Lab 1 - OpenMP basics

Files to be used in this lab (download):

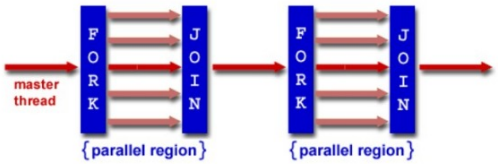
- hello_world.cpp

- mat_mult.cpp

1) OpenMP basics

OpenMP is an API for parallel programming with implementations available in C, C++ and Fortran. In our classes we'll use the C++ variant.

OpenMP as a standard consists of compile-time preprocessor directives, run-time library calls and a set of environment variables. Under the hood, the parallelisation operates according to the fork-join model.



All OpenMP programs start with a single master thread. Upon encountering a parallel region, the master thread spawns (forks) a so-called team of parallel worker threads. When these new threads finish their work, they join back into the master thread.

Each OpenMP program is a single process. It can be verified in your OS' task manager - an OpenMP application should figure as a single entry with CPU utilisation over 100%.

2) Directives

In C++, OpenMP directives are specified with the pragma preprocessing directive, as such:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

a) `#pragma omp parallel`

The `parallel` directive denotes regions of code that are to be executed in parallel by several threads. It creates a team of threads, the size of which is determined at runtime. That size can be controlled globally by an environment variable or separately for each parallel section via a `num_threads` clause.

Example:

```
#pragma omp parallel num_threads(3)
{
  cout << "Hello!" << endl;
}
```

In the above example, the "Hello!" string will be printed 3 times - 1 time for each thread from the team of 3.
Note that opening bracket "`{`" cannot be located in line containing `#pragma` directive.

b) `#pragma omp for`

This directive splits the for loop into N pieces, where N is the number of threads in the current team. Each thread is responsible for handling a different piece.

Example:

```
#pragma omp for
for (int i = 0; i < 10; i++) {
  cout << i << endl;
}
```

c) `#pragma omp sections`

The `sections` directive is useful for indicating which fragments of code can run in parallel and which must be executed in a certain order.

Example:

```
#pragma omp sections
{
  #pragma omp section
  {
    Job1();
  }
  #pragma omp section
  {
    Job2();
    Job3();
  }
  #pragma omp section
  {
    Job4();
  }
}
```

Each block marked with the `section` directive is executed by exactly one thread from the current team. This means that these blocks can be ran in parallel, but their contents need to be finished in sequence (such as Job2 and Job3 above).

d) `#pragma omp parallel for/sections`

The `for` and `sections` directives delegate the work to threads from the current team. If we use them in isolation, the team comprises only of the master thread, so there is no parallelism at all. They need to be inside the `parallel` block, like so:

```
#pragma omp parallel
{
  #pragma omp for
  for (...) {
    ...
  }
}
```

Alternatively, a following shorthand can be used:

```
#pragma omp parallel for
```

```
for (...) {
  ...
}
```

3) Data-access clauses

The `parallel` directive has a way to control which variables are shared between threads and which are unique (thread-local, i.e. each thread has its own copy). By default, all variables - except those declared inside the parallel block - are shared. This behaviour can be changed with the `private`, `firstprivate`, `lastprivate` and `shared` clauses.

Example:

```
int var1, var2 = 0;
#pragma omp parallel for private(var1) shared(var2)
for (var1 = 0; var1 < 50; var1++) {
  var2 += var1;
}
```

In this example, `var2` is shared among all threads. It's not exactly safe to use it like that - there may be race conditions between threads. We'd need to use some kind of synchronisation - more on that next week. In contrast to `var2`, `var1` has a thread-local value. It's not copied from the outer scope - it starts with a default value in each thread. To preserve its value from outside the `parallel` block, you need to use the `firstprivate` clause. Similarly, the `lastprivate` clause causes the value from the last task to be copied back to the original variable after the end of the `for`/`sections` block.

Additionally, there is also the `default` clause. Its main use is to verify if you remembered to include all variables in your private/shared clauses.

Example:

```
int var1, var2 = 0;
#pragma omp parallel default(none) shared(var2) {
  var2 += var1;
}
```

The above code won't compile - variable `var1` has unspecified access.

4) Exercises:

1. Hello world

OpenMP doesn't need to be installed - it's included with most C++ compilers. In order to determine the version of OpenMP supported by your installation of gcc, you can use the following command:

```
echo | cpp -fopenmp -dM | grep -i open
```

The result is a date in format YYYYMM, where YYYY is a year and MM a month in which your version of OpenMP was first published. Based on that you can find the specific version number (look it up here: http://www.openmp.org/specifications/).

Download the files for today's class. To compile the Hello World example, issue the following command:

```
g++ -fopenmp hello_world.cpp -o hello_world
```

Run the resulting executable

How many threads were executed? Why does the output look so strange?

To change the number of threads for OpenMP programs, set the environment variable OMP_NUM_THREADS like so:

```
export OMP_NUM_THREADS=4
```

Uncomment line 12. Try to execute the compiled program again and observe the output.

2. Matrix multiplication (1 point)

Modify the mat_mult.cpp file so that it contains a parallelised implementation of matrix multiplication. You need to fill in the matrixMultParallel function.

5) Additional info

By default, the `for` directive causes the loop to be split evenly. That is, if the team consists of 5 threads and the loop has 10 iterations, then each threads is responsible for handling 2 iterations. This behaviour can be customised with the `schedule` clause. There are 2 main scheduling types: `static` and `dynamic`. Static is the default type - upon entering the loop, each thread independently decides which chunk of the loop will be handled by it. The dynamic type has no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime library for an iteration number and handles it, then asks for the next available one, and so on. Furthermore, the chunk size can also be specified.

Example:

```
#pragma omp for schedule(dynamic, 3)
for (int i = 0; i < 10; i++) {
  cout << i << endl;
}
```

In the above example, each thread asks for at most 3 iterations at a time.

Additionally, a scheduling modifier can be added. The `monotonic` modifier makes chunks execute in an increasing iteration order, while the `nonmonotonic` modifier causes an unspecified order.

Example:

```
#pragma omp for schedule(nonmonotonic:dynamic)
for (int i = 0; i < 10; i++) {
  cout << i << endl;
}
```

If we want to parallelise a nested loop, we can use the `collapse` clause.

Example:

```
#pragma omp for collapse(2)
for (int i = 0; i < 10; i++) {
  for (int j = 0; j < 10; j++) {
    ...
  }
}
```

6) Sources

Your main source for learning OpenMP should be the official documentation

Ostatnia modyfikacja: piątek, 12 marca 2021, 18:18

Skontaktuj się z nami

Obserwuj nas

Skontaktuj się z pomocą techniczną

Jesteś zalogowany(a) jako Stanisław Bitner (Wyloguj)

Podsumowanie zasad przechowywania danych

Pobierz aplikację mobilną

Pobierz aplikację mobilną

GET IT ON Google Play

Download on the App Store