member*, *IEEE*, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang

Abstract—OpenMP has been very successful in exploiting structured parallelism in applications. With increasing application complexity, there is a growing need for addressing irregular parallelism in the presence of complicated control structures. This is evident in various efforts by the industry and research communities to provide a solution to this challenging problem. One of the primary goals of OpenMP 3.0 was to define a standard dialect to express and to exploit unstructured parallelism efficiently. This paper presents the design of the OpenMP tasking model by members of the OpenMP 3.0 tasking subcommittee which was formed for this purpose. This paper summarizes the efforts of the subcommittee (spanning over two years) in designing, evaluating, and seamlessly integrating the tasking model into the OpenMP specification. In this paper, we present the design goals and key features of the tasking model, including a rich set of examples and an in-depth discussion of the rationale behind various design choices. We compare a prototype implementation of the tasking model with existing models, and evaluate it on a wide range of applications. The comparison shows that the OpenMP tasking model provides expressiveness, flexibility, and huge potential for performance and scalability.

Index Terms—Parallel programming, OpenMP, task parallelism, irregular parallelism.

1 Introduction

In the last few decades, OpenMP has emerged as the de facto standard for shared-memory parallel programming. OpenMP provides a simple and flexible interface for developing portable and scalable parallel applications. OpenMP grew in the 1990s out of the need to standardize the different vendor specific directives related to parallelism. It was structured around parallel loops and was meant to handle dense numerical applications.

Modern applications are getting larger and more complex, and this trend will continue in the future. Irregular and dynamic structures, such as `while` loops and recursive routines are widely used in applications today. The set of features in the OpenMP 2.5 specification is ill equipped to exploit the concurrency available in such applications. Users now need a simple way to identify independent units of work and not concern themselves with scheduling these work units. This model is typically called "tasking" and has been embodied in a number of projects, such as Cilk [1]. Previous OpenMP-based extensions for tasking (for example, workqueueing [2] and dynamic sections [3]) have demonstrated the feasibility of providing such support in OpenMP.

With this in mind, a subcommittee of the OpenMP 3.0 language committee was formed in September 2005, with the goal of defining a simple tasking dialect for expressing irregular and unstructured parallelism. Representatives from Intel, UPC, IBM, Sun, CASPUR, and PGI formed the core of the subcommittee. Providing tasking support became the single largest and most significant feature targeted for the OpenMP 3.0 specification.

This paper presents the work of the OpenMP tasking subcommittee spanning over two years. Section 2 discusses the motivation behind our work and explores the limitations of the current OpenMP standard and existing tasking models. Section 3 describes the task model and presents the paradigm shift in the OpenMP view from thread-centric to task-centric. Section 4 discusses our primary goals, design principles, and the rationale for several design choices. In Section 5, we illustrate several examples that use the task model to express parallelism. Section 6 presents an evaluation of our model (using a prototype implementation) against existing tasking models. Section 7 explores future research directions and extensions to the model.

2 Motivation and Related Work

Many applications, ranging from document-based indexing to adaptive mesh refinement, have a lot of potential parallelism which is not regular in nature and which varies with the data being processed. Irregular parallelism in these applications is often expressed in the form of dynamically generated units of work that can be executed asynchronously. The OpenMP Specification Version 2.5, however, does not provide a natural way to express this type of irregular parallelism, since OpenMP was originally "somewhat tailored for large array-based applications" [4]. This is evident in the two main mechanisms for distributing work

- *E. Ayguadé, A. Duran, and X. Teruel are with the Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya, C/ Jordi Girona, 1-3 Campus Nord, Mod D6-210, E-08034 Barcelona, Spain, and also with the Barcelona Supercomputing Center, C/Jordi Girona, 29 Campus Nord, Edifici Nexus-II, E-08034 Barcelona, Spain. E-mail: {eduard, aduran}@ac.upc.edu, xavier.teruel@bsc.es.*
- *N. Copty and Y. Lin are with Sun Microsystems Inc., Mailstop UMPK15-239, 15 Network Circle, Menlo Park, CA 94025. E-mail: {nawal.copty, yuan.lin}@sun.com.*
- *J. Hoeflinger is with Intel, 1906 Fox Drive, Champaign, IL 61820. E-mail: jay.p.hoeflinger@intel.com.*
- *F. Massaioli is with CASPUR, Via dei Tizii 6/b, I-00185 Rome, Italy. E-mail: federico.massaioli@caspur.it.*
- *P. Unnikrishnan and G. Zhang are with IBM Toronto Software Lab, 8200 Warden Ave., Markham, ON L6G 1C7, Canada. E-mail: {priyau, guansong}@ca.ibm.com.*

Manuscript received 24 Jan. 2008; revised 4 June 2008; accepted 11 June 2008; published online 19 June 2008.
Recommended for acceptance by R. Bianchini.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-01-0031.
Digital Object Identifier no. 10.1109/TPDS.2008.105.

```
1  p = listhead; num_elements=0;
2  while (p) {
3    list_item[num_elements++]=p;
4    p=next(p);
5  }
6
7  #pragma omp parallel for
8    for (int i=0; i < num_elements; i++)
9      process(list_item[i]);
```

Fig. 1. Parallel pointer chasing with the *inspector-executor* model.

among threads in OpenMP. In the *loop* construct, the number of iterations is determined upon entry to the loop and cannot be changed during its execution. In the sections construct, the units of work (sections) are statically defined at compile time.

Fig. 1 shows an example of dynamic linked list traversal. First, a `while` loop is used to traverse a list and store pointers to the list elements in an array called `list_item`. Second, a `for` loop is used to iterate over the elements stored in the `list_item` array and call `process()` routine for each element. Since the iterations of the `for` loop are independent, OpenMP is used to parallelize the `for loop`, so that the iterations of the loop are distributed among a team of threads and executed in parallel.

A common operation like dynamic linked list traversal is therefore not readily parallelizable in OpenMP. One possible approach is to store pointers to the list elements in an array, as shown in Fig. 1. Once all the pointers are stored in the array, we can process the data in the array using a `parallel for` loop. The `parallel for` directive creates a team of threads and distributes the iterations of the associated `for` loop among the threads in the team. The threads execute their subsets of the iterations in parallel.

This approach of storing pointers to the list elements in an array incurs the overhead of array construction, which is not easy to parallelize.

Another approach is to use the `single nowait` construct inside a `parallel` region, as shown in Fig. 2. The `parallel` directive creates a team of threads. All the threads in the team execute the `while loop` in parallel, traversing all of the elements of the list. The `single` directive is used to ensure that only one of the threads in the team actually processes a given list element.

While elegant, this second approach is unintuitive and inefficient because of the relatively high cost of the `single` construct [5], and each thread needs to traverse the whole list and determine for each element whether another thread has already executed the work on that element.

The OpenMP Specification Version 2.5 also lacks the facility to specify structured dependencies among different

```
1 #pragma omp parallel private (p)
2 {
3   p = listhead;
4   while(p) {
5     #pragma omp single nowait
6       process(p);
7     p = next(p);
8   }
9 }
```

Fig. 2. Parallel pointer chasing using `single nowait`.

```
1 void traverse(binarytree *p) {
2   #pragma omp parallel sections num_threads(2)
3   {
4     #pragma omp section
5       if (p->left) traverse(p->left);
6     #pragma omp section
7       if (p->right) traverse(p->right);
8   }
9   process(p);
10 }
```

Fig. 3. Parallel depth-first tree traversal.

units of work. The `ordered` construct imposes a sequential ordering of execution. Other OpenMP synchronization constructs, like `barrier`, synchronize a whole team of threads, not work units. This is a serious limitation that affects the coding of hierarchical algorithms such as tree data structure traversal, multiblock grid solvers, adaptive mesh refinement [6], and dense linear algebra [7], [8], [9], to name a few. In principle, nested parallelism can be used to address this issue, as shown in the example in Fig. 3. The `parallel` directive in routine `traverse()` creates a team of two threads. The `sections` directive is used to specify that one of the threads should process the left subtree and the other thread should process the right subtree. Each of the threads will call `traverse()` recursively on its subtree, creating nested parallel regions. This approach can be costly, however, because of the overhead of parallel region creation, the risk of oversubscribing system resources, difficulties in load balancing, and different behaviors of different implementations. All of these issues make the nested parallelism approach impractical.

There have been several proposals for expressing irregular parallelism in programming languages. We list a few here.

Compositional C++ (CC++) [10] is an early extension of C++ designed for the development of task-parallel object-oriented programs. CC++ introduces the `par` block and the `parfor` and `spawn` statements. The `par` block executes each statement in the block in a separate task. The `parfor` statement executes each iteration of the following for loop in a separate task. The `spawn` statement executes an arbitrary CC++ expression in a new thread.

The Cilk programming language [1] is an elegant, simple, and effective extension of C for multithreading that is based on dynamic generation of tasks. Cilk is instructive, particularly because of the *work-first* principle and the *work-stealing* technique adopted. However, Cilk lacks several features, such as *loop* and `sections` constructs, that make OpenMP very efficient for solving many computational problems.

The Intel *work-queuing* model [2] is an attempt to add dynamic task generation to OpenMP. This proprietary extension to OpenMP allows the definition of tasks in the lexical extent of a `taskq` construct. Hierarchical generation of tasks can be accomplished by nesting `taskq` constructs. Synchronization of tasks is controlled by means of implicit barriers at the end of `taskq` constructs. The implementation, however, was shown to exhibit some performance issues [5], [8].

The Nanos group at UPC proposed *dynamic sections* as an extension to the OpenMP `sections` construct to allow

dynamic generation of tasks [3]. Direct nesting of `section` blocks is allowed, but hierarchical synchronization of tasks can only be accomplished by nesting parallel regions. The Nanos group also proposed the `pred` and `succ` constructs to specify precedence relations among statically named `sections` in OpenMP [11]. This is an extension that may be explored as part of our future work.

Intel Threading Building Blocks (TBB) [12] is a C++ runtime library without special compiler support or language extensions. It allows the user to program in terms of tasks (represented as instances of a task class). The runtime library takes full responsibility for scheduling the tasks for locality and load balancing. TBB's higher-level loop templates (for example, parallel reduction) are built upon the task scheduler and are responsible for dividing work into tasks. TBB also provides concurrent container classes that allow concurrent access of various containers (for example, hash maps and queues) by either fine-grained locking or lock-free algorithms.

The Task Parallel Library (TPL) developed by Microsoft [13] supports parallel constructs like `parallel for` by providing the `Parallel.For` method. TPL also supports other constructs such as `task` and `future`. A task is an action that can be executed concurrently with other tasks. A `future` is a specialized task that returns a result; the result is computed in a background thread encapsulated by the `future` object, and the result is buffered until it is retrieved.

Both TBB and TPL offer a task model similar to our proposal. But our model follows the incremental parallelization and sequential consistency principles that are part of the OpenMP philosophy (and of its success). As well, our proposal is not targeted to a specific language but works for all the different OpenMP base languages (C, C++ and Fortran).

The need to support irregular forms of parallelism in HPC is evident in the features being included in new programming languages, notably X10 (asynchronous activities and futures using `async` and `future`) [14], Chapel (the `cobegin` statement) [15], and Fortress (`tuple` expressions) [16].

Moreover, previous works [17], [18], [19], [20] have found that mixing data and task parallelism can improve the performance of many applications, although integrating both models can be quite challenging particularly in the thread-centric model of OpenMP [21].

Our tasking proposal aims to make OpenMP more suitable for expressing irregular parallelism and for parallelizing units of work that are dynamically generated. One observation is that, conceptually, OpenMP already has tasks, and every part of an OpenMP program is part of one task or another. Our proposal simply adds the ability to create explicitly defined tasks to OpenMP.

3 Task Proposal

OpenMP version 2.5 is based on threads. The execution model is based on the fork-join model of parallel execution where all threads have access to a shared memory. The `parallel` directive is used to create a team of threads. Worksharing directives (such as `for`, `sections`, and `single`) are used to distribute units of work among threads in the team. Each unit of work is assigned to a specific thread in the team and is executed from start to finish by that same thread. A thread may not suspend the execution of one unit of work to work on another.

```
1 #pragma omp task shared(tot), private(st), firstprivate(p)
2 {
3   st = process(p);
4   #pragma omp critical
5     tot += st;
6 }
```

Fig. 4. Task definition.

OpenMP version 3.0 shifts the focus to tasks. A `parallel` directive still starts a team of threads and distributes and executes the work in the same fashion as in 2.5, but we say that the threads are each executing an *implicit task* during the parallel region. Version 3.0 also introduces the `task` directive, which allows the programmer to specify a unit of parallel work called an *explicit task*. Explicit tasks are useful for expressing unstructured parallelism and for defining dynamically generated units of work, to be added to the work that will be done by the team. A task will be executed by one of the threads in the team, but different parts of a task may be executed by different threads, if the programmer so specifies.

3.1 The `task` Construct

The syntax for the new `task` construct[1] is illustrated in Fig. 4. Whenever a thread encounters a `task` construct, a new explicit task, i.e., a specific instance of executable code and its data environment, is generated from the associated structured block. An explicit task may be executed by any thread in the current team, in parallel with other tasks, and the execution can be immediate or deferred until later. The task a thread is currently executing is called its *current task*. Consistent with the established OpenMP terminology, all code encountered during execution of a task is termed a task region. Different encounters of the same `task` construct give rise to different tasks, whose execution corresponds to different task regions.

References within a task to a variable listed in the `shared` clause refer to the variable with that name known immediately prior to the `task` directive. New storage is created for each `private` and `firstprivate` variable, and all references to the original variable in the lexical extent of the task construct are replaced by references to the new storage. `firstprivate` variables are initialized with the value of the original variables at the moment of task generation, while `private` variables are not.

Data-sharing attributes of variables that are not listed in clauses of a `task` construct, and are not predetermined according to the usual OpenMP rules, are implicitly determined as follows: If a `task` construct is lexically enclosed in a `parallel` construct, variables that are shared in all scopes enclosing the `task` construct remain shared in the generated task. All other variables (even formal arguments of routines enclosing an orphaned `task` construct) are implicitly determined `firstprivate`. These

1. Fortran syntax is not shown in this paper because of space limitations.

```
1  void traverse(binarytree *p, int preorder) {
2    #pragma omp task if(!preorder)
3      process(p);
4    if (p->left) {
5      #pragma omp task
6        traverse(p->left, preorder);
7    }
8    if (p->right) {
9      #pragma omp task
10       traverse(p->right, preorder);
11   }
12 }
```

Fig. 5. Parallel, possibly preorder, tree traversal using tasks.

```
1  int postorder(binarytree *p) {
2    int l, r;
3    l = r = 0;
4    if (p->left) {
5      #pragma omp task shared(l)
6        l = postorder(p->left);
7    }
8    if (p->right) {
9      #pragma omp task shared(r)
10       r = postorder(p->right);
11   }
12   #pragma omp taskwait
13   return l + r + process(p);
14 }
```

Fig. 6. Postorder tree traversal using tasks.

default rules can be altered, specifying a `default` clause on the construct.

Worksharing regions cannot be closely nested, without an intervening parallel region. However, explicit tasks can be generated in a worksharing region. Moreover, `task` constructs can be lexically or dynamically nested, as illustrated in Fig. 5. A task is a *child* of the task that generated it. A child task region is not part of its generating task region. Nesting of tasks gives a new opportunity to an OpenMP programmer: sharing a variable that was private in the generating task (or in one of its ancestors). In this case, as the child task execution is concurrent with generating task execution, it is the programmer's responsibility to add proper synchronization to avoid data races, and to avoid allowing the shared variable to go out of existence if the parent task terminates before its child, as discussed later.

When an `if` clause is present on a task construct and the value of the scalar-expression evaluates to `false`, the encountering thread must suspend the current task region, and immediately execute the encountered task. The suspended task region will not be resumed until the encountered task is complete. The `if` clause does not affect descendant tasks. It gives opportunities to reduce generation overheads for too finely grained tasks, and allows users to express conditional dependencies as in Fig. 5.

3.2 Task Synchronization

All explicit tasks generated within a `parallel` region, in the code preceding an explicit or implicit barrier, are guaranteed to be complete on exit from that barrier region.

The `taskwait` construct can be used to synchronize the execution of tasks on a finer-grained basis, as illustrated in Fig. 6, where it enforces postorder traversal of the tree, and at the same time avoids shared variables going out of scope prematurely.

The `taskwait` construct suspends execution of the current task until all children tasks of the current task, generated since the beginning of the current task, are complete. Only child tasks are waited for, not their descendants.

Explicit or implicit barriers cannot be closely nested in explicit tasks. Implicit tasks (i.e., the execution by each thread in the team of the structured block associated with a `parallel` construct) are slightly different from explicit tasks in that they are allowed to execute closely nested barrier regions. They are guaranteed to be complete on exit from the implicit barrier at the end of the parallel region, but continue execution across other implicit or explicit barriers.

3.3 Task Execution

Once a thread in the current team starts execution of a task, the two become *tied* together: the same thread will execute the task region from beginning to end.

This does not imply that execution is continuous. A thread may suspend execution of a task region at a *task scheduling point*, to resume it at a later time. In tied tasks, task scheduling points may only occur at `task`, `taskwait`, explicit or implicit `barrier` constructs, and upon completion of the task. When a thread suspends the current task, it may perform a *task switch*, i.e., resume execution of a task it previously suspended, or start execution of a new task, under the Task Scheduling Constraint: *In order to start the execution of a new tied task, the new task must be a descendant of every suspended task tied to the same thread, unless the encountered task scheduling point corresponds to a barrier region.* The rationale for this constraint is discussed in the following section.

Most of the aforementioned restrictions are lifted for untied tasks (indicated by the `untied` clause on the `task` construct). Any thread in the team reaching a task scheduling point may resume any suspended untied task, or start any new untied task. Also, task scheduling points may in principle occur at any point in an untied task region.

Because parts of untied tasks may be executed by different threads, OpenMP 3.0 lock ownership is associated with tasks rather than threads.

4 Design Principles

Unlike the structured parallelism currently available in OpenMP, the tasking model is capable of exploiting irregular parallelism in the presence of complicated control structures. One of our primary goals was to design a model that is easy for a novice OpenMP user to use and one that provides a smooth transition for seasoned OpenMP programmers. We strived for the following as our main design principles: *simplicity of use*, *simplicity of specification*, and *consistency with the rest of OpenMP*, all without losing the expressiveness of the model. In this section, we outline some of the major decisions we faced and the rationale for our choices, based on available options, the trade-offs and our design goals.

```
1 #pragma omp task
2 {
3   for (i=0; i < huge_number; i++) {
4     #pragma omp task
5       foo();
6   }
7 }
```

Fig. 7. Simple code generating a large amount of tasks.

4.1 What Form Should the Tasking Construct(s) Take?

We considered two possibilities:

1. *A new worksharing construct pair*. It seemed like a natural extension of OpenMP to use a worksharing construct analogous to `sections` to set up the data environment for tasking and a `task` construct analogous to `section` to define a task. Under this scheme, tasks would be bound to the worksharing construct. However, these constructs would inherit all the restrictions applicable to worksharing constructs, such as a restriction against nesting them. Because of the dynamic nature of tasks, we felt that this would place unnecessary restrictions on the applicability of tasks and interfere with the basic goal of using tasks for irregular computations.
2. *A new OpenMP construct*. The other option was to define a single task construct that could be placed anywhere in the program and that would cause a task to be generated each time a thread encounters it. Tasks would not be bound to any specific OpenMP constructs. This makes tasking a very powerful tool and opens up new parallel application areas, previously unavailable to the user due to language limitations. Also, using a single tasking construct significantly reduces the complexity of construct nesting rules. The flexibility of this option seemed to make it easier to merge into the rest of OpenMP, so this was our choice.

4.2 Where Can Task Scheduling Points Be?

OpenMP has always been thread-centric. Threads provided a very useful abstraction of processors, and people have taken great advantage of this. OpenMP 3.0 provides another abstraction with the move toward tasks, and sometimes these abstractions conflict, so the OpenMP 3.0 committee wrestled with the implications of this, to find the best design to make tasking coexist in a natural way with legacy OpenMP codes.

An early decision we made was not to mandate that implementations execute a task from beginning to end. We wanted to give implementations more flexibility. *Task scheduling points* offer flexibility in scheduling the execution of a tasking program. When a thread encounters a *task scheduling point*, a decision can be made to suspend the current task and schedule the thread on a different task.

For example, in the code from Fig. 7, the outer task generates a large number of inner tasks. If the outer task could not be preempted, then an implementation might need to keep track of a large number of generated tasks, which may not be practical. On the other hand, if a task directive includes a task scheduling point, then when the structures holding generated tasks fill up, it becomes possible to suspend the generating task and allow the thread to execute some of the generated tasks, until there is room to generate tasks again and the original task is resumed. This is the flexibility provided by *task switching*.

But task switching can lead to load imbalance. Suppose for the code above that the same situation occurs—the generating task is suspended and the thread begins executing one of the generated tasks. If the tasks differ greatly in runtime, then it is possible that the thread starts executing a task that is extremely time consuming, and meanwhile all other threads finish executing all the other generated tasks. If the generating task is *tied*, then the other threads will have to remain idle until the original thread finishes its lengthy task and resumes generating tasks for the other threads to execute.

A way to deal with the load imbalance is to make the generating task *untied*. In this case, any thread may resume the generating task, allowing the other threads to do useful work even when the original generating thread gets stuck in a lengthy task as described above.

A very important thing to notice is that the value of a threadprivate variable,[2] or thread-specific information like the thread number, may change across a task scheduling point. If the task is untied, then the resuming thread may be different from the suspending thread; therefore, both the thread numbers and the threadprivate variables used on either side of the task scheduling point may differ. If the task is tied, then the thread number would remain the same, but the value of a threadprivate variable may change because the thread may switch at the task scheduling point to another task that modifies the threadprivate variable.

But, do people use thread-specific features in real codes? Unfortunately, yes. Threadprivate storage, thread-specific features, and thread-local storage provided by the native threading package or the linker are all useful for making library functions thread-safe. We wanted to make it possible to continue using thread-specific information in OpenMP 3.0, so we needed to provide a way to use that thread-specific information predictably. For these reasons, we decided to specify exactly where task scheduling points will occur in tied tasks. This makes it predictable where thread-specific information may change (`task` and `taskwait` directives, implicit and explicit barriers).

For untied tasks, we wanted to give implementations as much flexibility as possible. For an untied task region, task scheduling points may occur anywhere in the region, and the programmer cannot rely on two implementations defining them at the same places. Therefore, the use of threadprivate variables or anything dependent on thread ID is **strongly** discouraged in an untied task.

4.3 How Do Locks and Critical Sections Relate to Tasking?

OpenMP 2.5 provides mechanisms for mutual exclusion, namely critical sections and OpenMP locks, used in many codes and libraries. Moreover, many libraries and runtimes also resort to non-OpenMP locks for performance or other reasons in critical parts of the code. Because of task switching, and the fundamentally asynchronous way in which tasks can be scheduled, mutual exclusion among threads can lead to unintended deadlocks.

2. A threadprivate variable is a global variable which is replicated in a private storage area for each thread.

```
1 for (i=0; i < 10*omp_get_num_threads(); i++) {
2   #pragma omp task  // outer tasks
3   {
4     foo();
5     #pragma omp critical
6     {
7       bar();
8       #pragma omp task  // inner tasks
9       foobar();
10    }
11  }
12 }
```

Fig. 8. Simple code with a critical section and nested tasks.

Consider the code in Fig. 8. Imagine that a thread executing one of the outer tasks reaches the inner task construct. At the associated task scheduling point, the thread can legally switch to a different task. If the thread switches to one of the other outer tasks, it will eventually reach the `critical` section again, but this time will not be able to enter (because it is already inside the `critical` as another task!), and will wait there forever. All threads will eventually have to wait at the `critical` and the code will hang.

It would be a natural choice to switch from thread-based mutual exclusion to task-based mutual exclusion, and add task scheduling points at the entrance of critical regions and in OpenMP lock acquire routines. However, this would not address the issue with non-OpenMP mutex mechanisms employed by existing libraries. Moreover, we felt that the risk of breaking subtle assumptions made in existing, OpenMP parallelized libraries was too high. Eventually, we adopted a split decision.

Since the `critical` construct's structured block makes its usage lexically structured, we decided to leave the `critical` construct as a thread-based mutual exclusion mechanism, and added the Task Scheduling Constraint described in Section 3.3. The combination of these ensures that if a parallel program with `task` directives disabled does not deadlock, then enabling the task directives will not deadlock either.

Once again, untied tasks are treated more liberally: they are not subject to scheduling restrictions of any sort. Since task scheduling points can occur anywhere in an untied task (even inside a critical region), the usage of `critical` constructs in an untied task is discouraged.

On the other hand, usage of OpenMP locks is much less structured than that of critical regions, and acquisition and release of the same lock frequently takes place in separate lexical contexts. We decided that once a lock is acquired, the current task owns it, and the same task must release it before task completion. Programmers should be very careful about using locks in untied tasks.

An interesting byproduct of the change of lock ownership from threads to tasks results from a gray area in the previous OpenMP specs: when a thread executing in an original parallel region encounters a parallel directive, its thread number changes from whatever it was in its original team to "0" in the new team—does this make it a "new" thread in the new team? Or is it the same thread, just renumbered? If you take the point of view that it is the same thread, and combine that with the rule that the same thread that acquired the lock must also release it, then it would follow that the thread could acquire a lock outside the parallel region and release it inside the parallel region.

The 3.0 spec clarifies this situation. A thread begins executing a new implicit task in the new parallel region, so it is not allowed to acquire a lock in the original parallel region and release it in the new parallel region, since they are different tasks and the task that acquires a lock must also release it.

4.4 Should the Implementation Guarantee that Task References to Stack Data Are Safe?

A `task` is likely to have references to the data on the stack of the routine where the `task` construct appears. Since the execution of a task is not required to be finished until the next associated task barrier, it is possible that a given task will not execute until after the stack of the routine where it appears is already popped and the stack data overwritten, destroying local data listed as `shared` by the task.

The committee's original decision on this issue was to require the implementation to guarantee stack safety by inserting task barriers where required. We soon realized that there are circumstances where it is impossible to determine at compile time exactly when execution will leave a given routine. This could be due to a complex branching structure in the code, but worse would be the use of `setjmp/longjmp`, C++ exceptions, or even vendor-specific routines that unwind the stack. When you add to this the problem of the compiler understanding when a given pointer dereference is referring to the stack (even through a pointer argument to the routine), you find that in a significant number of cases the implementation would conservatively be forced to insert a task barrier immediately after many task constructs, unnecessarily restricting the parallelism possible with tasks.

Our final decision was simply to state that it is the user's responsibility to insert task barriers when necessary to ensure that variables are not deallocated before the task is finished using them.

4.5 What Should Be the Defaults for the Data-Sharing Attribute Clauses of Tasks?

OpenMP data-sharing attributes for variables can be predetermined, implicitly determined or explicitly determined. Variables in a task that have predetermined sharing attributes are not allowed in clauses (except for loop indices), and explicitly determined variables do not need defaults, by definition. However, determining data-sharing attributes for implicitly determined variables requires defaults.

The sharing attributes of a variable are strongly linked to the way in which it is used. If a variable is shared among a thread team and a task must modify its value, then the variable should be `shared` on the task construct and care must be taken to make sure that fetches of the variable outside the task wait for the value to be written. If the variable is read-only in the task, then the safest thing would be to make the variable `firstprivate`, to ensure that it is not deallocated before its use. Since we decided not to guarantee stack safety for tasks, we faced a hard choice. We could

1. make data primarily `shared`, analogous to using `shared` in the rest of OpenMP, or
2. make data primarily `firstprivate`.

The first choice is consistent with existing OpenMP. However, the danger of data going out of scope before being used in a task is very high with this default. This would put a heavy burden on the user to ensure that all

```
1 #pragma omp parallel
2 {
3   /* a single thread traverses the list */
4   #pragma omp single
5   {
6     p = listhead;
7     while(p) {
8       /* create a task for each element */
9       #pragma omp task
10        process(p)
11      p=next(p);
12    }
13  }
14 }
```

Fig. 9. Parallel pointer chasing using `task`.

the data remains allocated while it is used in the task. Debugging can be a nightmare for things that are sometimes deallocated prematurely. The biggest advantage of the second choice is that it minimizes the "data-deallocation" problem. The user only needs to worry about maintaining allocation of variables that are explicitly `shared`. The downside to using `firstprivate` as the default is that Fortran parameters and C++ reference parameters will, by default, be `firstprivate` in tasks. This could lead to errors when a task writes into reference parameters.

In the end, we decided to make all variables with implicitly determined sharing attributes default to `firstprivate`, with one exception: when a `task` construct is lexically enclosed in a `parallel` construct, variables that are shared in all nested scopes separating the two constructs, are implicitly determined shared. While not perfect, this choice gives programmers the most safety, while not being overly complex, and not forcing users to add long lists of variables in a `shared` clause.

5 Examples of Use

In this section, we use some examples to illustrate how tasks enable new parallelization strategies in OpenMP programming. Most code excerpts are part of the benchmarks that are later used in Section 6 to evaluate tasking with the reference implementation. We also revisit the two examples we used in Section 2.

In order to organize the presentation of the examples, we divide them into three subgroups. First, we describe situations showing how tasking allows one to express more parallelism (or to exploit it more efficiently) than current OpenMP worksharing constructs. Second, we describe situations in which tasking replaces the use of nested parallelism. Finally, we describe situations that impose a great amount of effort by the programmer to parallelize with OpenMP 2.5 (e.g., by programming their own tasks).

5.1 Worksharing versus Tasking

In this section, we illustrate some examples where the use of the new OpenMP tasks allows the programmer to express more parallelism (and thus obtain better performance) than could be expressed with OpenMP 2.5 worksharing constructs.

Pointer chasing. One of the simplest cases that motivated tasking in OpenMP was pointer chasing (or *pointer following*). As shown in Figs. 1 and 2, the execution in parallel of work units that are based on the traversal of a list

```
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             /* only create tasks for non-empty blocks */
11             #pragma omp task
12               fwd(A[kk][kk], A[kk][jj]);
13         /* bdiv phase */
14         for (ii=kk+1; ii < NB; ii++)
15           if (A[ii][kk] != NULL)
16             /* only create tasks for non-empty blocks */
17             #pragma omp task
18               bdiv (A[kk][kk], A[ii][kk]);
19         /* wait for previous tasks */
20         #pragma omp taskwait
21         /* bmod phase */
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                 /* only create tasks for non-empty blocks */
27                 #pragma omp task
28                 {
29                   if (A[ii][jj]==NULL)
30                     A[ii][jj]=allocate_clean_block();
31                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
32                 }
33         /* wait for all previous tasks */
34         #pragma omp taskwait
35       }
36 }
```

Fig. 10. Main code of SparseLU with OpenMP tasks.

(of unknown size) of data items linked by pointers can be done using worksharing constructs (`for` and `single`, respectively). But, they require either transforming the list into an array that is suitable for the traversal or all threads to go through each of the elements and compete to execute them. Both approaches are highly inefficient.

All these problems go away with the new task proposal. The pointer chasing problem could be parallelized as shown in Fig. 9, where the `single` construct ensures that only one thread will traverse the list and encounter the `task` directive.

The `task` construct gives more freedom for scheduling (as described in the following paragraphs).

Dynamic work generation and load balancing. The `for` worksharing construct is able to handle load imbalance situations by using dynamic scheduling strategies. Tasking is an alternative option to parallelize this kind of loop, as shown in the code excerpt in Fig. 10. In this code, the *if* statements that control the execution of functions *fwd*, *bdiv*, and *bmod* for nonempty matrix blocks are the sources of load imbalance. One could use an OpenMP `for` worksharing construct with `dynamic` scheduling for the loops on lines 9, 14, and 21 and 23 (for the *bmod* phase one can either parallelize the outer, line 21, or the inner loop, line 23, with different load balance versus overhead trade-offs). Using tasks, a single thread could create work for all those nonempty matrix blocks, achieving both load balance and low overhead in the generation and assignment of work.

It is interesting to note that, if the proposed extension included mechanisms to express point-to-point dependencies among tasks, it would be possible to express additional parallelism that exists between tasks created in lines 11 and 16 and tasks created in line 25. Also, it would be possible to

```
#pragma omp parallel private (p)
{
  /* lists are distributed between all threads */
  #pragma omp for
    for (int i=0; i < num_lists; i++) {
      p = listheads[i];
      while(p) {
        /* create a task for each element */
        #pragma omp task
          process(p)
        p=next(p);
      }
    }
}
```

Fig. 11. Parallel pointer chasing on multiple lists using `task`.

express the parallelism that exists across consecutive iterations of the *kk* loop. Instead, the taskwait reduces parallelism to ensure those dependences are not violated.

Combined worksharing and tasking. Current `for` and `sections` worksharing constructs can be used to have multiple task generators running in parallel. For example, the code in Fig. 11 is processing, in parallel, elements from multiple lists. This results in better load balancing when the number of lists does not match the number of threads, or when the lists have very different lengths.

Another example of combined use of worksharing constructs and tasking is shown in Fig. 12. In this code excerpt, using only worksharing constructs, 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.

OpenMP tasks can efficiently exploit the parallelism available in the inner loop in conjunction with the parallelism available in the outer loop, which uses a `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 executed immediately.

```
/* all threads pick up some sequences */
#pragma omp for
  for (si = 0; si < nseqs; si++) {
    len1 = compute_sequence_length(si+1);
    /* compare to the other sequences */
    for (sj = si + 1; sj < nseqs; sj++) {
      /* create a task for each comparison */
      #pragma omp task
      {
        len2 = compute_sequence_length(sj+1);
        compute_score_penalties(...);
        forward_pass(...);
        reverse_pass(...);
        diff(...);
        mm_score = tracepath(...);
        if (len1 == 0 || len2 == 0) mm_score  = 0.0;
        else mm_score /= (double) MIN(len1,len2);
          /* printing in mutual exclusion */
        #pragma omp critical
          print_score();
      }
    }
  }
```

Fig. 12. Main code of the pairwise alignment with tasks.

```
void traverse(binarytree *p, bool postorder) {
  /* create task for left branch */
  #pragma omp task
    if (p->left) traverse(p->left, postorder);
  /* create task for right branch */
  #pragma omp task
    if (p->right) traverse(p->right, postorder);
  if (postorder) {
    /* wait for child tasks to finish before processing */
    #pragma omp taskwait
  }
  process(p);
}
```

Fig. 13. Parallel depth-first tree traversal.

5.2 Nested Parallelism versus Tasking

In this section, we illustrate some examples where the use of the new OpenMP tasks allows a programmer to express parallelism that in OpenMP 2.5 would be expressed using nested parallelism. As we have discussed in Section 2, the versions using nested OpenMP, while simple to write, usually do not perform well because of a variety of problems (load imbalance, synchronization overheads, ...).

Handling recursive code structures. Another simple case that motivated tasking in OpenMP was recursive work generation, as shown in Fig. 3. Nested parallelism can be used to allow recursive work generation but at the expense of the overhead in creating a rigid tree structure of thread teams and their associated (unnecessary) implicit barriers. That code example could be rewritten as shown in Fig. 13. In this figure, we use `task` to avoid the nested `parallel` regions. Also, we can use a flag to make the postorder processing optional. Notice that a task can create new tasks inside the same team of threads.

Another example is shown in Fig. 14, in this case for multisort (a variation of the ordinary mergesort). The parallelization with tasks is straightforward and makes use of a few `task` and `taskwait` directives.

```
void sort(ELM *low, ELM *tmp, long size) {
  if (size < quick_size) {
    /* quicksort when reach size threshold */
    quicksort(low, low + size - 1);
    return;
  }
  /* split into 4 pieces: A, B, C and D */
  quarter = size / 4;
  A = low; tmpA = tmp;
  B = A + quarter; tmpB = tmpA + quarter;
  C = B + quarter; tmpC = tmpB + quarter;
  D = C + quarter; tmpD = tmpC + quarter;
  /* create tasks to sort vector splits A, B, C and D */
  #pragma omp task
    sort(A, tmpA, quarter);
  #pragma omp task
    sort(B, tmpB, quarter);
  #pragma omp task
    sort(C, tmpC, quarter);
  #pragma omp task
    sort(D, tmpD, size - 3 * quarter);
  /* wait for all sort tasks to finish */
  #pragma omp taskwait
  /* create tasks to merge A with B and C with D */
  #pragma omp task
    merge(A, A+quarter-1, B, B+quarter-1, tmpA);
  #pragma omp task
    merge(C, C+quarter-1, D, low+size-1, tmpC);
  /* wait for AB and CD merge to finish */
  #pragma omp taskwait
  /* merge AB with CD */
  merge(tmpA, tmpC-1, tmpC, tmpA+size-1, A);
}
```

Fig. 14. Sort function using OpenMP tasks.

```
1  void add_cell(int id, coor FOOTPRINT, ibrd BOARD,
2          struct cell *CELLS) {
3    int i, j, nn, area;
4    ibrd board;
5    coor footprint, NWS[DMAX];
6
7    for (i = 0; i < CELLS[id].n; i++) {
8      nn = compute_possible_locations(id, i, NWS, CELLS);
9      /* for all possible locations */
10     for (j = 0; j < nn; j++) {
11       /* create a task for each possible configuration */
12       #pragma omp task private(board, footprint,area) \
13         shared(FOOTPRINT,BOARD,CELLS)
14       {
15         /* copy parent state*/
16         struct cell cells[N+1];
17         memcpy(cells,CELLS,sizeof(struct cell)*(N+1));
18         memcpy(board, BOARD, sizeof(ibrd));
19         compute_cell_extent(cells,id,NWS,j);
20         /* if cell cannot be layed down, prune search */
21         if (! lay_down(id, board, cells)) goto _end;
22         area = compute_new_footprint(footprint,FOOTPRINT,
23                         cells[id]);
24         /* if last cell */
25         if (cells[id].next == 0) {
26           if (area < MIN_AREA)
27             #pragma omp critical
28               if (area < MIN_AREA)
29                 save_best_solution();
30         } else if (area < MIN_AREA)
31           /* only continue if area is smaller to best area,
32              otherwise prune */
33           add_cell(cells[id].next, footprint, board,cells);
34       _end:;
35       }
36     }
37   }
38   /* This taskwait ensures parent state remains alive
39      for child's to copy it*/
40   #pragma omp taskwait
41 }
```

Fig. 15. Floorplan kernel with OpenMP tasks.

Handling data copying. Fig. 15 shows the excerpt of a recursive branch and bound kernel. In this parallel version, we hierarchically generate tasks for each branch of the solution space. But this parallelization has one caveat: 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-sharing 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.

5.3 Almost Impossible in OpenMP 2.5

In this section, we illustrate two situations where OpenMP 2.5 would require from the programmer a high effort in parallelizing the code. We show that tasks naturally reduce the parallelization effort to a minimum.

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 Fig. 16. Basically, each time a request is ready, a new task is created for it.

```
1  #pragma omp parallel
2    /* a single thread manages the connections */
3    #pragma omp single nowait
4    while (!end) {
5      process signals (if any)
6      foreach request from the blocked queue {
7        if ( request dependences are met ) {
8          extract from the blocked queue
9          /* create a task for the request*/
10         #pragma omp task untied
11             serve_request(request);
12       }
13     }
14     if ( new connection ) {
15       accept_it();
16       /* create a task for the request*/
17       #pragma omp task untied
18             serve_request(new connection);
19     }
20     select();
21   }
```

Fig. 16. Boa webserver main loop with OpenMP tasks.

The important performance metric for this application is response time. In the proposed OpenMP tasking model, threads can switch from the current task to a different one. This task switching is needed to avoid starvation, and prevent overload of internal runtime data structures when the number of generated tasks overwhelms the number of threads in the current team.

User interface (UI). We developed a small kernel that simulates the behavior of UIs. 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 UIs: 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 Fig. 17). The thread executing the `single` construct will keep executing it indefinitely. To be able to communicate

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

Fig. 17. Simplified code for a UI with OpenMP tasks.

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

6 Evaluation

6.1 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 [22]. We developed the prototype on top of a research OpenMP compiler (source-to-source restructuring tool) and runtime infrastructure [23].

The runtime infrastructure is an implementation of a user-level thread package based on the nano-threads programming model introduced first by Polychronopoulos [24]. The implementation uses execution units, called nano-threads that are managed through different execution queues (usually one *global queue* for all threads and one *local queue* for each thread used by the application). Then, a nano-thread on the global queue can be executed by any thread but a nano-thread in a local queue can only be executed by the related thread.

The nano-thread layer is implemented on top of *POSIX Threads* (also know as *pthreads*). We decided to use *pthreads* to ensure that they will be portable across a wide range of systems.

This layered implementation can have a slight impact on efficiency. However, by using user-level threads, the runtime can manage the scheduling to decide when a nano-thread is executed and on which processor. Furthermore, the need to support *thread switching* for the new tasks requires this level of flexibility.

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).

When the library finds a task directive, it can execute it immediately or create a work unit that will be queued and managed through the runtime scheduler, according to internal parameters: *maximum depth level* in task hierarchy, *maximum number of tasks*, or *maximum number of tasks* by thread. This new feature is provided by adding a new set of queues: *team queues*. Team queues are bound to a team of threads (members of a parallel region). Then, any nano-thread on a team queue can be executed by any member of the related team. 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 *task scheduling* points, we can expect two different behaviors. First, the task is 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 that are created in the current scope (i.e., the current task). Each task data structure has a *successor* field that points to the counter the task must decrement.

6.2 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 with several applications against other existing options (nested OpenMP, Intel's task queues, and Cilk).

The applications used in this evaluation are the following:

- *Strassen*. Strassen's algorithm [25] for multiplication of large dense matrices uses hierarchical decomposition of a matrix. We used a 1,280 $\times$ 1,280 matrix for our experiments.
- *N Queens*. 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 $n \times n$ chessboard such that none of the queens attacks any other. In our experiments, we used three chessboard sizes: 12 $\times$ 12, 13 $\times$ 13, and 14 $\times$ 14.
- *FFT*. FFT computes the 1D Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey algorithm [26]. We used a vector with 33,554,432 complex numbers.
- *Multisort*. Multisort is a variation of the ordinary mergesort, which uses a parallel divide-and-conquer mergesort and a serial quicksort when the array is too small. In our experiments, we were sorting random arrays of three different sizes: of 16,777,216, 33,554,432, and 50,331,648 integer numbers.
- *Alignment*. This application aligns all protein sequences from an input file against every other sequence and computes the best scorings for each pair by means of a full dynamic programming algorithm. In our experiments, we used 100 sequences as input for the algorithm.
- *Floorplan*. The Floorplan kernel computes the optimal floorplan distribution of a number of cells. The algorithm is a recursive branch and bound algorithm. The number of cells to distribute in our experiments was 20. This application cannot be parallelized with task queues nor Cilk because we use a worksharing loop with nested tasks.
- *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. In our experiments, the matrix had 50 blocks each of 100 $\times$ 100 floats.

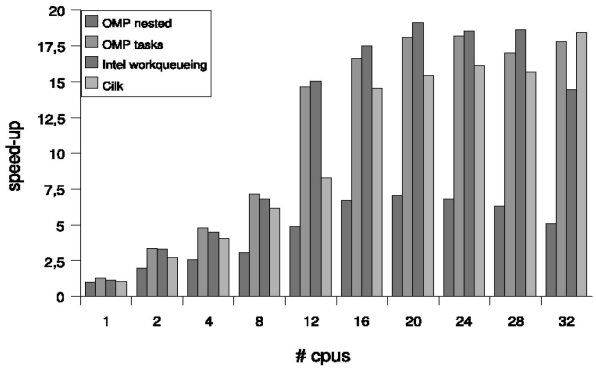

Fig. 18. FFT kernel speedups (32 millions of complex numbers).

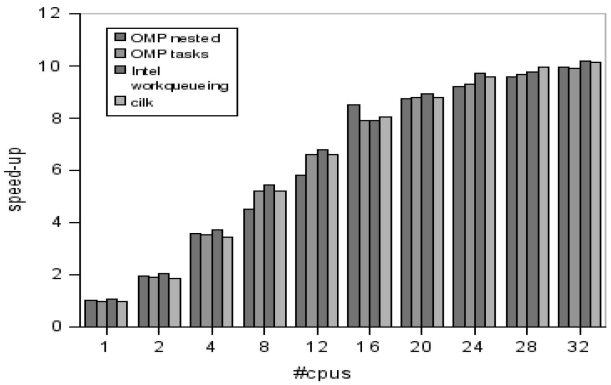

Fig. 20. Multisort speedups (32 millions of integers).

We decided the input size of each application so the tasks would not have a very fine granularity (i.e., tasks of under 10 μs of execution time). We show the results with different input sizes for two of them: N Queens and Multisort. Other applications have similar results but, for space considerations, are not shown here.

For each application, we have tried the following three OpenMP versions: 1) a single level of parallelism (labeled as `OpenMP worksharing`), 2) multiple levels of parallelism (labeled as `OpenMP nested`), and 3) OpenMP tasks. We also compare how the new tasks perform relative to other tasking models like Intel's task queues [2] and Cilk [1]. So, when possible, we have also evaluated those versions.

We evaluated all the benchmarks on an SGI Altix 4700 with 128 processors, although they were run on a CPU set comprising a subset of the machine to avoid interference with other running applications.

We compiled the codes with task queues and nested parallelism with Intel's icc compiler version 9.1 at the default optimization level. The versions using tasks uses our OpenMP source-to-source compiler and runtime prototype implementation, using icc as the backend compiler. For the Cilk versions, we use the Cilk compiler version 5.4.3 (which uses gcc as a backend).

The speedup of all versions is computed, using as a baseline the serial version of each kernel. In order to increase the fairness of our comparison, we used the serial version compiled with gcc for the Cilk evaluation and the serial version compiled with Intel's icc for the evaluation of the remaining versions. That is because Cilk uses gcc as a backend and the level of code optimization that gcc produces in some cases is inferior to icc and we are more interested in the scalability of the different models than in absolute performance, taking into account that our prototype is far from fully optimized.

6.3 Results

Fig. 18 shows the speedups achieved for the FFT kernel using OpenMP nested parallelism, our OpenMP task proposal, Intel's task queues, and Cilk. The version that uses OpenMP nested parallelism flattens out very quickly while the OpenMP version using tasks competes closely with the task queues and Cilk versions.

Figs. 19, 20, and 21 show the speedup results for the multisort kernel with different input sizes. We can see that all the different versions have problems in scaling because, in this benchmark, there is a lot of memory movement that impacts its scalability. Overall, all the different models obtain a similar performance.

Figs. 22, 23, and 24 show the speedups obtained for the N Queens kernel, with different input sizes, for all different models (OpenMP nested, OpenMP tasks, task queues, and Cilk). We can see that nested OpenMP version does not scale well but the version with the new tasks scales up very well, obtaining slightly better speedups than the task queues and Cilk versions. We can also observe that as we increase the granularity of the

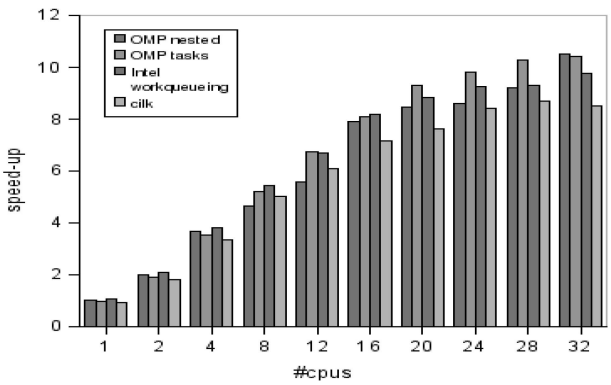

Fig. 19. Multisort speedups (16 millions of integers).

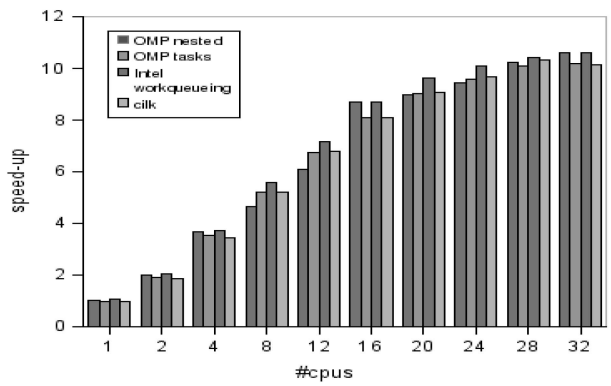

Fig. 21. Multisort speedups (48 millions of integers).

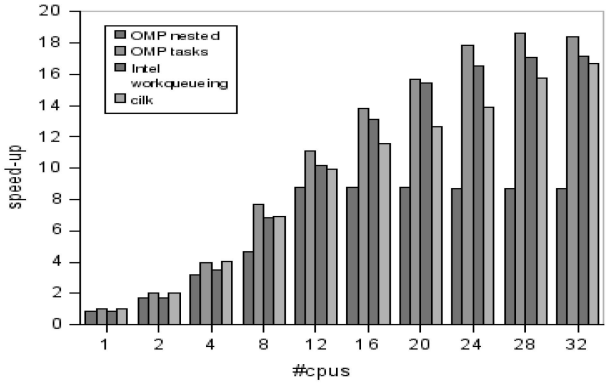

Fig. 22. N Queens speedups (12 × 12 board size).

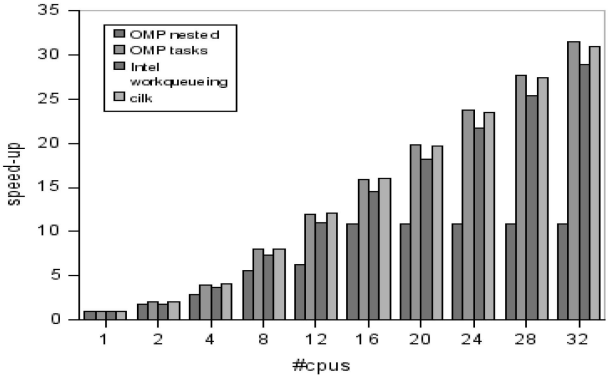

Fig. 24. N Queens speedups (14 × 14 board size).

tasks (by increasing the board size), we obtain an increase in performance with all models, something that did not happen in the multisort kernel. This is because granularity is the dominant factor in N Queens whereas that is not the case for multisort.

We have evaluated two versions of the Strassen kernel (see Fig. 25): one with the new OpenMP tasks and one with task queues. The task queues version performs better than the OpenMP tasks version, particularly with 16 CPUs or more. We can see also that the speedup curve for the OpenMP tasks version seems to flatten after 16 CPUs which is not unexpected as the runtime has not gone through the proper tuning to scale up to a large number of processors.

Fig. 26 shows the speedups for Floorplan. Here, we see again the same pattern as in FFT. The OpenMP nested version does not scale at all while the version with tasks scales as well as the task-queue version. We can see again that the speedup starts to flatten as we scale to larger number of CPUs.

In Fig. 27, we show the speedups for the SparseLU kernel. We evaluated five versions: with one level of parallelism (OpenMP workshare), with two levels of parallelism (OpenMP nested), with the new OpenMP tasks, with task queues, and with Cilk. The OpenMP tasks version performs much better than the rest. The only close one is the task-queue version. The versions using only workshares (OpenMP workshare and OpenMP nested) actually decrease in performance with larger CPU counts. The Cilk version does not scale at all because it has granularity problems (as the block size was increased, we started to see some speedup).

Fig. 28 shows the *alignment* application speedups. We have evaluated a single-level OpenMP version, another with nested parallelism, and a third one that has task parallelism nested into a regular OpenMP workshare (labeled "OpenMP tasks"). This third kind of parallelization cannot be done easily using either task queues or Cilk. The results show that the regular OpenMP versions scale quite well up to 16 processors then they start to flatten. But the version that uses tasks continues scaling up to 32 processors. The reason behind this is that the tasks nested inside the workshare are executed immediately while the number of processors is small but are generated when the number of processors increases, allowing more work to be shared (i.e., increasing the amount of available parallelism).

Overall, the OpenMP task versions perform equally well or better than other versions in most applications (*FFT*, *N Queens*, *Floorplan*, *SparseLU*, and *alignment*) and, while there seems some issues regarding scalability (*Strassen* and *Floorplan*) and locality exploitation (*multisort*), taking into account that the prototype implementation has not been well tuned, the results show that the new model will allow codes to obtain at least the performance of other models and is even more flexible.

7 Conclusion

We have presented the work of the OpenMP 3.0 tasking subcommittee: A proposal to integrate task parallelism into

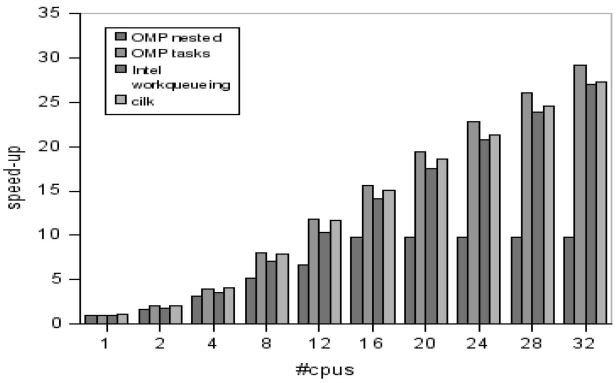

Fig. 23. N Queens speedups (13 × 13 board size).

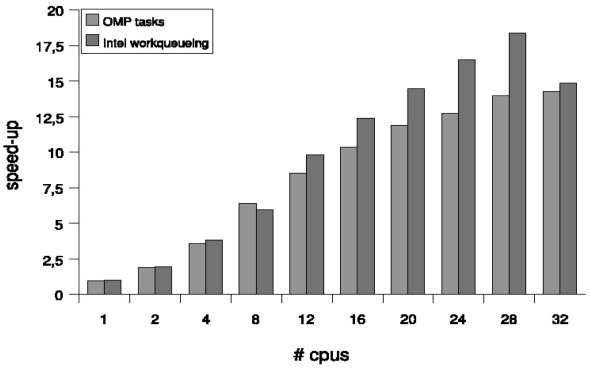

Fig. 25. Strassen speedups (1,280 × 1,280 matrix).

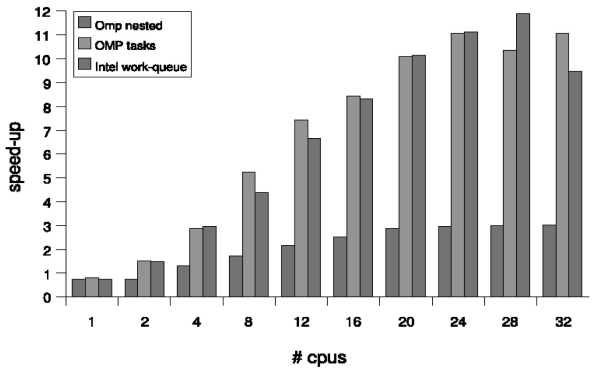

Fig. 26. Floorplan speedups (20 cells).

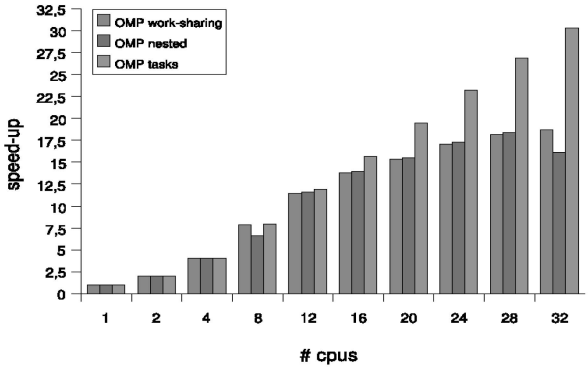

Fig. 28. Alignment speedups (100 sequences).

the OpenMP specification. This proposal allows programmers to parallelize program structures like `while` loops and recursive functions more easily and efficiently. We have shown that, in fact, these structures are easy to parallelize with the new proposal.

The process of defining the proposal has not been without difficult decisions, as we tried to achieve conflicting goals: *simplicity of use*, *simplicity of specification*, and *consistency with the rest of OpenMP*. Our discussions identified trade-offs between the goals, and our decisions reflected our best judgments of the relative merits of each. We also described how some parts of the current specification had to change to accommodate our proposal.

We have also presented a reference implementation that allows us to evaluate the samples we have discussed in this paper. The comparisons of these results show that expressiveness is not incompatible with performance and the OpenMP tasks implementation can achieve very promising speedups when compared to other established models.

Overall, OpenMP tasks provide a balanced, flexible, and very expressive dialect for expressing unstructured parallelism in OpenMP programs.

8 Future Work

So far, we have presented a proposal to seamlessly integrate task parallelism into the current OpenMP standard. The proposal covers the basic aspects of task parallelism, but other areas are not covered by the current proposal and may be the subject of future work.

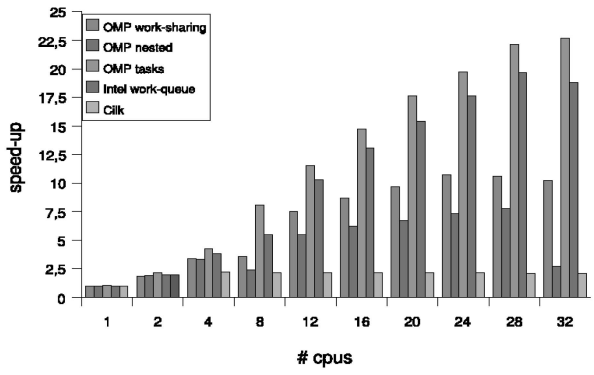

Fig. 27. SparseLU speedups (50 100 $\times$ 100 blocks).

One such possible extension is a reduction operation performed by multiple tasks. Another is specification of dependencies between tasks, or point-to-point synchronizations among tasks. These extensions may be particularly important when dealing with applications that can be expressed through a task graph or that use pipelines. Another possible extension to the language would be to enhance the semantics of the data capturing clauses so it would be easier to capture objects through pointers (as in the Floorplan example).

The OpenMP task proposal allows a lot of freedom for the runtime library to schedule tasks. Several simple strategies for scheduling tasks exist but it is not clear which will be better for the different target applications as these strategies have been developed in the context of recursive applications. Furthermore, more complex scheduling strategies can be developed that take into account characteristics of the application that can be found either at compile time or runtime. Another option would be developing language changes that allow the programmer to have greater control about the scheduling of tasks so they can implement complex schedules. This can be useful for applications that need schedules that are not easily implementable by the runtime environment (e.g., shortest job time, round-robin) [8]. One such language change that is quite simple is defining a `taskyield` directive that allows the programmer to insert switching points in specific places of the code. This would help, for example, the Boa Webserver and UI from Section 5.3 as it could be used to decrease the response time of the generated tasks [27].

Another exploration path from this proposal is the redefinition of different aspects of the OpenMP specification. For example, redefining worksharing loops in terms of tasks would allow us to define the behavior of worksharing loops for unknown iteration spaces easily or to allow the nesting of worksharing constructs. But this redefinition is not without problems. It is not clear how different aspects of the thread-centric nature of OpenMP (e.g., threadprivate and schedule) can be redefined in terms of tasks (if they can be at all).

Acknowledgments

The authors would like to acknowledge the rest of participants in the tasking subcommittee (Brian Bliss, Mark Bull, Eric Duncan, Roger Ferrer, Grant Haab, Diana King,

Kelvin Li, Xavier Martorell, Tim Mattson, Jeff Olivier, Paul Petersen, Sanjiv Shah, Raul Silvera, Ernesto Su, Matthijs van Waveren, and Michael Wolfe) and the language committee members for their contributions to this tasking proposal. 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). They would like also to acknowledge the Barcelona Supercomputing Center for letting them access to its computing resources.

REFERENCES

[1] M. Frigo, C.E. Leiserson, and K.H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '98)*, pp. 212-223, 1998.

[2] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible Control Structures for Parallelism in OpenMP," *Proc. First European Workshop OpenMP (EWOMP '99)*, Sept. 1999.

[3] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: A Research Compiler for OpenMP," *Proc. Sixth European Workshop OpenMP (EWOMP '04)*, pp. 103-109, Sept. 2004.

[4] *OpenMP Application Program Interface, Version 2.5*, OpenMP Architecture Review Board, May 2005.

[5] F. Massaioli, F. Castiglione, and M. Bernaschi, "OpenMP Parallelization of Agent-Based Models," *Parallel Computing*, vol. 31, nos. 10-12, pp. 1066-1081, 2005.

[6] R. Blikberg and T. Sørevik, "Load Balancing and OpenMP Implementation of Nested Parallelism," *Parallel Computing*, vol. 31, nos. 10-12, pp. 984-998, 2005.

[7] S. Salvini, "Unlocking the Power of OpenMP," *Proc. Fifth European Workshop OpenMP (EWOMP '03)*, invited lecture, Sept. 2003.

[8] F.G.V. Zee, P. Bientinesi, T.M. Low, and R.A. van de Geijn, "Scalable Parallelization of FLAME Code via the Workqueuing Model," *ACM Trans. Math. Software*, submitted, 2006.

[9] J. Kurzak and J. Dongarra, *Implementing Linear Algebra Routines on Multi-Core Processors with Pipelining and a Look Ahead*, Dept. Computer Science, Univ. of Tennessee, LAPACK Working Note 178, Sept. 2006.

[10] K.M. Chandy and C. Kesselman, "Compositional C++: Compositional Parallel Programming," Technical Report CaltechCSTR: 1992.cs-tr-92-13, California Inst. Technology, 1992.

[11] M. Gonzàlez, E. Ayguadé, X. Martorell, and J. Labarta, "Exploiting Pipelined Executions in OpenMP," *Proc. 32nd Ann. Int'l Conf. Parallel Processing (ICPP '03)*, Oct. 2003.

[12] J. Reinders, *Intel Threading Building Blocks.* O'Reilly Media Inc., 2007.

[13] D. Leijen and J. Hall, "Optimize Managed Code for Multi-Core Machines," *MSDN Magazine*, pp. 1098-1116, Oct. 2007.

[14] T.X.D. Team, "Report on the Experimental Language X10," technical report, IBM, Feb. 2006.

[15] D. Callahan, B.L. Chamberlain, and H.P. Zima, "The Cascade High Productivity Language," *Proc. Ninth Int'l Workshop High-Level Parallel Programming Models and Supportive Environments (HIPS '04)*, pp. 52-60, Apr. 2004.

[16] *The Fortress Language Specification, Version 1.0 B*, Mar. 2007.

[17] J. Subhlok, J.M. Stichnoth, D.R. O'Hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," *Proc. Fourth ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP '93)*, pp. 13-22, 1993.

[18] S. Chakrabarti, J. Demmel, and K. Yelick, "Modeling the Benefits of Mixed Data and Task Parallelism," *Proc. Seventh Ann. ACM Symp. Parallel Algorithms and Architectures (SPAA '95)*, pp. 74-83, 1995.

[19] T. Rauber and G. Rünger, "Tlib: A Library to Support Programming with Hierarchical Multi-Processor Tasks," *J. Parallel and Distributed Computing*, vol. 65, no. 3, pp. 347-360, 2005.

[20] S. Ramaswamy, S. Sapatnekar, and P. Banerjee, "A Framework for Exploiting Task and Data Parallelism on Distributed Memory Multicomputers," *IEEE Trans. Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1098-1116, Nov. 1997.

[21] H. Bal and M. Haines, "Approaches for Integrating Task and Data Parallelism," *IEEE Concurrency*, see also *IEEE Parallel and Distributed Technology*, vol. 6, no. 3, pp. 74-84, July-Sept. 1998.

[22] X. Teruel, X. Martorell, A. Duran, R. Ferrer, and E. Ayguadé, "Support for OpenMP Tasks in Nanos v4," *Proc. Conf. Center for Advanced Studies on Collaborative Research (CASCON '07)*, Oct. 2007.

[23] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos Mercurium: A Research Compiler for OpenMP," *Proc. Sixth European Workshop OpenMP (EWOMP '04)*, Oct. 2004.

[24] C. Polychronopoulos, "Nano-Threads: Compiler Driven Multithreading," *Proc. Fourth Int'l Workshop Compilers for Parallel Computing (CPC '93)*, Dec. 1993.

[25] P.C. Fischer and R.L. Probert, "Efficient Procedures for Using Matrix Algorithms," *Proc. Second Int'l Colloquium Automata, Languages and Programming (ICALP '74)*, pp. 413-427, 1974.

[26] J. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Math. Computation*, vol. 19, pp. 297-301, 1965.

[27] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel, "An Experimental Evaluation of the New OpenMP Tasking Model," *Proc. 20th Int'l Workshop Languages and Compilers for Parallel Computing (LCPC '07)*, Oct. 2007.

Eduard Ayguadé received the engineering degree in telecommunications and the PhD degree in computer science from the Universitat Politècnica de Catalunya (UPC), Barcelona, in 1986 and 1989, respectively. Since 1987, he has been lecturing on computer organization and architecture and parallel programming models. Since 1997, he has been a full professor in the Departament d'Arquitectura de Computadors, UPC. He is currently an associate director for research on computer sciences at the Barcelona Supercomputing Center (BSC), Barcelona. His research interests include the areas of multicore architectures, and programming models and compilers for high-performance architectures.

Nawal Copty received the PhD degree in computer science from Syracuse University. She leads the OpenMP project at Sun Microsystems Inc., Menlo Park, California. She represents Sun at the OpenMP Architecture Review Board. Her research interests include parallel languages and architectures, compilers and tools for multithreaded applications, and parallel algorithms. She is a member of the IEEE Computer Society.

Alejandro Duran received the degree in computer engineering from the Universitat Politécnica de Catalunya (UPC), Barcelona, in 2002, where he is currently a PhD candidate in the Departament d'Arquitectura de Computadors. He also holds as an assistant professor position. His research interests include parallel environments, programming languages, compiler optimizations, and operating systems.

Jay Hoeflinger received the BS, MS, and PhD degrees from the University of Illinois at Urbana-Champaign in 1974, 1977, and 1998, respectively. He has worked at the Center for Supercomputing Research and Development and the Center for Simulation of Advanced Rockets. He joined Intel, Champaign, Illinois in 2000. He has participated in the OpenMP 2.0, 2.5, and 3.0 language committee work. His research interests include automatic parallelization, compiler optimizations, parallel languages, and tools for programming parallel systems.

Yuan Lin received the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2000. He is a senior staff engineer in the software organization of Sun Microsystems Inc., Menlo Park, California. Before that, he was a compiler architect at Motorola StarCore Design Center. His research interests include compilers, tools, and languages support for parallel programming.

Federico Massaioli received the degree in physics from the University of Roma Tor Vergata in 1992. His main activities involve parallel simulation and data analysis of Physics and Fluid Dynamics phenomena. He is the head of the Computational Physics support group in the HPC Department of CASPUR interuniversity consortium, Rome. He has participated in the OpenMP 3.0 language committee work. His research interests include application and teaching of parallel programming models and tools, HPC architectures, and Operating Systems. He is a member of the IEEE and the IEEE Computer Society.

Xavier Teruel received the BSc and MSc degrees in computer science from the Universitat Politécnica de Catalunya (UPC), Barcelona, in 2003 and 2006, respectively. He is currently a PhD student in the Departament d'Arquitectura de Computadors, UPC. His research interests include shared memory environments and parallel programming models.

Priya Unnikrishnan received the MS degree in computer science and engineering from Pennsylvania State University in 2002. She has been a staff software engineer in the Compiler Group at the IBM Toronto Software Lab, Markham, Ontario, Canada, since 2003. She works on the IBM XL compilers focusing on OpenMP and automatic parallelization. Her research interests include parallel computing, parallelizing compilers, tools, and multicore architectures. She represents IBM at the OpenMP Language Committee.

Guansong Zhang received the PhD degree from Harbin Institute of Technology in 1995. He has been a staff software engineer in the Compiler Group at the IBM Toronto Software Lab, Markham, Ontario, Canada, since 1999, where he has been in charge of OpenMP implementation and performance improvement for Fortran and C/C++ on PowerPC and cell architecture and performance-related compiler optimization techniques, including array data flow analysis, loop optimization, and auto parallelization. He was a research scientist at NPAC Center, Syracuse University.