Assignment - Programming with OpenMP

Submit Assignment

Due Tuesday by 11:59pm **Points** 1 **Submitting** a file upload **File Types** pdf

Overview

This assignment consists of three topics: improving memory bandwidth with threads, eliminating false sharing and parallelizing applications. For each exercise compile a short report according to instructions and submit your code according to a Github repo. Refer to **Assignment - Submission Template** on how to obtain the assignment templates. The submission should be done by a group of two according to the signed up groups.

The assignments should be done using the GNU environment.

Exercise 1: STREAM benchmark and the importance of threads

STREAM is a benchmark to measure the sustained memory bandwidth of a platform. This exercise studies the impact of using different number of threads and schedules when measuring the memory bandwidth on a Beskow node. Download the code **stream.c** and compile with the following:

```
cc -fopenmp -O -o stream.out stream.c
```

Run the program on the compute node and an output like this will be produced.

```
-------------------------------------------------------------
Number of Threads requested = 4
Number of Threads counted = 4
-------------------------------------------------------------
...
-------------------------------------------------------------
Function     Best Rate MB/s     Avg time     Min time     Max time
Copy:            11839.5        0.013682     0.013514     0.013894
```

Submission

Compile a report on the following:

1. Run the STREAM benchmark for five times and record the average values of bandwidth and its standard deviation for the copy kernel.
    1. Prepare a plot that compares the bandwidth using 1, 2, 4, 8, 16, 32 threads.
    2. Describe how the bandwidth measured with the copy kernel depends on the number of threads.
    3. Prepare a plot comparing the bandwidth measured with copy kernel with static, dynamic and guided scheduling using 32 threads. How do you set the scheduling algorithm in the STREAM code? and what is the fastest scheduling strategy? Describe and motivate.

Exercise 2: parallel for, race condition, atomic/critical and false sharing

In this exercise, you will parallelize a simple C code to sum up values in an array. Implement your code in **src/sum.c** and do not modify any other parts except for the specified functions. The code can be compiled with make and is already programmed to compute the average and standard deviation of the execution time for each implementation for you. Implement the code and compile a short report according to the instructions. We will use your sum.c and run across test cases by checking the answers returned by the functions.

When implementing the functions you can directly **use external variable "x" which points to an array and "size" specifies the size of the array**. The variables are initialized in **src/main.c**. For the experiments, **use a significantly large array size in order avoid noisy results**. **Use the same array size for different versions to allow comparison.**

Serial version

Run the serial version of the code repeat the tests 50 times. Report the execution time and standard deviation.

Parallel construct

Implement omp_sum() in **src/sum.c** to parallelize the serial version of the code using only the parallel construct. Run the code with 32 threads on Beskow for a number of times. Is the code correct? If not, why not?

Critical section

Implement omp_critical_sum() in **src/sum.c** and use what you have implemented in omp_sum(). Use the omp critical to protect the regions that might be updated by multiple threads concurrently.

Atomic update

implement omp_atomic_sum() in **src/sum.c** and use atomic to perform atomic update of sum.

Local array variable

Implement omp_local_sum() in **src/sum.c** and avoid using critical section or atomic update. Instead create an array to store partial sum by threads. The size of the array should be equal or larger than the number of threads used such as the following.

```
int sum[MAX_THREADS];
```

The threads only read and modify their own respective cell of the array and finally the global sum value is computed outside of the parallel region.

Padded local variable

Implement omp_padded_sum() in **src/sum.c** and use what you have implemented in omp_local_sum(). Use the padding technique to remove false sharing.

Private variable

Implement omp_private_sum() in **src/sum.c** by using private variables for accumulating partial sums instead of an array. The threads only read and modify their own respective variable, and finally the global sum is computed outside the computation of local sum and before the end of the parallel region.

Optimized

Implement omp_reduction_sum() in **src/sum.c** and use the reduction method for accumulation.

Submission

Submit your code in a Github repo with the exact same structure. Perform a performance study of all the versions by executing the code and repeat the tests 50 times to measure the execution time and standard deviation when running with 1, 2, 4, 8, 16, 32 threads. Show the results by a plot. Submit a report that details the performance study of the versions that you have developed.

Exercise 3: DFTW, The fastest DFT in the west

The FFTW is the most famous library to solve a Fourier Transform (FT) using the Fast Fourier Transform algorithm. The FFTW takes its names from the fact that it is the fastest FFT in the West if not in the world. The goal of this exercise is to develop the fastest DFT (Discrete Fourier Transform, a different algorithm to calculate FT) in the west by parallelizing the serial version of the code provided. We have developed a DFT in serial that calculates forward and inverse DFTs and timing the total time taken for the computation of the two DFTs. The computational core of the program is the DFT subroutine that takes idft argument as 1 for direct DFT and -1 for inverse DFT. Implement your changes in **src/DFT.c** and do not modify any other parts except for the specified function. The code is already programmed to compute the average and standard deviation of the execution time for each implementation. We will use your DFT.c and run across test cases by checking the forward and backward DFT results by the function.

Submission

Submit your code in a Github repo with the exact same structure. Use the test case with size 8000 provided, measure the performance on Beskow using 1, 2, 4, 8, 16, 32 (oversubscribing) threads, report the average values and standard deviation of execution time. Submit a report that describe your changes with a plot showing the effect of changing the number of threads.

Exercise 4: N-Body simulation

N-Body simulation is a simulation of point particles interacting through gravitational force. Examples include the simulation of planetary movement in the solar system. By solving the Newton's law of gravity and Newton's law of motion, position of point particles are updated through a time stepping process:

1. Compute the force exerted on a particle by other particles through Newton's law of gravity.
2. Compute the acceleration of the particle in (x,y,z) directions through Newton's law of motion.
3. Update velocity then position of the particle with Forward Euler's method.
4. Repeat 1-3 for all particles.
5. Proceed to next time step and repeat 1-4

The algorithm is called direct N-Body simulation and has a complexity of O(N^2). In practice, the method consists of three loops. The first one being the time stepping loop; the second to iterate and update the position of all particles; the third one to compute the force exerted on a particular particle by all other particles. Below is an example of a simulation with 5000 particles that are randomly initialized in a cube between -1 and 1 in all directions.

(N = 5000)

Parallelize the code and introduce necessary changes in **src/nbody.c**. Ensure that the parallelized code produces correct result. The code is programed to output the execution time of each time steps.

Perform a execution time performance study on a Beskow computing node using 1, 2, 4, 8, 16, 32 (oversubscribing) threads and report the average and standard deviation of the execution time of the time steps. Repeat the test by varying N (the size of the simulation). Optionally visualize your simulation results. The results can be stored by using the flag on steps per checkpoint. **However it is important to <u>turn off I/O by not writing any outputs when performing benchmark</u> to avoid interference.**

Submission

Submit your code in a Github repo with the exact same structure. Submit a report that explains the changes that you have made and detailing the performance study of the versions that you have developed.

Exercise 5: Consumer-Producer (Optional)

Use OpenMP to implement a producer-consumer program in which some of the threads are producer and others are consumers. The producers read text from a collection of files, one per producer. They insert lines of text into a single shared queue. The consumers take the lines of text and tokenize them. Tokens are "words" separated by white space. When a consumer finds a token, it writes it to stdout. (*Pacheco, P. (2011). An introduction to parallel programming. Elsevier.)*

Submit by create a folder in the same level as other exercises called "exercise_5" and put your code there.