Introduction to OpenMP

Christian Terboven, Dirk Schmidl, Paul Kapinos
Center for Computing and Communication, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, schmidl, kapinos}@rz.rwth-aachen.de

Abstract

This document guides you through the hands-on examples and the exercises. Please follow the instructions given during the lecture/exercise session on how to login to the cluster. You can find this document in PDF version as well as an archive containing all files required to work on the following tasks at the event website: www.rz.rwth-aachen.de/go/id/ttt, navigate to *Material*.

Linux: Please download the `(F)ex_omp_ppces.tar(.gz)` archive and execute the following command to extract the exercises to your `$HOME` directory:

```
C/C++ exercises:
cd $HOME
cp Downloads/ex_omp_ppces.tar.gz .
tar -xzvf ex_omp_ppces.tar.gz

Fortran exercises:
cd $HOME
cp Downloads/Fex_omp_ppces.tar.gz .
tar -xzvf Fex_omp_ppces.tar.gz
```

If you need help or have any question please do not hesitate to ask.

Linux: 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.

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 writing to the same shared variable without proper synchronization.

Exercise 2: If you work on a multicore system (e.g. the cluster at RWTH Aachen University) measure the speedup and the efficiency of the parallel Pi program.

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

3 First steps with Tasks: Fibonacci and two small code snippets

During these two exercises you will examine the new Tasking feature of OpenMP 3.0.

Exercise 1: Go to the `fibonacci` directory. This code computes the Fibonacci number using a recursive approach – which is not optimal from a performance point of view, but well-suited for this exercise.

Examine the `fibonacci` code. Parallelize the code by using the Task concept of OpenMP 3.0. Remember: The *Parallel Region* should reside in main() and the fib() function should be entered the first time with one thread only. You can compile the code via '`gmake [debug|release]`'.

Exercise 2: The code below performs a traversal of a dynamic list and for each list element the process() function is called. The for-loop continues until `e->next` points to null. Such a loop could not be parallelized in OpenMP so far, as the number if loop iterations (= list elements) could not be computed. Parallelize this code using the Task concept of OpenMP 3.0. State the scope of each variable explicitly.

```
List l;
Element e;




for(e = l->first; e; e = e->next)
{


   process(e);

}
```

4 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: Neither OpenMP 2.5 nor OpenMP 3.0 provide min/max reduction operations in C/C++ (but OpenMP 3.1 does). The task of this exercise is to implement this functionality manually. Add the necessary code to compute `dMin` and `dMax` (as denoted in lines 29 and 30) in parallel. Think about various options and select the one delivering the best performance / scalability.

5 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?

Exercise 3: After hearing about the Intel VTune Amplifier XE you can perform an analysis to investigate your parallel code for possible load imbalances.

Linux: Load the appropriate module ('`module load intelvtune`'). You than have to start the GUI with '`amplxe-gui`'. When you finished using the Amplifier return to your previous session ('`exit`') for the other exercises.

To create a project the following steps are required after you opened the GUI:

1. Click File -> New -> Project
2. Choose any name you like and a directory to store the project and click '`create project`'
3. Select "for.exe" as application out of your example directory with the browse button.
4. Click the '`new analysis`' button ▷, choose '`Locks and Waits`' and click Start to start the analysis.
5. Change to the timeline view and investigate how long different threads are waiting.
   Can you see a load imbalance?

6 Finding Data Races: Primes

Go to the `primes` directory. Compile the `PrimeOpenMP` 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: Execute the program twice, with a given number of threads (at least two). You will find that the number of primes found in the specified interval will change - which of course is not the correct result. Try to find the Data Race by looking at the source code …

Exercise 2: Use the Intel Inspector XE (this tool has not been taught during PPCES 2013, but instructions how to use it are below, and instructors are ready to help you) to find the datarace. In order to not wait for the analysis result too long, shorten the search interval. The interval is provided as arguments to the program. The input arguments need to be specified in the Inspector project. *Note*: To use the Inspector, you have to load the appropriate module (‘`module load intelixe`’), but you do **not** have to switch to a different machine. Set OMP_NUM_THREADS to at least 2 (‘`export OMP_NUM_THREADS=2`’) and start the GUI with ‘`inspxe-gui`’.

Linux: The following steps are needed to check for dataraces.

1. Click “`File -> New -> Project`”
2. Enter any name you like and chose a location to store the result data.
3. Choose “PrimeOpenMP.exe” as application in your example directory by using the “`browse`” button.
4. As application argument specify a small search interval (e.g. “0 1000”).
5. Click the “`new analysis`” button ▶.
6. Choose “Locate Deadlocks and Data Races” as analysis type and press the ▶ Start button.

Exercise 3: Correct the `PrimeOpenMP` code using appropriate OpenMP synchronization constructs. Use the Inspector XE to verify that you have eliminated all Data Races.

Exercise 4: What are the limitations of Data Race detection tools like the Intel Inspector XE?

Exercise 5: Can you image why program verification at compile time can only be very limited and why it cannot detect the issues the thread checking tools are able to report?

7 From MPI and OpenMP to *Hybrid Parallelization*

You learned about MPI to program for Clusters, i.e. Distributed-Memory systems, and you learned about OpenMP to program for Multicore / Multisocket Systems, i.e. Shared-Memory systems. Of course we can combine both in one program, which is then called Hybrid Parallelization. Go to the `jacobi_hybrid` directory. Compile the `jacobi.c` code via ‘`make [debug|release]`’ and execute the resulting executable via ‘`OMP_NUM_THREADS=procs2 make MPI_PROCS=procs1 run`’, where procs1 and procs2 denotes the number of processes and/or threads to be used.

Exercise 1: First, parallelize this code with OpenMP. Second, parallelize this code with MPI by applying the following hints:

- Only the process with rank 0 should be responsible for the I/O parts of the program.
- The parallelization, i.e. the work distribution among the MPI processes, should be implemented in the `jacobi()` function, side-by-side with the OpenMP parallelization.

Exercise 2: Extend the program by collecting the runtime and performance information (in MFLOPS) per process and per thread and print the summary at the end of the program.

8 OpenMP Puzzles

The following declarations and definitions occur in all exercises before the Parallel Region:

```
int i;
double A[N] = { ... }, B[N] = { ... }, C[N], D[N];
const double c = ...;
const double x = ...;
double y;
```

Exercise 1: Insert missing OpenMP directives to parallelize this loop:

```
for (i = 0; i < N; i++)
{
   y = sqrt(A[i]);
   D[i] = y + A[i] / (x * x);
}
```

Exercise 2: Insert missing OpenMP directives to make both loops run in parallel:

```
#pragma omp parallel
{

for (i =                    ; i < N; i +=                     )
{
   D[i] = x * A[i] + x * B[i];
}

#pragma omp for
for (i = 0; i < N; i++)
{
   C[i] = c * D[i];
}

} // end omp parallel
```

Exercise 3: Can you parallelize this loop – if yes how, if not why?

```
#pragma omp parallel for
for (int i = 1; i < N; i++)
{
   A[i] = B[i] - A[i - 1];
}
```