Introduction to High-Performance Computing, Winter 2019

Exercise 4: OpenMP

Preparation
Please follow the instructions given during the lecture and the first exercise on how to log into the cluster. Download the provided archive and extract the tar-ball in your current directory:
`$ tar -xvf <archive>`
The archive contains subfolders for the various problems of this exercise. Each of them contains a code template and a makefile providing the following options:

1. debug: The code is compiled with OpenMP and full debug support.
2. release: The code is compiled with OpenMP and several compiler optimizations. Use this version for benchmarking.
3. run: Execute the compiled code. The `OMP_NUM_THREADS` environment variable should be set in the calling shell to specify the number of threads to be executing the binary.
4. clean: Clean any existing build files.

Problem 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.

a) Change the code that the thread number (*thread id*) and 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);
```

b) 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?

Problem 2. 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.

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 `lstopo` can be used to get a graphical representation of the machine architecture.

a) 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 ($A = B + scalar \cdot C$). Parallelize this operation with OpenMP. Take care of producing correct time measurements.

b) Compile the `stream` code via '`make release`' and execute the program on login-t (or the mpi backend) via '`OMP_NUM_THREADS=2 taskset -c binding ./stream.exe`', for each binding in the table below. With an array size of 40,000,000, each array is approximately 300 MiB in size. How do you explain the performance variations?
Hint: Consider NUMA effects.

#Threads	Binding (no blanks in between)	SAXPYing [MiB/s]
2	0,1	
2	0,2	
2	0,12	
2	0,24	

Problem 3. First steps with OpenMP tasks
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 until the task directive was introduced, as the number of loop iterations (= list elements) could not be computed. Parallelize this code using the task concept of OpenMP. **State the scope of each variable explicitly.**

```
List l;
Element e;




for(e = l->first; e; e = e->next)
{


    process(e);

}
```

Problem 4. 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.

a) Examine the code and think about where to put the parallelization directive(s).

b) Measure the speedup and the efficiency of the parallelize code. How good does the code scale and which scaling did you expect?

#Threads	Runtime [sec]	Speedup	Efficiency
1			

Problem 5. OpenMP Puzzles
Try to parallelize the following loops by inserting the missing OpenMP directives. If a loop cannot be parallelized, state reasons why you think so.

Puzzle a)

```
double A[N] = { ... }, B[N] = { ... }, D[N];
const double c = ...;
const double x = ...;
double y;




for (int i = 0; i < N; i++ )
{


    y = sqrt(A[i]);


    D[i] = y + A[i] / (x*x);
}
```

Puzzle b)

```
double A[N] = { ... }, B[N] = { ... }, D[N];
const double c = ...;
const double x = ...;
double y;




for (int i = 1; i < N; i++ )
{


    A[i] = B[i] - A[i - 1];
}
```

Problem 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 a 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
06
07     for (i = 0; i < m; i++)
08     {
09         A[i] = 0.0;
10
11
12
13         for (j = 0; j < n; j++)
14         {
15             A[i] += B[i * n + j] * C[j];
16         }
17     }
18 }
```

a) Parallelize the `for` loop in line 07 by providing the appropriate OpenMP pragmas. Which variables have to be private and which variables have to be shared?

b) 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.

c) Would it be possible to parallelize the `for` loop in line 13? 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.

Discussion of this exercise on January 7, 2020.