Arquiteturas de Alto Desempenho 2024/2025

Practical class AAD_P03 (2024-10-07 and 2024-10-08)

Tomás Oliveira e Silva

1 Objectives

Learn how to parallelize code the easy way (OpenMP), but with partial control of what is going on. List of things you will learn about:

- parallelize a section of code,
- parallelize a for cycle, and
- reduce a for cycle.

This is only the tip of a large iceberg. If you want to learn more, and there is much more to learn, consult the OpenMP documentation.

2 Running a code section in parallel

To parallelize a piece of code using pthreads, we had to write down a function, place that piece of code inside it, add glue code to pass parameters and return values, and call pthread_create and pthread_join to launch and regroup the threads. OpenMP does that in a much simplified way. It is only necessary to annotate the source code, and the compiler does the rest of the work (creating temporary variables for input/output data, thread management, and so on). The annotation is done using so called pragmas. For example, to run a section of code in parallel, using 8 threads, one just has to write

```
# pragma omp parallel num_threads(8)
{
  // int thread_number = omp_get_thread_num();
  // section of code
}
```

and that's it! All automatic variables (without static in their declarations) declared inside the section of code will not be shared between the threads. Variables declared outside of the loop are shared. Inside the section of code we can use the omp_get_thread_num() function to find out the thread number, and that can be used to specify the work that each thread will be doing.

It is also possible to parallelize a for loop. The annotation is quite similar. Here's an example:

```
# pragma omp parallel for num_threads(8)
for(i = 0;i < 1000000;i++)
{
  // loop code
}
```

The compiler splits the loop range in smaller ranges and each thread does its own range. Nice!

The file create_and_join.c, which has the same name as one of the previous practical class, but which has different code, puts these two annotations in action. Compare its code with the one of the previous class:

```
$ meld create_and_join.c ../P02/create_and_join.c
$ # or
$ vi -d create_and_join.c ../P02/create_and_join.c
```

They do essentially the same thing, one with `pthreads`, the other with `OpenMP` pragma directives. Which one is simpler?

Compile and run it:

```
$ make create_and_join
$ ./create_and_join
```

Notice how the for loop indices are split among the threads.

It is now time to roll up our sleeves and take a look at the code the compiler generates using `OpenMP` pragmas. This can be done using

```
$ objdump --disassemble ./create_and_join | less
```

or compiling the code to assembly (this is already done for you in the makefile):

```
$ make create_and_join.s
```

If the latter, open the file `create_and_join.s` with a text editor and examine its contents. Besides the code of the `main()` function, do you see code for other functions?

3 Critical sections

It is also possible to put critical sections inside a parallel code section:

```
# pragma omp parallel
  {
    // bla, bla, bla
#   pragma omp critical
    {
      // critical section
    }
    // bla, bla, bla
  }
```

The compiler will automatically create and use a mutex to control accesses to the critical section. It is possible to give a name to the critical section: just append it at the end of the pragma line, inside parenthesis:

```
#   pragma omp critical (your_name_of_the_critical_section)
```

If two or more critical sections share the same name, they will use the same mutex.

The file `lock_and_unlock.c`, which has the same name as one of the previous practical class, but which has different code, puts this annotation in action. Compare its code with the one of the previous class. Which one is simpler?

Compile and run it:

```
$ make lock_and_unlock
$ ./lock_and_unlock
```

It does the two cases — with and without a critical section – in one go. Is its behavior similar to that of the code of the previous class?

Add code, perhaps using the `wall_time()` function declared in the `create_and_join.c` file, to measure the time it takes to execute the parallel code with, and without, the critical section. How do execution times compare with those of the code of the previous class?

4 Reduction

Sometimes we use a for loop to sum the elements of an array (or, for example, to get their minimum or maximum). `OpenMP` has a way to do this, minimizing the amount of code we have to write. One just appends `reduction(op:var_list)` and the end of a `parallel` pragma. For example, to compute the sum of the elements of an array in parallel, we can do

```
  int sum = 0;
# pragma omp parallel for reduction(+:sum)
  for(int idx = 0;idx < array_size;idx++)
    sum += array[idx];
```

The compiler will compute partial sums in each thread, using a local variable and without mutexes, and at the end the values of the local variables are “reduced” (now with a mutex) using the specified reduction function (in this case, $+$). All variables specified after `:` are treated in this way.

To see this in action, compile and run the code provided in `reduction.c`:

```
$ make reduction
$ ./reduction
```

As usual, you should take a good look at the source code. Modify the code so that the reduction returns the minimum and maximum values of the array.

After doing all this, cleanup:

```
$ make clean
```

5 Putting it all together: parallelization of `MandelbrotCount.c`

It is now time to put all that you learned about `OpenMP` in action. Modify the `MandelbrotCount.c` source code so that it runs in parallel. Which is easier: doing it with `pthreads` or using `OpenMP`?