Technische Universität München
Chair of Scientific Computing
Dr. Ralf-Peter Mundani
Dipl.Inf. Atanas Atanasov

SS 2010

Parallel Programming and HPC Exercise Sheet 5: Programming with OpenMP

23.06.2010

1 Getting Started

1.1 How to Compile OpenMP Code

Today, most of the recent C-compilers are capable of generating multi-threaded executables from source code using OpenMP-directives. The usual way is to pass a command line option, which differs from vendor to vendor. Which option to use can be taken from the compiler documentation, e.g. the man-pages in UNIX environments. As an example, the correct flag for the *GNU C Compiler (GCC)* would be `-fopenmp`:

```
gcc -fopenmp -Wall -o example_omp example.c
```

Note: If the compiler supports OpenMP, and especially which standard (e.g. OpenMP 3.0), depends from its version. For example, the GCC supports OpenMP 2.5 from version 4.2 on and OpenMP 3.0 in version 4.4 or higher.

1.2 How to Handle Compiler Directives and Library Calls

Usually multi-threaded applications are written in a manner so that they can be run in both, sequential and parallel (e.g. on single-core machines or if no OpenMP-capable compiler is available). As you have already learned, there are two different constructs provided by the OpenMP interface: compiler directives and runtime routines.

The compiler-directives are used via the `#pragma` keyword, which is an ANSI-C preprocessor directive. Thus, if a non-OpenMP compiler is used, OpenMP-specific pragmas are simply ignored and do not have any influence on the sequential executable.
However, calls to OpenMP routines are not ignored by such a compiler. This usually ends up in linker errors, since their symbol name can not be resolved due to the missing object file (the `-fopenmp` flag adds the library path to the directory list of GCC). To prevent those problems,

the #ifdef directive is usually used in combination with the environment-variable _OPENMP, which is set by the compiler when compiling OpenMP code. See how this construct is used in the following section to make a program executable both single- and multithreaded.

1.3 Sample Program

The listing below shows a simple C program using OpenMP:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #ifdef _OPENMP
5  #include <omp.h>
6  #endif /* _OPENMP */
7
8
9  int main(){
10 #ifdef _OPENMP
11   printf("Total number of processes: %d\n", omp_get_num_procs());
12 #pragma omp parallel
13   {
14     printf("Process %d\n", omp_get_thread_num());
15   }
16 #else
17   printf("No OpenMP, only one thread\n");
18 #endif
19   return EXIT_SUCCESS;
20 }
```

2 Compile and Run an OpenMP application

a) Implement and compile the in 1.3 given source code. Generate a single-threaded and a multi-threaded executable and run it on a multicore processor. Does the output match your expectations?

b) There is a routine `int omp_get_num_threads()` which returns the number of currently running threads. Call this routine before and after the `parallel` directive (line 12) and print its results to the standard output. How do these numbers compare with the number of running processes (line 11)?

3 Numerical Computation of π - Monte Carlo

There are numerous ways to compute π numerically. One common way is a geometrical approach: The unit circle has an area of exactly π. Imagine a unit circle drawn in a Cartesian plane around the origin. Assume there is a unit square in the first quadrant, with corners at $\{(0,0),(0,1),(1,0),(1,1)\}$. It is easy to see that one quarter of the unit circle in the first quadrant lies perfectly inside the unit square.

Let the area of this quarter be A_c. It can be approximated in the following way: A random number generator generates n coordinates (x, y) which lie inside the unit square and are equally distributed.
A generated coordinate can also be interpreted as a vector $\vec{v} \in \mathbb{R}^2$ with the origin as initial point. $\vec{v}$ lies inside the unit circle if its magnitude $m_{\vec{v}} \leq 1$, otherwise it is outside. $m_{\vec{v}}$ is the norm of $\vec{v}$ and is defined as:

$$m_{\vec{v}} = \|\vec{v}\| = \sqrt{x^2 + y^2}$$

Let n_c be the number of points which lie inside the quarter of the unit circle. If n is high enough, its area can be approximated by:

$$A_c \approx \frac{n_c}{n}$$

We also know that

$$A_c = \frac{\pi}{4}$$

and can thus say:

$$\pi \approx 4\frac{n_c}{n} \quad \text{for} \quad n \gg 1$$

a) Write a sequential program in C which computes π numerically in the above described way.

b) Use the OpenMP compiler directive `for` to parallelise your sequential code. Where can you find race conditions? What are the consequences?

c) Use the OpenMP compiler directive `critical` for synchronisation to prevent these race conditions.

d) Make a second parallel implementation and use the directive `reduction` instead of `critical`. Measure the runtime of all implementations (including the sequential one) and compare them to each other. Interpret your results.

Hints

- The OpenMP runtime library provides the routine `double omp_get_wtime()` for accurate time measurement.
- You can use the function `int rand_r(unsigned int * seedptr);` from the C Standard Library as random number generator.

4 Numerical Computation of π - Through numerical integration

In the previous task we have seen a Monte-Carlo method to compute an aproximation of π. Another approximation can derived from the following equation:

$$\int_0^1 \frac{4}{1+x^2}\,dx = [4\arctan(x)]_0^1$$

- Find a suitable numerical approach to approximate the given equation!
- Create a serial and parallel version(via OpenMP) of the chosen approach!
- Compare the effieciency and the convergence behaiviour of the given approach and the Monte-Carlo algorithm from the previous exercise