OpenMP: Hands-on - Part 1

Richard Berger

February 14, 2018

Exercise 1: Hello OpenMP!

1. Create a multi-threaded version of the famous "Hello World!" Program. Compile using `g++` and with the `-fopenmp` flag.
2. Run you program by submitting the job script `job.sh`

   ```
   sbatch job.sh
   ```

3. Wait until your job has completed. `.out` and `.err` files should appear once your job completes. Look at the output of these files.
4. Modify `job.sh` and use different values for `OMP_NUM_THREADS` (2, 4, 8, 16, 28). Resubmit with `sbatch`
5. Observe how the output changes with different values of `OMP_NUM_THREADS`
6. Is the output deterministic? Will running the code multiple time always show the same output?

Listing 1: Example Output

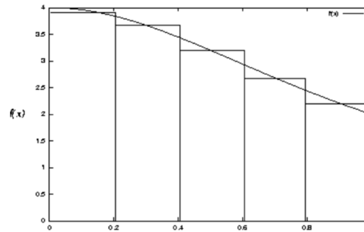
```
Hello from thread 1/4!
Hello from thread 2/4!
Hello from thread 0/4!
Hello from thread 3/4!
```

Exercise 2: Approximate π

π can be approximated using the following formula:

$$\pi = 4\int_0^1 \frac{1}{1+x^2}\mathrm{d}x \approx 4\sum_{i=0}^{N-1} hf(x_{i+\frac{1}{2}})$$

where $h = \frac{1}{N}$ and $f(x) = \frac{1}{1+x^2}$. The integral is approximated by slices of $h \cdot f(x)$ at midpoints. Use $N = 100000000$.



1. Implement the approximation in serial and then parallelize the code using OpenMP
2. Measure the time it takes to compute the sum using `omp_get_wtime()` and output the duration.
3. Create a job script which runs the program with 1, 2, 4, 8, 14, 16, and 28 threads. Submit the script using sbatch. The output of your completed job should be similar to this:

```
M_PI:        3.14159265358979312
 1 threads: 3.14159265359042639 (duration: 0.452553s)
 2 threads: 3.14159265358990991 (duration: 0.226431s)
 4 threads: 3.14159265358968254 (duration: 0.113476s)
 8 threads: 3.14159265358981576 (duration: 0.056814s)
14 threads: 3.14159265358983664 (duration: 0.032638s)
16 threads: 3.14159265358988238 (duration: 0.028629s)
28 threads: 3.14159265358983175 (duration: 0.032850s)
```

4. Try out using a critical section, atomic and reduction to protect the summation variable. Observe the scalability of each variant. Please note that you might have to increase the time in your job script and wait for your jobs to complete with `squeue -u $USER`.
5. (Extra): use **`long double`** as summation variable and notice the change of accuracy.

Exercise 3: OpenMP Loop Schedules

In this exercise you will create a visualization of all different OpenMP schedules.

1. First we create an array of size 80 and iterate over all elements in a loop. In the loop we save the current thread index in the array. Here is the serial version:

```
const int N = 80;
int a[N];
int thread_id = 0;
int nthreads = 1;

for(int i = 0; i < N; ++i) {
    a[i] = thread_id;
}
```

2. Use the `print_usage(a, N, nthreads)` function (provided in the example folder) to visualize the result. It takes the array a as first parameter, its size N as second parameter and the total number of threads as *nthreads*. When running only on one thread, this is the output:

Listing 2: Example Output with 1 thread

```
serial:

0: ********************************************************************************
```

3. Parallelize the serial program using different schedules and use `print_usage()` to visualize how work is being distributed:

   - static
   - static, with chunk size 1
   - static, with chunk size 4
   - dynamic
   - dynamic, with chunk size 1
   - dynamic, with chunk size 4
   - guided
   - guided, with chunk size 1
   - guided, with chunk size 4

   Write this program with only one **`omp parallel`** region, multiple **`omp for`**, and use **`omp`** `single` blocks for printing the output. Only one thread should be using `printf`!

4. Write a job script and run your code on the Abacus cluster using 4 threads.

Listing 3: Example Output with 4 threads and static schedule

```
schedule(static):

0: ********************
1:                     ********************
2:                                         ********************
3:                                                             ********************
```

Exercise 4: Critical Section / Atomic / False Sharing

1. Write a program that generates a large array (1024*1024*1024) of random integer numbers from 0-99

```
// requires stdlib.h and time.h headers

// initialize RNG
srand(time(0));

for(int i = 0; i < N; i++) {
    // create integer from 0 - 99
    a[i] = rand() % 100;
}
```

2. Afterwards count how many times each number occurs and store these counters in an array of size 100.
3. Print out how many times each number occurs

```
0:   100
1:    89
2:    92
3:    81
...
99:   90
```

(extra) create a histrogram, by printing # relative to the count of each number

```
0:   #########
1:   ########
2:   ######
3:   ##########
...
99:  ##########
```

4. Write a parallel version of this program. Protect the counter array using a `critical` section.
5. Add timing information to your application using `omp_get_wtime()`. Write a job script to run your code on the cluster using 1,2,4,8 and 14, 16 and 28 threads.
6. Create a copy of your code and replace the critical section with `omp atomic`.
7. Run your atomic version of the code and compare the run times with the critical section implementation.
8. (Extra) Write a version of this code which gives each thread its own private counter array. After completion, collect the results of all thread arrays into the global counter array. This is called array reduction. Measure the speedup compared to atomics.

Exercise 5: Matrix Multiplication

1. Create an OpenMP parallelization of the matrix multiplcation code. Use matrices of at least size 2048x2048.
2. Run your application on the cluster using 1,2,4,8,14,16, and 28 cores. How well does your parallelization scale?
3. Use the MKL BLAS library to implement the matrix multiplication. Use the Intel compiler with the `-mkl=parallel` option to enable OpenMP support. Run your application on the cluster using 1,2,4,8,14,16, and 28 cores.