Introduction to OpenMP

Bastian Küppers, Christian Terboven
IT Center, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
kueppers@itc.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.

Please download the exercises and save them in your virtual machine. Then extract the exercises with the command:

tar -xvf omp_exercises.tar

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

You can run the examples in two ways:

1. Open the *.c file and compile and run it
2. There are prepared Makefiles in the directories. Switch into the example directory in your shell and use the make targets described below.

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.

Exercise 1: Change the code that (a) the thread number (*thread id*) and (b) the total number of threads in the *team* are printed. 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 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 a 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. 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 cannot be parallelized in OpenMP with a *for* construct, as the number if loop iterations (= list elements) cannot 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 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 (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?

```
for (int i = 1; i < N; i++)
{
   A[i] = B[i] – A[i – 1];
}
```

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: Can the scaling be improved by different scheduling strategies? Try out different strategies to find the best one for this example.

6 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 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*.

Exercise 3: 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.

7 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: Correct the `PrimeOpenMP` code using appropriate OpenMP synchronization constructs.

8 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 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.