

# Introduction to OpenMP

---

*Christian Terboven, Dirk Schmidl*

*Center for Computing and Communication, RWTH Aachen University*

*Seffenter Weg 23, 52074 Aachen, Germany*

*{terboven, schmidl}@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/ppces](http://www.rz.rwth-aachen.de/ppces), navigate to *Course Material*.

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

```
cd $HOME
cp Downloads/ex_omp_ppces.tar.gz .
tar -xzf ex_omp.tar.gz
```

**Windows:** Please download the exercises and store the archive to your `H:` drive. On our cluster this corresponds to the `$HOME` directory. The 7-Zip archive manager is available from the context menu: right click on the file, and then select 7-Zip -> Extract Here.

**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.

**Windows:** There is a Visual Studio 2010 solution for every coding exercise. Since we use the Intel C/C++ compiler to profit from OpenMP 3.0 support, please proceed as follows to start Visual Studio 2010 after you have logged in to our cluster: Click Start -> All Programs -> Intel Parallel Studio 2010 XE -> Command Prompt -> and click IA-32 Visual Studio 2010 mode. Then enter `devenv` to start Visual Studio 2010.

## 1 Hello World

Go to the `hello` directory. Compile the `hello.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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**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.

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.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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**Exercise 1:** Parallelize the Pi code with OpenMP. The compute intensive part resides in one single loop in the `CalcPi()` function, thus 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. During the second lecture day we will see how to verify OpenMP programs automatically.

**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			

### 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.c` code. Parallelize the code by using the Task concept of OpenMP 3.0. Hint: 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]'`.

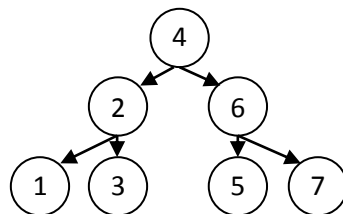
**Exercise 2:** The code below performs a traversal of a dynamic list and for each list element the `process()` function is called. The for-loop continues until `e->next` points to null. Such a loop could not be parallelized in OpenMP so far, as the number of loop iterations (= list elements) could not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

```

01 List l;
02 Element e;
03
04
05
06
07 for(e = l->first; e; e = e->next)
08 {
09
10
11     process(e);
12
13 }
```

### 4 Tree Traversal with tasks

Given the following tree:



**Exercise 1:** The code below performs a parallel depth-first tree traversal and for each element the `process()` function is called. Parallelize this code using the Task concept of OpenMP 3.0 and allow for *parallel* pre- and postorder traversal depending on the value of the post variable. State the scope of each variable explicitly. In a parallel preorder traversal, each node needs can be processed independently as long as all nodes between it and the root have already been processed. For postorder, all of a node's subtrees have to have already been processed for the node itself.

```

01 void traverse(node *p, bool post)
02 {
03     if ( post)
04     {
05
06     }
07
08     if (p->left)
09
10         traverse(p->left, post);
11
12     if (p->right)
13
14         traverse(p->right, post);
15
16     if ( post) /* postorder ! */
17     {
18
19     }
20
21     process(p);
22 }

```

## 5 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];
}
```

## 6 Dry Runs on Various Aspects

The code snippet below implements a Matrix times Vector (MxV) operation, where  $a$  is a vector of  $\mathbb{R}^m$ ,  $B$  is Matrix of  $\mathbb{R}^{m \times n}$  and  $c$  is a Vector of  $\mathbb{R}^n$ :  $a = B \cdot c$ .

```
01 void mxv_row(int m, int n, double *A, double *B, double *C)
02 {
03     int i, j;
04
05     for (i=0; i<m; i++)
06     {
07         A[i] = 0.0;
08         for (j=0; j<n; j++)
09             A[i] += B[i*n+j]*C[j];
10     }
11 }
```

**Exercise 1:** Parallelize the `for` loop in line 05 by providing the appropriate line in OpenMP. Which variables have to be private and which variables have to be shared?

**Exercise 2:** Which schedule would you propose for this parallelization? Explain your answer and briefly list the pros and cons for each of the three OpenMP worksharing schedules (namely *static*, *dynamic* and *guided*) for this specific case.

**Exercise 3:** Would it be possible to parallelize the `for` loop in line 09? If your answer is yes, provide the appropriate line in OpenMP and explain what scaling you would expect. If your answer is no, explain why you think it is not possible.

**Exercise 4:** If the code would be called as shown below, how would the parallelization look like (in line 05) and which variables would be private and shared? Provide the appropriate line in OpenMP and state for each variable (`m`, `n`, `A`, `B`, `C`, `i`, `j`) whether it is private or shared.

```

21 int m = ...;
22 int n = ...;
23 double* A = ...;
24 double* B = ...;
25 double* C = ...;
26
27 [ ... program logic here ... ]
28
29 #pragma omp parallel
30 {
31     mxv_row(m, n, A, B, C);
32

```

**Exercise 5:** In Exercise 8, the Fibonacci number `fib(n)` has been computed as:

```

fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2)

```

This algorithm can be transformed to an iterative approach, which typically is more efficient:

```

a = 0, b = 1
for (i = 2; i <= n; i++)
    c = a + b; a = b; b = c
fib(n) = b

```

Can this algorithm be parallelized (in OpenMP) as well? If you think so, provide a sketch of the parallelization. If you think not, explain why.

**Exercise 6:** Parallelize the following – recursive - Fibonacci algorithm with OpenMP 2.5, that means without using Tasks. The parallelization should be internal in this function, such that it could be called as a library routine.

```

01  int fib(int n)
02  {
03      int x, y;
04      if (n < 2) return n;
05      x = fib(n - 1);
06      y = fib(n - 2);
07      return x + y;
08  }

```

If you think that it might improve the scalability of your approach, you are free to restructure the code according to your needs, as long as the same interface is presented to the user.

## 7 Min/Max-Reduction in C/C++

Go to the `minmaxreduction` directory. Compile the `MinMaxReduction.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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**Exercise 1:** Neither OpenMP 2.5 nor OpenMP 3.0 provide min/max reduction operations in C/C++. 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.




## 8 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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**Exercise 1:** Use the VTune Amplifier 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):




**Linux:** To use the Amplifier login to `cluster-linux-tuning` (`'ssh cluster-linux-tuning'`) and load the appropriate modules on this machine (`'module switch intel intel/12.1.3.293; module load intelvtune'`). You then 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 "jacobi.exe" as application out of your example directory with the browse button.
4. Specify "< input" as application parameter to pass the input file to the application and click 'ok'.
5. Click the 'new analysis' button , choose 'Hotspots' and click  to start the analysis.
6. Use the  view to have a look at the time consuming functions.
7. Double clicking on a function opens the source code and shows the hotspots in the code.

### Windows:

To use the Amplifier you need to:

1. Click the the 'new analysis' button  in your Visual Studio toolbar.
2. Select "Hotspots" and click the  button.
3. Use the  view to have a look at the time consuming functions.
4. Double clicking on a function opens the source code and shows the hotspots in the code.

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*.

## 9 Thinking about Work-Distribution

Go to the `for` directory. Compile the `for.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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**Exercise 1:** Examine the code and think about where to put the parallelization directive(s). You may want to use the VTune Amplifier again to do a performance analysis of the code.

**Linux:** Create a project as before, but the application name is now "for.exe" and the arguments field can be left empty.



**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			

**Exercise 3:** You may want to use the VTune Amplifier to examine the (parallel) runtime behavior of the code to investigate why the scaling does not meet your expectation. Improve the scaling of the code, think about the scheduling of the worksharing construct(s).

*Note:* Choose the "locks and waits" analysis type and see in the timeline which threads have long waiting times.

## 10 Finding Data Races: Primes



Go to the `primes` directory. Compile the `PrimeOpenMP.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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**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 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.

Windows:

1. Click the  button and choose the Analysis Type “Threading Errors”.
  2. Mark “Where are the deadlocks and data races.” and press the “Run Analysis” button.
- How many Data Races are reported in how many program lines?
  - In how many variables do the Data Races occur? Name the variables.

**Exercise 3:** Correct the `PrimeOpenMP.c` 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 imagine 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?

## 11 Memory Access Pattern

The memory access pattern has an important influence on the performance.

**Exercise 1:** Which of the following code snippets do you think delivers the better performance (ignore that compilers may change the code in the course of the optimization)? Provide a reason for your answer.

Code snippet A:

```
double dLarge2dArray[dimension][dimension];
// some code to fill the array with meaningful data
for (int i = 0; i < dimension; i++)
{
    for (int j = 0; j < dimension; j++)
    {
        sum += dLarge2dArray[i][j];
    }
}
```

Code snippet B:

```
double * dLarge2dArray = malloc(dimension * dimension);
// some code to fill the array with meaningful data
for (int j = 0; j < dimension; j++)
{
    for (int i = 0; i < dimension; i++)
    {
        sum += dLarge2dArray[i][j];
    }
}
```

**Exercise 2:** Do you think the result may be different for other programming languages than C/C++?

## 12 Finding Data Races: Jacobi

Go to the `jacobi` directory. Compile the code and use the Inspector XE to check for dataraces.

**Exercise 1:** Correct the `jacobi.c` code using appropriate OpenMP synchronization or privatization constructs. Use the Inspector to verify that you have eliminated all Data Races.

**Linux:** Create the project as described for exercise 10, but choose the “`jacobi.exe`” as application and “`< input`” as application argument.

**Exercise 2:** 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. Compare the performance (either MFLOP/s or runtime) of both versions (each executed with two threads) over the performance of the code when the Inspector was used. Which performance ratio do you get? If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

## 13 Exploiting Architectures: STREAM

The STREAM memory benchmark is a widely used instrument for memory bandwidth measurements. It tries to measure the bandwidth that is actually available to a program, which typically differs from what a vendor specifies as the maximum memory bandwidth. The version provided for this exercise has been modified to use OpenMP and runs only one test case.

Please note that these exercises will only show the desired effects if you have logged in to `cluster-linux-nehalem.rz.rwth-aachen.de` via the following command:

```
ssh -Y -l your_TIM_id cluster-linux-nehalem.rz.rwth-aachen.de
```

On Linux, the `taskset` command can be used to restrict a process to a subset of all the cores in a system. The syntax that should be used in this exercise is as follows:

```
taskset -c cpu-list cmd [args]
```

where `cmd` denotes the program to be executed under the specified restriction. The program `cpuinfo` can be used to query the core numbering on a given machine (not available on all machines, not available in the virtual machine). The program `lstopo` can be used to get a graphical presentation of the machine architecture.

**Exercise 1:** Go to the `stream` directory. Take a look at the source code and examine the operation that is done in order to measure the performance of the memory subsystem.

**Exercise 2:** Compile the `stream.c` code via `'make release'` and execute the program via `'OMP_NUM_THREADS=2 taskset -c binding ./stream.exe'`, for each binding in the table below. With an array size of 10000000, each array is approximately 75mb in size. How do you explain the performance variations?

# Threads	Binding (no blanks in between)	SAXPYing [MB/s]
2	0,1	
2	0,2	
2	0,4	

**Exercise 3:** Compile the `stream.c` code via `'make release DIM=200000'` and execute the program via `'OMP_NUM_THREADS=2 taskset -c binding ./stream.exe'`, for each binding in the table below. How do you explain the performance difference from what you observed before?

# Threads	Binding (no blanks in between)	SAXPYing [MB/s]
2	0,1	
2	0,2	
2	0,4	

**Exercise 4:** Login to machine `cluster-linux-opteron` via `'ssh -Y cluster-linux-opteron'`. Compile the `stream.c` code via `'make release'` and execute the program via `'OMP_NUM_THREADS=2 taskset -c binding ./stream.exe'`, for each binding in the table below. How do you explain the performance difference from what you observed before? How can you improve the performance on this architecture?

# Threads	Binding (no blanks in between)	SAXPYing [MB/s]
2	0,1	
2	0,2	
2	0,4	

**Exercise 5:** Login to the ScaleMP machine (`'ssh linuxscale3'`) and start the stream benchmark on this machine. What is the best memory bandwidth you can get? Note: Remember what you have learned about thread binding.

## 14 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. If you are using Visual Studio, the environment variable `OMP_NUM_THREADS` to control the number of threads can be set by right-clicking on the project, selecting Properties and then under Debugging -> Environment.

**Exercise 1:** Re-examine the OpenMP parallelization developed in Exercise 4 (part 1). Now 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.