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 lecture/exercise session on how to login to the cluster.

The .tar archive containing the C++ exercises: **https://t1p.de/2025-ppces-openmpx-cpp**
The .tar archive containing the Fortran exercises: **https://t1p.de/2025-ppces-openmpx-fortran**

Linux: Please download the corresponding archive and extract it to your `$HOME` directory.

C++:
wget https://terboven.site/index.php/s/Acq2mqBZs2yRdHL/download/cpp-exercises.tar.gz

Fortran:
wget https://terboven.site/index.php/s/ZMJwC7ekMjaYYWo/download/fortran-exercises.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 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. 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 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 (optional - only if you are already familiar with the tool): Use the Intel VTune Profiler XE to find the compute-intensive program parts of the Jacobi solver. There should be three performance hotspots in the program (depending on the input dataset):

Number	Line Number	Function Name	Runtime Percentage
1			
2			
3			

Exercise 2: Parallelize the compute-intensive program parts with OpenMP. For a simple start, create one *parallel region* for each performance hotspot.

Exercise 3: Try to combine *parallel regions* that are in the same routine into one *parallel region*.

Exercise 4: If you are working on a NUMA machine, think about the data distribution of the jacobi code. Change the data initialization for a better data distribution if needed. If you wish, you can also parallelize the error check as well.

4 First steps with Tasks: Fibonacci

During this exercise you will examine the Tasking feature introduced in 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 '`make [debug|release]`'.

(optional) During the presentation you have heard that creating tasks after a certain roadblock is inefficient. Implement that idea into you code and stop creating new tasks when n is smaller than 30. For the last fibonacci numbers write a separate function which continues to compute in serial execution.

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?

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

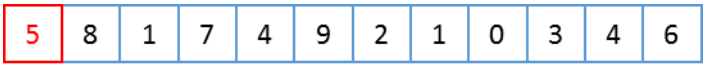
Go to the `minmaxreduction` directory. Compile the `MinMaxReduction` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

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 29 and 30) in parallel.

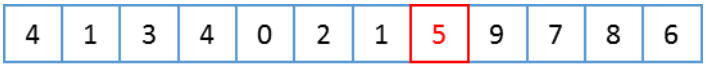
7 Quicksort

Quicksort is a recursive algorithm which, in this case, is used to sort an array of random integer numbers. How it works is described in the following steps.

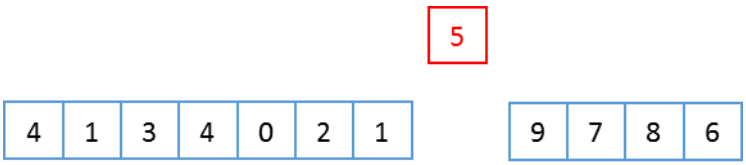
A pivot element is chosen. The value of this element is the point where the array is split in this recursion level.



All values smaller than the pivot element are moved to the front of the array, all elements larger than the pivot element to the end of the array. The pivot element is between both parts. Note, depending on the pivot element the partitions may differ in size.



Both partitions are sorted separately by recursive calls to quicksort.



The recursion ends, when the array reaches a size of 1, because one element is always sorted.

Go to the `quicksort` directory. Compile the `Quicksort` 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: The partitions created in step 3 can be sorted independent from each other, so this could be done in parallel. Use OpenMP Tasks to parallelize the quicksort program.

Exercise 2: Creating tasks for very small partitions is inefficient. Implement a cut-off to create tasks only if enough work is left. E.g. when more than 10k numbers have to be sorted, a task can be created, for smaller arrays no task is created.

Hint: You can add if clauses to the task pragmas.

Exercise 3: The if clause needs to be evaluated every time the function is called, although the array size does not exceed 10k elements on a lower level. Implement a serial_quicksort function and call this function when the array gets too small. This can help to avoid the overhead of the if clause.

8.1 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 will be covered on Tuesday of PPCES 2016, 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 Inspector`'), 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 parameters specify a small search interval (e.g. "0 1000") and leave the dialog window by pressing "ok".
5. Click the "`new analysis`" button  .
6. Choose "Threading Error Analysis > Locate Deadlocks and Data Races" as analysis type and press the 

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?

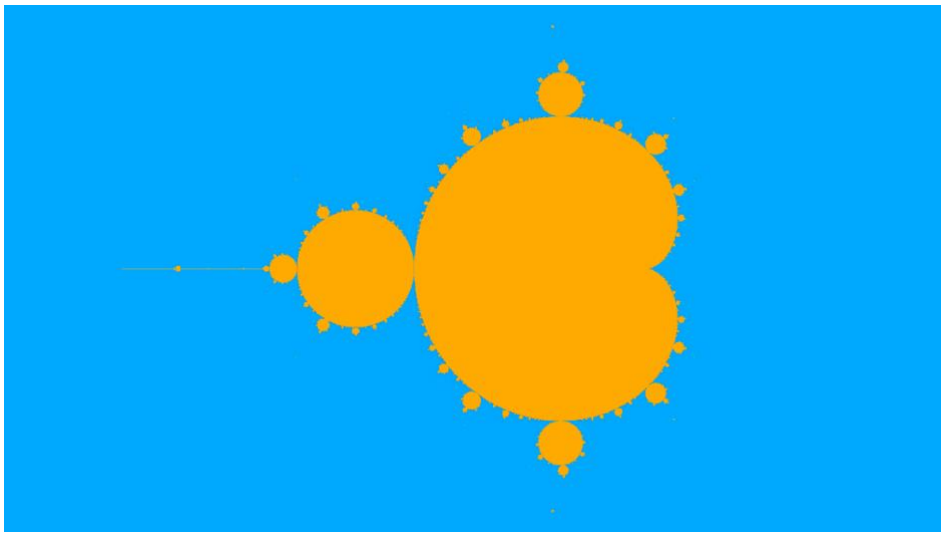
8.2 Finding Data Races: Pi

Go to the `datarace_detection` directory.

Exercise 1: Follow the manual on how to run a data race analysis with Intel Inspector and fix the errors in the source code.

9 Mandelbrot

The Mandelbrot set is a set of complex numbers that has a highly convoluted fractal boundary when plotted. The given code computes and plots the Mandelbrot set. The generated plot looks like this:



Go to the `mandelbrot` directory. Compile the `mandelbrot` 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 code with one thread and with multiple threads and compare the resulting pictures. Do they look as the picture above?

Exercise 2: One of the pictures is incorrect. Do you have an idea what is going wrong? Do you know a tool which can help you to find the error? Try to detect and fix the error in the code.