|  |
| --- |
| **Objective** |
| The objective of this project is to gain experience with OpenMP to develop parallel applications by developing a reasonably straightforward producer-consumer type application. |

# Starter Code

1. DataHelper.h and DataHelper.o: These two files provide the declaration and implementation for a simple class called DataHelper that has the following two public methods:
   1. The getData method: Obtain a number from a given input source. This method essentially simulates the process of reading data from a network-service. This method returns a number that can be analyzed via a call to the analyze method below.

NOTE: This method is not MT-Safe (\*not\* multi-threading safe => cannot be called from multiple threads) as it reads from a single source.

* 1. The analyze method: This method performs some given set of analysis on a supplied number (must have been obtained via call to getData) and returns a result to be displayed to the user.

NOTE: This method is indeed MT-Safe (multi-threading safe => can be safely called from multiple threads).

## Compiling:

This is a standard OpenMP program that does not require any additional libraries. Since this is similar to many of the programs already completed in this course, you are expected to figure out the necessary compiler command to be used.

# Parallelize the process method in program.cpp

## Description:

The objective of the project of the homework requires development of a suitably parallelized OpenMP-based implementation for the process method in program.cpp. You may assume that there will be at least 2 Open MP threads that will be used to run the parallel version.

Note that the parallelized version must preserve the following features form the sequential version:

1. The output from the parallel version must be exactly the same as those generated by the sequential version (assuming the command-line arguments are exactly the same).
2. The results from analyzing numbers (via call to the DataHelper::analyze method) must be displayed as soon as possible to the user (while preserving earlier requirement). In other words, your program must strive to achieve the lowest delay possibly in printing results to the user. Given this requirement, printing all the results at the end is not a valid solution to this homework.

## Testing

Verifying the second requirement (to achieve the lowest delay possible in printing results) has to be done visually. Ideally you will see outputs being printed at least as fast as the sequential version (if not faster) without much delay between successive outputs.

The first requirement for this project which is to ensure that the output from the sequential and parallel versions is identical and be automated using the Linux diff command as shown below:

|  |
| --- |
| $ ./program, 1000 5 > sequential\_output.txt  $ export OMP\_NUM\_THREADS=8  $ ./raodm\_Homework10 1000 5 > parallel\_output.txt  $ diff sequential\_output.txt parallel\_output.txt |

If the outputs from the sequential and parallel versions are identical then diff will not generate any outputs. However, if there are differences then diff will generate output containing the differences in the two versions. I will be using a similar approach when grading.

## Sample Outputs

Some sample outputs from the supplied sequential version and sample solution are shown below (note that the outputs from sequential and parallel runs are identical):

|  |  |
| --- | --- |
| **Output from supplied sequential version** | **Output from parallelized version** |
| [raodm@mualhpcp02 homework10]$ time ./Homework10 10 5  0: Processed 590011675. Result = 2314885530330529083  1: Processed 4394756061. Result = 2314885535208998909  2: Processed 10721860202. Result = 2314885540466571850  3: Processed 13056765960. Result = 2314885543871008808  4: Processed 17497028460. Result = 2314885548311271244  5: Processed 21645872112. Result = 2314885552460131280  6: Processed 26372315696. Result = 2314885556112833040  7: Processed 31027821721. Result = 2314885560768322745  8: Processed 35429717441. Result = 2314885565174429153  9: Processed 40574560045. Result = 2314885570315061005  real 0m1.277s  user 0m0.001s  sys 0m0.004s | [raodm@mualhpcp02 homework10]$ time ./Homework10 10 5  0: Processed 590011675. Result = 2314885530330529083  1: Processed 4394756061. Result = 2314885535208998909  2: Processed 10721860202. Result = 2314885540466571850  3: Processed 13056765960. Result = 2314885543871008808  4: Processed 17497028460. Result = 2314885548311271244  5: Processed 21645872112. Result = 2314885552460131280  6: Processed 26372315696. Result = 2314885556112833040  7: Processed 31027821721. Result = 2314885560768322745  8: Processed 35429717441. Result = 2314885565174429153  9: Processed 40574560045. Result = 2314885570315061005  real 0m0.459s  user 0m0.072s  sys 0m0.002s |

## Performance Analysis

Once you have tested and verified correct operation of the parallel program, develop a report containing a comprehensive performance analysis (using at least 5 sample runs with 95% confidence intervals) of your program using 2 through 8 threads and plot the three different sets of data using a suitable graphs that contrasts the theoretical vs. observed speedups. Your report must also include the following analyses:

* Using your observations prove that your parallel version is scalable.