The Birth of OpenMP 3.0
by Larry Meadows
CEO, The OpenMP ARB

OpenMP is a set of directives for C++, C, and Fortran programs that makes it easier to express shared-memory parallelism. The advent of commodity inexpensive multi-core processors and corresponding OpenMP-capable compilers (including gcc) has recently increased the popularity of OpenMP. However, since its invention in 1997, very few new features have been introduced. As originally specified, OpenMP is primarily suited to loop-level parallelism; other styles of parallelism are possible, but usually involve explicit domain decomposition and thread-number dependent code (much like an MPI program).

For the past 18 months, the language committee of the OpenMP has been hard at work on the next revision of the OpenMP standard. A draft of revision 3.0 for public comment was released on Oct. 21, 2007 and is available at http://www.openmp.org/drupal/mp-documents/spec30_draft.pdf.

OpenMP 3.0 contains a number of new features:
- Support for tasking
- Addition of loop collapse directive
- Enhanced loop schedules
- Improved nested parallelism support

The most innovative feature in OpenMP 3.0 is tasking. Tasking allows the dynamic generation of unstructured parallelism. Programming models like Cilk and the tasking extensions in the Intel compiler are antecedents to the OpenMP 3.0 tasking model.

Integrating tasks into the thread-centric model of OpenMP was a challenge. The key observation was that OpenMP already has tasks; an OpenMP program that does not contain explicit task directives consists of implicit tasks. OpenMP 3.0 simply adds the ability to create explicit tasks.

A task is similar to a lambda function in C#. The task directive is used to describe an explicit task as follows (C/C++ is used for the examples for convenience, there are analogs for Fortran):

```c
#pragma omp task [clauses]
{
    // code
}
```

When a thread encounters the task directive, the data environment is captured. That environment, together with the code represented by the structured block, constitutes the generated task. The data-sharing attribute clauses private, firstprivate, and shared determine whether variables are private to the data environment, copied to the data
environment and made private, or shared with the thread generating the task, respectively. The task may be executed immediately or may be queued for execution.

A linked list can be traversed with code like this:

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

The parallel pragma creates a team of threads. One of those threads enters the single region with a private copy of the pointer `p`, and traverses the list, generating a task for each item in the list. The other threads wait at the barrier at the end of the single region. As tasks are generated, the threads waiting in the barrier execute them, and obtain new tasks to execute as tasks are completed. After the thread generating the tasks has traversed the entire list, it enters the barrier at the end of the single region and becomes available to execute tasks. No thread leaves the barrier at the end of the single region until all tasks have been executed.

The above example shows that all tasks created by a team in a parallel region are completed at the next barrier. It is also possible to wait for all tasks generated by a given task (whether implicit or explicit) using the new taskwait directive, as in:

```c
#pragma taskwait
```

Taskwait waits only for the children of a given task. A barrier is needed to wait for all descendants.

Only the threads in the same team as the thread that generates a task are allowed to execute that task. By default, the same thread that begins execution of a task will execute that task to completion; such as task is called a tied task. This allows the use of threadprivate data in tied tasks. In contrast, untied tasks (indicated with the untied clause on the task directive) may be executed by any thread in the team and different portions of an untied task may be executed by different threads. This is called thread switching.

Thread switching and task switching occur only at task scheduling points. Task scheduling points are constrained in tied tasks to very specific points. In untied tasks, the implementation is allowed much more freedom in inserting task scheduling points. Thus untied tasks give an implementation more freedom in runtime scheduling while limiting the assumptions that a user can make about task scheduling.
For more information about tasks, including a wealth of examples and precise descriptions of the tasking directives and task scheduling requirements, please refer to the OpenMP 3.0 draft specification.

The other main additions to OpenMP 3.0 are not as far reaching as tasking but are still important. They grew out of user experience with earlier versions of OpenMP and its shortcomings.

Perfectly-nested loops can be collapsed into a single loop and scheduled as such using the collapse directive. This provides larger pieces of parallel work. For example:

```c
#pragma omp collapse(2)
for (i = 0; i < N; ++i)
  for (j = 0; j < M; ++j)
    operate();
```

is collapsed into a single loop that is executed N*M times.

The new `schedule(auto)` clause allows the OpenMP implementation freedom to select the best schedule for an OpenMP loop construct. Further, the new `omp_set_schedule()` runtime routine can be extended by the implementation to allow the user to select schedule types that are not part of the OpenMP specification.

Support for nested parallelism has been improved in a two ways. First, most internal control variables, such as the number of threads to be used for the next parallel region, are now per-task variables, so they can be changed by runtime calls for the current task without affecting other tasks. Second, a number of new environment variables and runtime routines have been added to control the total number of threads in the program, and to allow the program to dynamically discover the parallel nesting structure.

Finally, a number of improvements and corrections have been made. For a complete list (or what we hope is a complete list) of changes from OpenMP 2.5 to OpenMP 3.0 please read Appendix F of the 3.0 draft. I would like to extend special thanks to Mark Bull of EPCC for his careful work in creating this appendix.

The OpenMP language committee is proud to have created the OpenMP 3.0 specification and hopes that you will find the new features valuable in extending OpenMP to even more shared-memory applications. Please let us know your comments, complaints and suggestions at http://www.openmp.org/forum or by sending email to feedback@openmp.org.