A proposal for task parallelism in OpenMP

Eduard Ayguadé[1], Nawal Copty[2], Alejandro Duran[1], Jay Hoeflinger[3], Yuan Lin[2], Federico Massaioli[4], Ernesto Su[3], Priya Unnikrishnan[5], Guansong Zhang[5]

[1] BSC-UPC
[2] Sun Microsystems
[3] Intel
[4] CASPUR
[5] IBM

Abstract. This paper presents a novel proposal to define task parallelism in OpenMP. Task parallelism has been lacking in the OpenMP language for a number of years already. As we show, this makes certain kinds of applications difficult to parallelize, inefficient or both. A subcommittee of the OpenMP language committee, with representatives from a number of organizations, prepared this proposal to give OpenMP a way to handle unstructured parallelism. While defining the proposal we had three design goals: *simplicity of use*, *simplicity of specification* and *consistency with the rest of OpenMP*. Unfortunately, these goals were in conflict many times during our discussions. The paper describes the proposal, some of the problems we faced, the different alternatives, and the rationale for our choices. We show how to use the proposal to parallelize some of the classical examples of task parallelism, like pointer chasing and recursive functions.

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 meant to handle dense numerical applications. But, as the sophistication of parallel programmers has grown and the complexity of their applications has increased, the need for a less structured way to express parallelism with OpenMP has grown. Users now need a way to simply identify units of independent work, leaving the decision about scheduling them to the runtime system. This model is typically called "tasking" and has been embodied in a number of projects, for example Cilk [1].

The demonstrated feasibility of previous OpenMP-based tasking extensions (for example workqueueing [2] and dynamic sections [3]) combined with the desire of users to standardize it, has caused one of the priorities of the OpenMP 3.0 effort to be defining a standardized tasking dialect. In September of 2005, the tasking subcommittee of the OpenMP 3.0 language committee began meeting, with a goal of defining this tasking dialect. Representatives from Intel, UPC, IBM, Sun, CASPUR and PGI formed the core of the subcommittee.

This paper, written by some of the tasking subcommittee members, is a description of the resulting tasking proposal, including our motivations, our goals for the effort, the design principles we attempted to follow, the tough decisions we had to make and our reasons for them. We will also present examples to illustrate how tasks are written and used under this proposal.

2 Motivation and related work

OpenMP "is somewhat tailored for large array-based applications"[4]. This is evident in the limitations of the two principal mechanisms to distribute work among threads. In the *loop* construct, the number of iterations must be determined on entry to the loop and cannot be changed during its execution. In the `sections` construct, the sections are statically defined at compile time.

A common operation like a dynamic linked list traversal is thus difficult to parallelize in OpenMP. A possible approach, namely the transformation at run time of the list to an array, as shown in fig. 1, pays the overheads of the array construction, which is not easy to parallelize. Another approach, using the `single nowait` construct as shown in fig. 2, can be used. While elegant, it's non-intuitive, and inefficient because of the usually high cost of the `single` construct[5].

```
1  p = listhead;
2  num_elements=0;
3  while (p) {
4     list_item[num_elements++]=p;
5     p=next(p);
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

```
#pragma omp parallel private (p)
{
  p = listhead;
  while(p) {
    #pragma omp single nowait
      process(p);
    p = next(p);
  }
}
```

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

Both techniques above lack generality and flexibility. Many applications (ranging from document bases indexing to adaptive mesh refinement) have a lot of potential concurrency, which is not regular in form, and varies with processed data. Dynamic generation of different units of work, to be asynchronously executed, allows one to express irregular parallelism, to the benefit of performance and program structure. Nested parallelism could be used to this aim, but at the price of significant performance impacts and increased synchronizations.

The present OpenMP standard also lacks the ability to specify structured dependencies among different units of work. The `ordered` construct assumes a sequential ordering of the activities. The other OpenMP synchronization constructs, like `barrier`, actually synchronize the whole team of threads, not work units. This is a significant limitation on the coding of hierarchical algorithms

like those used in tree data structure traversal, multiblock grid solvers, adaptive mesh refinement[6], dense linear algebra [7–9] (to name a few). In principle, nested parallelism could be used to address this issue, as in the example shown in fig. 3. However, overheads in parallel region creation, risks of oversubscribing system resources, difficulties in load balancing, different behaviors of different implementations, make this approach impractical.

```
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

The Cilk programming language[1] is an elegant, simple, and effective extension of C for multithreading, based on dynamic generation of tasks. It is important and instructive, particularly because of the *work-first* principle and the *work-stealing* technique adopted. However, it lacks most of the features that make OpenMP very efficient in many computational problems.

The need to support irregular forms of parallelism in HPC is evident in features being included in new programming languages, notably X10 [10] (*activities* and *futures*), and Chapel [11] (the `cobegin` statement).

The Intel *workqueueing* model [2] was the first attempt to add dynamic task generation to OpenMP. This proprietary extension allows hierarchical generation of tasks by nesting of `taskq` constructs. Synchronization of descendant tasks is controlled by means of implicit barriers at the end of `taskq` constructs. The implementation exhibits some performance issues [5, 8].

The Nanos group at UPC proposed *Dynamic Sections* as an extension to the standard `sections` construct to allow dynamic generation of tasks [3]. Direct nesting of `section` blocks is allowed, but hierarchical synchronization of tasks can only be attained by nesting of a parallel region. The Nanos group also proposed the `pred` and `succ` constructs to specify precedence relations among statically named `section`s in OpenMP [12]. These are issues that may be explored as part of our future work.

The point-to-point synchronization explored at EPCC [13] improves in performance and flexibility with respect to OpenMP `barrier`. However, the synchronization still involves threads, not units of work.

3 Task proposal

The current OpenMP specification (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. Usually, threads share work units through the use of worksharing constructs. Each work unit is bound to a specific thread for its whole lifetime. The work units use the data environment of the thread they are bound to.

Our proposal allows the programmer to specify deferrable units of work that we call *tasks*. Tasks, unlike current work units, are not bound to a specific thread. There is no a priori knowledge of which thread will execute a task. Task execution may be deferred to a later time, and different parts of a task may be executed by different threads. As well, tasks do not use the data environment of any thread but have a data environment of their own.

3.1 Terminology

Task A structured block, executed once for each time the associated task construct is encountered by a thread. The task has private memory associated with it that stays persistent during a single execution. The code in one instance of a task is executed sequentially.

Suspend/resume point A suspend point is a point in the execution of the task where a thread may suspend its execution and run another task. Its corresponding resume point is the point in the execution of the task that immediately follows the suspend point in the logical code sequence. Thus, the execution that is interrupted at a suspend point resumes at the matching resume point.

Thread switching A property of the task that allows its execution to be suspended by one thread and resumed by a different thread, across a suspend/resume point. Thread switching is disabled by default.

3.2 The task construct

The C/C++ syntax[6] is as follows:

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

The thread that encounters the `task` construct creates a task for the associated structured block, but its execution may be deferred until later. The execution gives rise to a task region and may be done by any thread in the current parallel team.

A task region may be nested inside another task region, but the inner one is not considered part of the outer one. The `task` construct can also be specified

[6] Fortran syntax is not shown in this paper because of space limitations

inside any other OpenMP construct or outside any explicit parallel construct. A task is guaranteed to be complete before the next associated thread or task barrier completes.

The optional clauses can be chose from:

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

The `untied` clause enables thread switching for this task. A task does not inherit the effect of a `untied` clause from any outer task construct.

The remaining three clauses are related to data allocation and initialization for a task. Variables listed in the clauses must exist in the enclosing scope, where each is referred to as the *original* variable for the variable in the task.

References within a task to a variable listed in its `shared` clause refer to the original variable. New storage is created for each `captureprivate` variable and initialized to the original variable's value. All references to the original variable in the task are replaced by references to the new storage. Similarly, new storage is created for `private` variables and all references to them in the task refer to the new storage. However, `private` variables are not automatically initialized.

The default for all variables with implicitly-determined sharing attributes is `captureprivate`.

3.3 Synchronization constructs

Two constructs (`taskgroup` and `taskwait`) are provided for synchronizing the execution of tasks.

The C/C++ syntax for `taskgroup` and `taskwait` is as follows:

```
#pragma omp taskgroup      | #pragma omp taskwait
      structured-block     |
```

The `taskgroup` construct specifies that execution will not proceed beyond its associated structured block until all direct descendant tasks generated within it are complete. The `taskwait` construct specifies that execution will not proceed beyond it until all direct descendant tasks generated within the current task, up to that point, are complete.

There is no restriction on where a `taskgroup` construct or a `taskwait` construct can be placed in an OpenMP program. `Taskgroup` constructs can be nested.

3.4 Other constructs

When a thread encounters the `taskyield` construct, it is allowed to look for another available task to execute. This directive becomes an explicit suspend/resume point in the code. The syntax for C/C++ is:

```
#pragma omp taskyield
```

3.5 OpenMP modifications

The tasking model requires various modifications in the OpenMP 2.5 specification. We list some of these below.

- **Execution Model.** In the 2.5 specification, there is no concept of the deferral of work or of suspend or resume points. In our proposal, a task may be suspended/resumed and execution of the task may be deferred until a later time. Moreover, a task may be executed by different threads during its lifetime, if thread switching is explicitly enabled.
- **Memory Model.** In the 2.5 specification, a variable can be shared among threads or be private to a thread. In the tasking model, a variable can be shared among tasks, or private to a task. The default data sharing attributes for tasks differs from the defaults for `parallel` constructs.
- **Threadprivate data and properties**. Inside OpenMP 2.5 work units, `threadprivate` variables and thread properties (e.g. thread id) can be used safely. Within tasks, this may not be the case. If thread switching is enabled, the executing thread may change across suspend or resume points, so thread id and `threadprivate` storage may change.
- **Thread Barrier.** In our proposal, a task is guaranteed to be complete before the associated thread or task barrier completes. This gives a thread barrier extra semantics that did not exist in the 2.5 specification.
- **Locks.** In the 2.5 specification, locks are owned by threads. In our proposal, locks are owned by tasks.

4 Design principles

Unlike the structured parallelism currently exploited using 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, their trade-offs and our design goals.

What form should the tasking construct(s) take?

Option 1 *A new work-sharing construct pair*: It seemed like a natural extension of OpenMP to use a work-sharing 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 work-sharing construct. However, these constructs would inherit all the restrictions applicable to work-sharing 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.

Option 2 *One new OpenMP construct*: The other option was to define a single task construct which could be placed anywhere in the program and which 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 the most easy to merge into the rest of OpenMP, so this was our choice.

Should we allow thread switching and what form should it take? Thread switching is a concept that is alien to traditional OpenMP. In OpenMP, a thread always executes a unit of work in a work-share from start to finish. This has encouraged people to use the thread number to identify a unit of work, and to store temporary data in `threadprivate` storage. This has worked very well for the kind of parallelism that can be exploited in regular control structures. However, tasking is made for irregular parallelism. Tasks can take wildly differing amounts of execution time. It is very possible that the only thread eligible to generate tasks might get stuck executing a very long task. If this happens, the other threads in the team could be sitting idle, waiting for more tasks, and starvation results. Thread switching would enable one of the waiting threads to take over task-generation in a situation like this.

Thread switching provides greater flexibility and potentially higher performance. It allows *work-first* execution of tasks, where a thread immediately executes the encountered task and another thread continues where the first thread left off. It has been shown that work-first execution can result in better cache reuse [1]. However, switching to a new thread changes the thread number and any *threadprivate* data being used, which could be surprising to an experienced OpenMP programmer.

Balancing the benefits of thread switching against the drawbacks, it was decided to allow thread switching in a limited and controlled manner. We decided to disable thread switching by default. It is only allowed inside tasks using the `untied` clause. Without the `untied` clause, the programmer can depend on the thread number and `threadprivate` data in a task just as in other parts of OpenMP.

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 over-written, destroying local data listed as `shared` by the task.

The committee's original decision 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 be forced to conservatively insert a task barrier immediately after many task constructs, severely restricting the parallelism possible with tasks.

Our decision was to simply state that it is the user's responsibility to insert any needed task barriers to provide any needed stack safety.

What should be the default data-sharing attributes for variables in tasks? Data-sharing attributes for variables can be pre-determined, implicitly determined or explicitly determined. Variables in a task that have pre-determined sharing attributes are not allowed in clauses, and explicitly-determined variables do not need defaults, by definition. However, the data-sharing attributes for implicitly-determined variables require 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 `captureprivate`, to make sure that it will not be deallocated before it is used. Since we decided to not guarantee stack safety for tasks, we faced a hard choice:

Option 1 make data primarily `shared`, analogous to using `shared` in the rest of OpenMP. This choice is consistent with existing OpenMP. But, with this default, the danger of data going out of scope is very high. This would put a heavy burden on the user to ensure that all the data remains allocated while it is used in the task. Debugging can be a nightmare for things that are sometimes deallocated prematurely.

Option 2 make data primarily `captureprivate`. The biggest advantage of this 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 `captureprivate` as the default is that Fortran parameters and C++ reference parameters will, by default, be captured by tasks. This could lead to errors when a task writes into reference parameters.

Option 3 make some variables `shared` and some `captureprivate`. With this choice, the rules could become very complicated, and with complicated rules the chances for error increase. The most likely effect would be to force the

programmer to explicitly place all variables in some clause, as if there were no defaults at all.

In the end, we decided to make all variables with implicitly-determined sharing attributes default to `captureprivate`. While not perfect, this choice gives programmers the most safety, while not being overly complex.

5 Examples of use

In this section, we revisit examples we used in section 2, and write those applications based on our task proposal.

5.1 Pointer chasing in parallel

Pointer chasing (or *pointer following*) can be described as a segment of code working on a list of data items linked by pointers. When there is no dependence, the process can be executed in parallel.

We already described in Section 2 a couple of non-efficient or non-intuitive parallelization strategies for this kernel (fig. 1 and 2). In addition, both solutions fail if the list itself needs to be updated dynamically during the execution. In fact, with proper synchronization, the second solution may append more items at the end of the link, while the first one will not work at all.

All these problems go away with the new task proposal. This simple kernel could be parallelized as shown in fig. 4.

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5      p = listhead;
6      while(p) {
7        #pragma omp task
8          process(p)
9        p=next(p);
10     }
11   }
12 }
```

Fig. 4. Parallel pointer chasing using `task`

```
#pragma omp parallel private (p)
{
  #pragma omp for
  for (int i=0; i< num_lists; i++) {
    p = listheads[i];
    while(p) {
      #pragma omp task
        process(p)
      p=next(p);
    }
  }
}
```

Fig. 5. Parallel pointer chasing on multiple lists using `task`

The `task` construct gives more freedom for scheduling (see the next section for more). It is also more straightforward to add a synchronization mechanism to work on a dynamically updated list[7]

[7] We will leave it as an exercise for readers to write the `next` function in the different versions.

```
1 int fib(int n) {
2   int x, y, sum;
3   if (n<2)
4     return n;
5 #pragma omp taskgroup
6   {
7 #pragma omp task shared(x)
8     x=fib(n-1);
9 #pragma omp task shared(y)
10    y=fib(n-2);
11  }
12  return x+y;
13 }
```

Fig. 6. Fibonacci with recursive `task`

In fig. 4, the `single` construct ensures that only one thread will encounter the `task` directive and start a task nest. It is also possible to use other OpenMP constructs, such as `sections` and `master`, or even an `if` statement using thread id, to achieve the same effect.

More interestingly, we may have multiple lists to be processed simultaneously by the threads of the team, as in fig. 5. 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.

5.2 Recursive task

Another scenario of using the `task` directive is shown in fig. 6, inspired by one of the examples in the Cilk project [1]. This code segment calculates the Fibonacci number. If a call to this function is encountered by a single thread in a parallel region, a nested task region will be spawned to carry out the computation in parallel. The `TASKGROUP` construct defines a code region which the encountering thread should wait for. Both `i` and `j` are on the stack of the parent function that invokes the new tasks. Please refer to section 3.3 for the details. Notice that although OpenMP constructs are combined with a recursive function, it is still equivalent to its sequential counterpart.

Other applications falling into this category include: virus shell assembly, graphics rendering, n-body simulation, heuristic search, dense and sparse matrix computation, friction-stir welding simulation and artificial evolution.

We can also rewrite the example in fig. 3 as in fig. 7. In this figure, we use `task` to avoid the nested `parallel` regions. Also, we can use a flag to make the post order processing optional.

6 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

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

Fig. 7. Parallel depth-first tree traversal

be subject of future work. 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 for dealing with applications that can be expressed through a task graph or that use pipelines.

The 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 which can be found either at compile time or run time. Another option would be developing language changes that allow the programmer to have greater control of the scheduling of tasks so they can implement complex schedules (e.g. shortest job time, round robin) [8].

7 Conclusions

We have presented the work of the tasking subcommittee: a proposal to integrate task parallelism into the OpenMP specification. This 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 need to be changed to accommodate our proposal.

In the end, however, we feel that we have devised a balanced, flexible, and very expressive dialect for expressing unstructured parallelism in OpenMP programs.

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, Xavier Teruel, Matthijs van Waveren and Michael Wolfe) and the language committee members for their contributions to this tasking proposal. The Nanos group at BSC-UPC is supported 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. 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.
2. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible control structures for parallellism in OpenMP. In *1st European Workshop on OpenMP*, September 1999.
3. J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta. Nanos Mercurium: A Research Compiler for OpenMP. In *6th European Workshop on OpenMP (EWOMP '04)*, pages 103–109, September 2004.
4. OpenMP Architecture Review Board. *OpenMP Application Program Interface*, May 2005.
5. F. Massaioli, F. Castiglione, and M. Bernaschi. OpenMP parallelization of agent-based models. *Parallel Computing*, 31(10-12):1066–1081, 2005.
6. R. Blikberg and T. Sørevik. Load balancing and OpenMP implementation of nested parallelism. *Parallel Computing*, 31(10-12):984–998, 2005.
7. S. Salvini. Unlocking the Power of OpenMP. Invited lecture at 5th European Workshop on OpenMP (EWOMP '03), September 2003.
8. 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.
9. 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.
10. The X10 Design Team. Report on the Experimental Language X10. Technical report, IBM, February 2006.
11. D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *9th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS 2004)*, pages 52–60. IEEE Computer Society, April 2004.
12. Marc Gonzalez, Eduard Ayguadé, Xavier Martorell, and J. Labarta. Exploiting pipelined executions in OpenMP. In *32nd Annual International Conference on Parallel Processing (ICPP'03)*, October 2003.
13. J. M. Bull and C. Ball. Point-to-Point Synchronisation on Shared Memory Architectures. In *5th European Workshop on OpenMP (EWOMP '03)*, September 2003.