



Computer Science 400 Parallel Processing

Siena College
Fall 2008

Topic Notes: OpenMP

We have seen how to implement algorithms using pthreads – shared variables can be declared globally, private variables can be declared local to our thread function or functions called by the thread function.

But a compiler should be able to do a little of this work for us – convert some higher level constructs into appropriate thread calls and mutex locks.

We saw that the Sun compilers could do some parallelization completely automatically, but this doesn't always work, and it may not choose to parallelize the way we would like. A standard set of compiler directives and library functions called *OpenMP* have been developed to allow programmers to specify parallelism.

Chapter 17 of Quinn describes OpenMP. We will go through some examples here.

As always, we start with a “Hello, world” program:

See: `/cluster/examples/openmp_hello`

Things to note about the OpenMP hello world:

- We include `omp.h`, the OpenMP header file.
- We have some odd syntax just inside of `main()` that starts a parallel block:

```
#pragma omp parallel private(nthreads, tid)
```

It is a preprocessor directive, so the initial pass of the C compiler will replace this with some code to start up a number of tasks.

It means the block that follows is a parallel block which should give private copies to each task of the variables `nthreads` and `tid`.

Note the seemingly extraneous curly braces following the `#pragma`. They define the extent of the parallel block.

- Two self-explanatory query functions are called inside the parallel block:
 - `omp_get_thread_num()`
 - `omp_get_num_threads()`
- To compile this on Solaris (bullpen), we need to use Sun's `cc` with options `-xO3` and `-xopenmp=parallel`.
- To request a number of threads to be created, we set the environment variable `OMP_NUM_THREADS` to the number of threads we want. The system will only start a number of threads up to the number of available processors, by default.

Another example, back to the matrix-matrix multiply:

See: `/cluster/examples/matmult_openmp`

Again, we include `omp.h` but the only other change from a straightforward serial version is the

```
#pragma omp parallel for
```

which tells OpenMP to parallelize the upcoming for loop.

- Setting the maximum number of threads (`OMP_NUM_THREADS` environment variable or the `omp_set_num_threads()` function *requests* a certain number of threads. You are not guaranteed to get that many.
- When a thread reaches a `#pragma omp parallel` directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.

We can also take more control, similar to the way we did with `pthread`s:

- Explicit domain decomposition:

See: `/cluster/examples/matmult_omp_explicit`

Things to note:

- The simple thread creation and destruction, calling `worker()`
- The worker has a bunch of local variables, all of which are private to the calling thread
- The computation of the row range for each thread, based on `numthreads` and `threadid`

- Bag of tasks:

See: `/cluster/examples/matmult_omp_bagoftasks`

Things to note:

- No worker function here – the threads are defined by the block inside the `{ ... }` pair following the `#pragma omp parallel` directive.
- The `shared` and `private` clauses tell OpenMP which variables that we use inside the multithreaded block should be shared or private to each thread.

- The critical section for the concurrent access to `next_avail_task` is taken care of by the `#pragma omp critical(mutex)` directives. By naming the critical sections with the name “mutex” they are essentially the same critical section (as both deal with the same variable). The name “mutex” here is just a name – it could be anything. This replaces all the declarations, initialization, locking and unlocking, and destruction of the mutex in the pthread version.

More clauses to parallel directives

A parallel directive can take a number of clauses to define how variables are to be treated.

- `private(variables)`: indicate that the variables in the list are to be private to each thread created.

Any previous value is not seen by the threads, and that value is still there when the parallel block ends.

See: `/cluster/examples/openmp_private`

- `shared(variables)`: indicate that the variables in the list are to be shared among all threads created.

Any previous value is seen by all threads, and any changes made by threads will persist when the parallel block ends.

See: `/cluster/examples/openmp_shared`

- `firstprivate(variables)`: Give the copies of the variables within each thread the initial value which is the the value the variable had outside the parallel block.
- `lastprivate(variables)`: Take the values of the variables in the “last” thread (for a parallel for loop, or parallel sections) and store that in the variable outside the parallel block.
- `reduction(op:variable)`: Perform a reduction on the variable and store the reduced value in the variable when the parallel block finishes.

See: `/cluster/examples/openmp_reduction`

What is a *reduction*? Basically, the operator is applied to combine the given variable’s value in each thread, and the overall result is stored in the variable when the parallel block returns.

We’ll see many examples when we talk about message passing. Here’s one that’s a little more interesting than the made up example above:

See: `/cluster/examples/matmult_omp_explicit2`

Other parallel directives

There are several other directives worth looking at a bit:

- `sections`:

Define sections of code (that aren't a loop) that can be executed concurrently. An overly simplistic example:

See: `/cluster/examples/openmp_sections`

Each defined section is a block that can be assigned to a thread.

This is useful when we have different tasks to assign to each thread created.

- `single`:

Used within a parallel block, this specifies that the block inside the `single` should be executed by exactly one thread.

- `master`:

This is a lot like `single`, but we are guaranteed that the master thread does the execution.

- `critical`:

We've seen this – it defines a critical section.

- `barrier`:

Used within a parallel block, this causes the threads to synchronize at this point. This could be used, for example, to make sure that the threads all complete some preliminary computation before moving on to their next step.

- `atomic`:

Force a simple statement that modifies a single variable to be atomic. It is essentially a critical section, but since it is more restrictive, the compiler may choose more efficient techniques.

Yet another matrix-matrix multiply example that uses some of these:

See: `/cluster/examples/matmult_omp_explicit3`

Loop Scheduling

Before we consider the idea of loop scheduling in OpenMP, we look at a new example – computing the points in the Mandelbrot set.

Mandelbrot Set

From the Wikipedia article:

In mathematics, the *Mandelbrot set*, named after Benoit Mandelbrot, is a set of points in the complex plane, the boundary of which forms a fractal. Mathematically, the Mandelbrot set can be defined as the set of complex c -values for which the *orbit* of 0 under iteration of the complex quadratic polynomial $x_{n+1} = x_n^2 + c$ remains bounded. That is, a complex number, c , is in the Mandelbrot set if, when starting with $x_0 = 0$

and applying the iteration repeatedly, the absolute value of x_n never exceeds a certain number (that number depends on c) however large n gets.

E.g. $c = 1$ gives the sequence $0, 1, 2, 5, 26, \dots$ which leads to infinity. As this sequence is unbounded, 1 is not an element of the Mandelbrot set.

On the other hand, $c = i$ gives the sequence $0, i, (1 + i), i, (1 + i), i, \dots$ which is bounded, and so it belongs to the Mandelbrot set.

For graphical displays of the Mandelbrot set, such as what you can see on the Wikipedia page, we can assign the pixels of our display image to points in the complex plane. For each point, we compute whether the point is in the Mandelbrot set. We will not worry too much about the specifics of the computation, but we note a few things:

- The computation is iterative, and some points are more expensive than others to compute (more iterations).
- When parallelizing, we might need to take this into account.

We have an OpenMP parallelization of a Mandelbrot set calculator:

See: `/cluster/examples/mset_openmp`

Most of this program is pretty straightforward.

- We figure out what we're supposed to compute from the command line, using reasonable default values.
- We have a parallelized for loop that computes the values in our array – 0 if the point is not in the M-set, 1 if it is.
- Note the `schedule(runtime)` at the end of our OpenMP directive – more on this in a bit.
- We then print out which thread computed each row and how many total rows each thread computed.
- We optionally generate an image, but this is currently broken.

To this point, we have allowed OpenMP to do what it feels best in terms of how to break up the iterations in a parallel `for` loop. That's fine for the matrix-matrix multiply, where every row is the same cost.

However, in this case, each pixel has a potentially different cost.

- pixels far from the M-set can be determined to be outside the set very quickly – in just a few iterations.

- pixels inside the set require (at least when using our simple procedure) that we continue iterating until we have reached the maximum number of iterations.
- pixels near the set will require more iterations.

Unless we get lucky (as is the case for the 2-thread version of the whole set calculation), each thread is going to have a different workload assigned.

The default *static* decomposition will be insufficient.

We have already seen one way to deal with this problem - the bag of tasks approach. We could implement a bag of tasks in OpenMP like we did for pthreads.

Fortunately, OpenMP makes it even easier by allowing us to provide loop scheduling “hints” as to how to schedule the tasks in a loop:

```
#pragma omp parallel for schedule(kind [,chunk])
```

In the above, *kind* specifies the loop scheduling type and the optional parameter *chunk* specifies chunk sizes.

Here are the valid schedules we can specify for a loop with n iterations to be executed by t threads:

- *static* – assign about $\frac{n}{t}$ iterations to each thread
- *static,C* – interleaved allocation of chunks of C contiguous iterations to threads
- *dynamic* – the bag of tasks equivalent, where iterations are assigned to threads one at a time as they are completed
- *dynamic,C* – same idea, but iterations are assigned in contiguous chunks of size C
- *guided,C* – another dynamic allocation, but using the *guided self-scheduling* idea. Here, we start by allocating large chunks of work to each thread, then as they finish, successively smaller chunks are assigned to threads as they complete their work, with the smallest chunk allowed to be assigned specified by C .
- *guided* – same as above, but using the default value of 1 for C .
- *runtime* – specify one of the above options using the `OMP_SCHEDULE` environment variable.

Our program uses the *runtime* scheduler so we can experiment without having to recompile.