Introduction to OpenMP

Christian Terboven
Center for Computing and Communication, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
terboven@rz.rwth-aachen.de

Abstract

This document guides you through the hands-on examples and exercises. Please follow the instructions given during the lecture / exercise session on how to log in to the workstations and how to start the virtual machine, or how to log in to the Linux and / or Windows Cluster. You can run all hands-on examples and exercises either on your virtual machine or on the cluster at RWTH Aachen University. In case you are using the Linux cluster, please execute the following commands to download the exercises, extract them to your $HOME directory and initialize the environment for the use of the Sun Studio compilers:

```
cd $HOME
wget http://support.rz.rwth-aachen.de/public/openmp.ex.tar.gz
tar –xzvf openmp.ex.tar.gz
module load studio
```

In case you are using the Windows cluster, please copy the directory `openmp_exercises.vs2008` from the `P:\PPCES_2010` directory to your $HOME directory `H:\`.

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

The prepared makefiles provide several targets to compile and execute the code using the Sun Studio compilers on Linux:

- 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.

We also provide Visual Studio 2008 solution files using the Intel C/C++ compilers, since Visual Studio 2008 does not support the new features of OpenMP 3.0 and the Intel compiler integration is not yet available for Visual Studio 2010.

1 Hello World

Hands-on: Go to the `hello` directory. Compile the `hello.c` code via '`make [debug|release]`' or using the Visual Studio IDE and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', 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.

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]`' or the Visual Studio IDE and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', 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, 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 Parallelization of the Matrix- Multiplication

Please note: This exercise is available on Windows only. Exercise 4 of the *Windows-HPC and Visual Studio Basics* exercise sheet distributed on Monday included several versions of a Matrix-Multiplication code.

Exercise 1: Parallelize the Matrix-Multiplication with OpenMP. Compare the performance you can achieve with one, two, four and eight threads to the performance delivered by the Intel MKL version that has been created in Task 4.4 of that particular exercise sheet.

4 Thinking about Work-Distribution

Please note: Some parts of this exercise makes use of tools that will be presented in detail on Thursday, so you may want to skip these parts for now. You can work on this exercise already, but the tools will be there to help you solving these tasks. Go to the `for` directory. Compile the `for.c` code via '`make [debug|release]`' or the Visual Studio IDE and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', where procs denotes the number of threads to be used.

Exercise 1a: Examine the code and think about where to put the parallelization directive(s). On Linux you may want to use the Sun Studio Performance Analyzer to do a performance analysis of the code. Execute the performance analyzer like this: '`OMP_NUM_THREADS=procs collect for.exe`'. Then use the '`analyzer`' command to open the performance analysis result.

Exercise 1b: Examine the code and think about where to put the parallelization directive(s). On Windows you may want to use the Intel Parallel Amplifier to do a performance analysis of the code. Execute the Amplifier by selecting *Hotspots – Where is my program spending time?* and clicking the *Profile* button in the toolbar.

Exercise 2: If you work on a multicore system (e.g. the cluster at Aachen University) measure the speedup and the efficiency of the parallelized code. How good does the code scale and which scaling did you expect?

Exercise 3a: On Linux you may want to use the Sun Studio Performance Analyzer to examine the (parallel) runtime behavior of the code to investigate why the scaling does not meet your expectation. This time the Timeline View may give you an idea of why. Improve the scaling of the code, think about the scheduling of the Worksharing construct(s).

Exercise 3b: On Windows you may want to use the Intel Parallel Amplifier to examine the (parallel) runtime behavior of the code to investigate why the scaling does not meet your expectation. This time you have to select the Concurrency – Where is my Concurrency poor? Configuration and switch to the Thread-Function-Caller View to get an idea of why. Improve the scaling of the code, think about the scheduling of the Worksharing construct(s).

5 Parallelization of an iterative Jacobi Solver

Please note: Some parts of this exercise makes use of tools that will be presented in detail on Thursday, so you may want to skip these parts for now. You can work on this exercise already, but the tools will be there to help you solving these tasks. Go to the `jacobi_1` directory. Compile the `jacobi.c` code via '`make [debug|release]`' or the Visual Studio IDE and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', where procs denotes the number of threads to be used.

Exercise 1: Use the Sun Studio Performance Analyzer on Linux or the Intel Parallel Amplifier on Windows to find the compute-intensive program parts of the Jacobi solver.

Number	Line Number	Function Name	Runtime Percentage
1			
2			
3			

Execute the performance analyzer like this: '`collect jacobi.exe < input`'. Then use the '`analyzer`' command to open the performance analysis result. Execute the Amplifier by selecting *Hotspots – Where is my progam spending time?* and clicking the *Profile* button in the toolbar.

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*.

6 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];
}
```

7 Dry Runs on Various Aspects

The code snippet below implements a Matrix times Vector (MxV) operation, where a is a vector of R^m, B is Matrix of R^{mxn} and c is a Vector of 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 contras 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 08? 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  }
```

8 Finding Data Races: Primes

Please note: Some parts of this exercise makes use of tools that will be presented in detail on Thursday, so you may want to skip these parts for now. You can work on this exercise already, but the tools will be there to help you solving these tasks. Go to the `primes` directory. Compile the `PrimeOpenMP.c` code via '`make [debug|release]`' or using the Visual Studio IDE and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', 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 Sun Thread Analyzer on Linux or the Intel Parallel Inspector on Windows to find the Data Race in this code. Compile and run the `PrimeOpenMP.c` code via '`make check`' on Linux or select *Threading Errors* from the *Inspect* button on Windows. Note: In order to not wait for the analysis result too long, the search interval has been shortened in the Makefile on Linux, you should probably set the interval specified as the program argument to "0 1000" as well. Open the Thread Analyzer GUI program via '`tha tha.1.er`' or wait for the Intel Parallel Inspector to present analysis results.

- 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 Sun Thread Analyzer or the Intel Parallel Inspector to verify that you have eliminated all Data Races.

Exercise 4: What are the limitations of Data Race detection tools like the Sun Thread Analyzer or the Intel Parallel Inspector?

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?

9 Finding Data Races: Jacobi

Please note: Some parts of this exercise makes use of tools that will be presented in detail on Thursday, so you may want to skip these parts for now. You can work on this exercise already, but the tools will be there to help you solving these tasks. Go to the `jacobi_2` directory. Compile and run the `jacobi.c` code via '`make check`' or the Visual Studio IDE. Open the Thread Analyzer GUI program via '`tha tha.1.er`' or perform an analysis using the Intel Parallel Inspector.

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

Exercise 2: Compile the `jacobi.c` code via '`make [debug|release]`' or use the Visual Studio IDE and and execute the resulting executable. Don't forget to set '`OMP_NUM_THREADS=procs`', 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 Thread Analyzer was used. Which performance ratio do you get?

10 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]`' or the Visual Studio IDE.

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  }
```

Exercise 3: The code below performs a 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 pre- and postorder traversal depending on the value of the post variable. State the scope of each variable explicitly.

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

11 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 work on a cc-NUMA machine, so please follow the instructions given during the Lab.

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 `/home/hpc/bin/lstopo` can be used to get a graphical presentation of the machine architecture.

On Windows, the start command can be used to restrict a process to a subset of all the cores in a system. The system that should be used in the exercise is as follows:

```
set OMP_NUM_THREADS=2
start /B /wait /affinity <hex_affinity> cmd [args]
```

where `cmd` denotes the program to be executed under the specified restriction and `hex_affinity` denotes a bit-wise affinity mask as a hex number (bit = 1 → core is allowed for the program). The program `c:\Program Files (x86)\Intel\MPI\3.2.2.006\ia32\bin\cpuinfo.exe` can be used to query the core numbering on a given machine.

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`' or the Visual Studio IDE and execute the program via '`OMP_NUM_THREADS=2 taskset –c binding ./stream.exe`' or using the `start` command as described above, 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	

12 Dry Runs on Various Aspects

Exercise 1: In Exercise 8, the Fibonacci number fib(n) has been computed as:

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

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

```
a = 1, b = 1
for (i = 3; 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 2: 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.