Introduction to OpenMP

Christian Terboven, Paul Kapinos
IT Center, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, kapinos}@itc.rwth-aachen.de

Abstract

This document guides you through the exercises. Please follow the instructions given during the training session.

The prepared makefiles provide several targets to compile and execute the code:

- debug: The code is compiled with OpenMP enabled, still with full debug support.
- release: The code is compiled with OpenMP and several compiler optimizations enabled, should not be used for debugging.
- run: Execute the compiled code. The `OMP_NUM_THREADS` environment variable should be set in the calling shell.
- clean: Clean any existing build files.

The provided *.slurm is a batch script that you can use to submit a batch job to run on a Perlmutter compute node. Submit the job via command “sbatch *.slurm”, and check the output file after it is run.

1 Hello World

Go to the `hello` directory. Compile the `hello` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

Exercise 1: Change the code that (a) the thread number (*thread id*) and (b) the total number of threads in the *team* are printed. Re-compile and execute the code in order to verify your changes.

C/C++: In order to print a decimal number, use the `%d` format specifier with `printf()`:

```
int i1 = value;
int i2 = other_value;
printf("Value of i1 is: %d, and i2 is: %d", i1, i2);
```

Exercise 2: In which order did you expect the threads to print out the Hello World message? Did your expectations meet your observations? If not, is that wrong?

2 Parallelization of Pi (numerical integration)

Go to the `pi` directory. This code computes Pi via numerical integration. Compile the `pi` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where *procs* denotes the number of threads to be used.

Exercise 1: Parallelize the Pi code with OpenMP. The compute intensive part resides in one single loop in the `CalcPi()` function, hence the *parallel region* should be placed there as well. Re-compile and execute the code in order to verify your changes.

Note: Make sure that your code does not contain any data race – that is two threads accessing the same shared variable without proper synchronization and at least one of those accesses is for writing.

Exercise 2: If you work on a multicore system (e.g. NERSC Perlmutter), measure the speedup and the efficiency of the parallel Pi program.

# Threads	Runtime [sec]	Speedup	Efficiency
1			
2			
3			
4			
6			
8			
12			

3 Parallelization of an iterative Jacobi Solver

Go to the `jacobi` directory. Compile the `jacobi.c` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

Exercise 1: Parallelize the three compute-intensive program parts with OpenMP. For a simple start, create one *parallel region* for each performance hotspot.

Exercise 2: Try to combine *parallel regions* that are in the same routine into one *parallel region*.

4 Reasoning about Work-Distribution

Go to the `for` directory. Compile the `for` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where *procs* denotes the number of threads to be used.

Exercise 1: Examine the code and think about where to put the parallelization directive(s).

Exercise 2: Measure the speedup and the efficiency of the parallelized code. How good does the code scale and which scaling did you expect?

# Threads	Runtime [sec]	Speedup	Efficiency
1			

Is this what you expected?

5 Min/Max-Reduction in C/C++

Go to the `minmaxreduction` directory. Compile the `MinMaxReduction` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

Exercise 1: Since OpenMP 3.1 a reduction operation for min/max is supported. Add the necessary code to compute `dMin` and `dMax` (as denoted in lines 49 and 50) in parallel.