Computer Science 335
Parallel Processing and HPC
Siena College
Fall 2021

Topic Notes: OpenMP

We have worked with two of the major paradigms for parallel computing so far: distributed memory with message passing among cooperating processes (MPI) and shared memory with cooperating threads (pthreads).

We will continue with the latter model, but look at how a smart compiler can help us out.

As pthreads programmers, we saw how it is our responsibility to decide when to create threads, decide which memory should be shared among threads either by declaring variables globally or passing pointers through the parameter to `pthread_create`. When variables are shared, it is our responsibility to ensure safe concurrent access by using mutexes or other synchronization constructs.

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

Parallelizing compilers can do some parallelization completely automatically, just by analyzing the code to determine which operations could be done in parallel and creating threads complete those operations concurrently. This is a wonderful thing – no extra work for us as programmers! – but there are significant limitations on what can be done fully automatically. Doesn't always work, and it may not choose to parallelize the way we would like.

However, by giving the compiler some hints about what specifically in our code we would like to have parallelized and how, this *compiler-directed parallelism* approach can be very effective and relatively convenient for programmers.

To this end, a standard set of compiler directives and library functions called *OpenMP* have been developed to allow programmers to specify parallelism. The OpenMP standard has been implemented in many compilers, including the version of `gcc` we have on noreaster. So let's check it out.

As always, we start with a "Hello, world" program:

`https://github.com/SienaCSISParallelProcessing/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(num_threads, thread_num)
  ```

  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 `num_threads` and `thread_num`.
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 with modern versions of `gcc`, we need to add the flag `-fopenmp`. Different flags are likely needed for other compilers.

? **Question 1:**
Clone the respository on noreaster and Stampede2 and run the program on each. How many threads are created? Why? (4 points)

To request a specific 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.

If your shell is `bash`, the following command would set the environment variable to request 8 threads when running OpenMP:

```
export OMP_NUM_THREADS=8
```

? **Question 2:**
Try this on both noreaster and Stampede2, and paste your output from both. (4 points)

For a more interesting example, we revisit the matrix-matrix multiply problem. These programs are in this repository:

`https://github.com/SienaCSISParallelProcessing/openmp-matmult`

Aside: the programs in this repository introduce the idea of a common `Makefile` that is included by other `Makefiles` to reduce duplication, and to make it simpler if we wanted to run on a different system or use a different compiler that would require a change to the `CC` entry. This way, we can make the change in `Makefile.common` in the root directory of the repository and it will take effect in all of the subdirectories that have their own `Makefiles` that just include the common one.

The first we will look at is an incredibly simple OpenMP parallelization in the `simple` directory of the repository. 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.

? Question 3:
Run on noreaster with 1, 2, 4, 8, 16, 32 and 64 threads and report the timings. Use a `bash for` loop like the one below. (3 points)

To run the program with these numbers of threads, we can do it very simply with this `bash for` loop:

```
for nt in 1 2 4 8 16 32 64; do
> export OMP_NUM_THREADS=$nt
> echo "Run with $nt threads:"
> ./matmult_openmp
> done
```

Note that the > that you see at the start of all but the first line will be printed by the shell as a prompt to continue the incomplete command line.

? Question 4:
Does this program scale well up to the number of processors in the system? (2 points)

Given what you likely found in answering the questions above, and given how simple it was to achieve this parallelization with the single `#pragma` that was added to the serial code, there's not much reason to do more. We don't necessarily know exactly how the compiler and the OpenMP run-time system achieved the parallelization, but when it works so well, we probably don't really care either.

Nevertheless, we will look at how we can take more control over the parallelization. This makes the code more reminiscent of what we did with pthreads.

First, let's look at an explicit domain decomposition, in the `explicit` directory.

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 `num_threads` and `thread_num`

? Question 5:
Which variant of explicit domain decomposition is used here, block or interleaved? (2 points)

? Question 6:
Run some instances of the explicit domain decomposition version. How do the run times compare with the simple version? (2 points)

? Question 7:
How would you modify this example to use the other variant (block or interleaved)? (3 points)

Next, we can look at the bag of tasks approach, in the `bagoftasks` directory.

Things to note:

- We have no `worker` function here – the code to be executed in parallel by the threads is 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.

? Question 8:
Run some instances of the bag of tasks version. How do the run times compare with the previous versions? (2 points)

More OpenMP `parallel` directive clauses

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

We will look at some of these in the repository:

`https://github.com/SienaCSISParallelProcessing/openmp-directives`

- `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.

  An example of this is in the `private` directory.

- `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.

  An example of this is in the `shared` directory.

- `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.

  An example of this is in the `reduction` directory.

  ? Question 9:

  What is the MPI analog of this reduction? (3 points)

  Back in the `openmp-matmult` repository, the `explicit2` directory shows some of these directives applied to the matrix-matrix multiplication example.

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 is in the `sections` directory of the `openmp-directives` repository.

  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 is in the `explicit3` directory.

? **Question 10:**
For each OpenMP directive in this example, give a brief description of its purpose. (5 points)

Some Familiar Examples

OpenMP parallelizations of closest pairs and Conway's game of life are in

`https://github.com/SienaCSISParallelProcessing/openmp-closestpairs`

and

`https://github.com/SienaCSISParallelProcessing/openmp-life`

For the closest pairs when breaking up the work by parallelizing within each graph (the `byvertex` implementation), there are two modes implemented: "fine" and "coarse". One updates the closest pair after each distance calculation, the other only when the thread has completed its portion of the work.

? **Question 11:**
Run some comparisons of these two modes on various graphs and numbers of threads. Describe what you find and why this occurs. (5 points)

For the Game of Life, the parallelization is almost embarrassingly simple with OpenMP. There is just the addition of the `omp.h` header file and a single `#pragma omp` directive! It's a complicated one, including `private` and `shared` clauses, as well as three `reduction` clauses.

Question 12:
Time some runs of this program using a 1000×1000 grid and 10,000 iterations and describe your findings in terms of scalability of this parallelization. (5 points)