CSE 5441
Programming Assignment 5
Submitted by: Biplob Biswas (biswas.102)

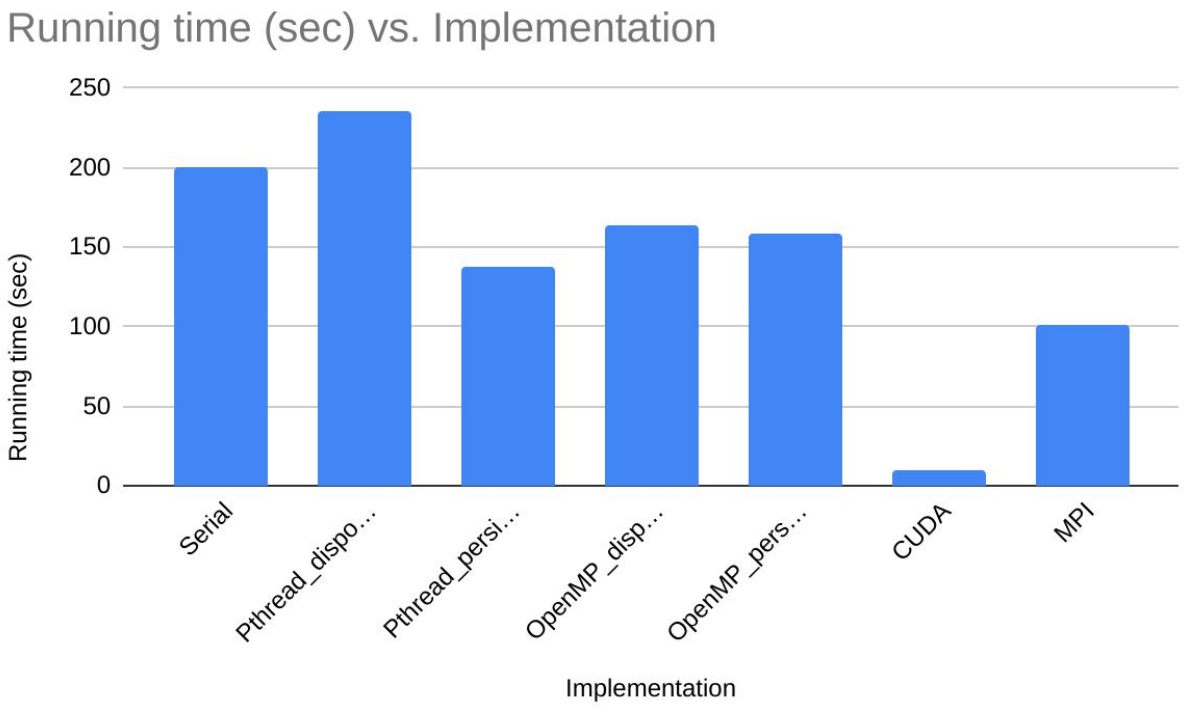
In this assignment, an MPI version of the simplified AMR program is implemented. A total of 5 MPI processes is used with a global communicator.

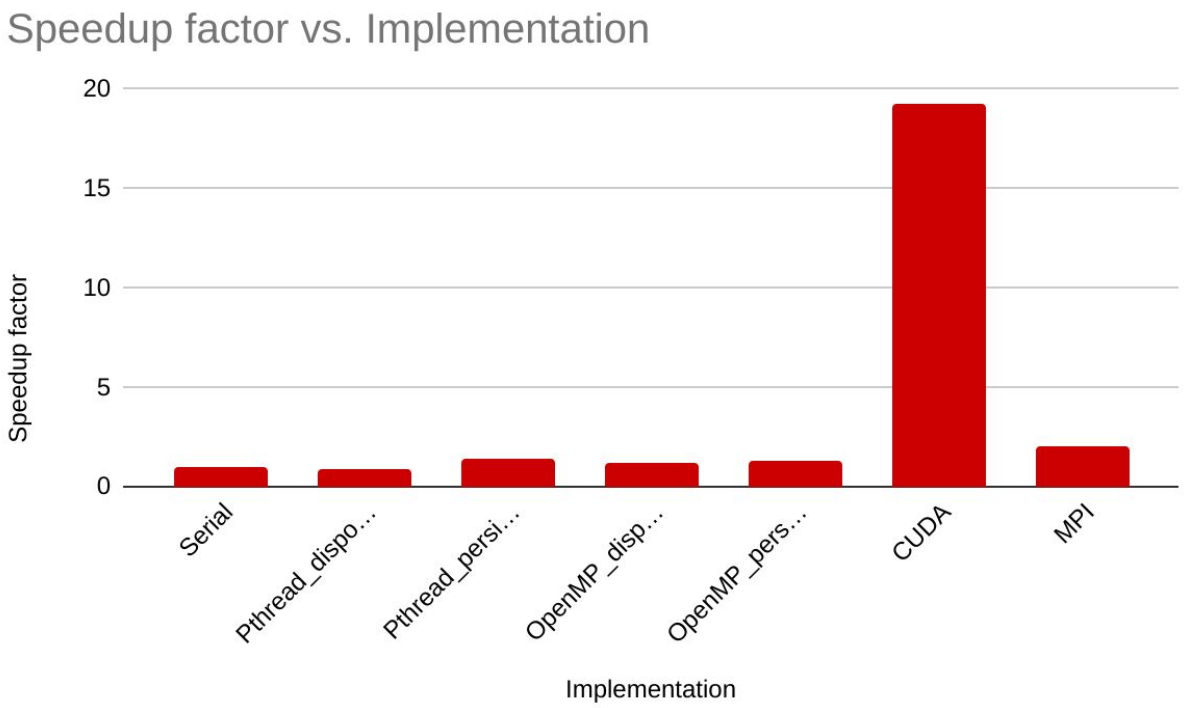
Workload distribution:
For this program, rank-0 MPI process works as a master MPI process. It takes the largest dataset 'testgrid_400_12206' as input, initializes DSVs and divides the input into 4 slices with little adjustment in the partition size for the last process as boxCount may not be divisible by 4 processes. The sliced input arrays include box perimeter, box neighbor Id, shared edge length etc. However, all the box temperature is not divided rather sent as a whole along with the starting box Id values for each process to locate which portion of the array to update. If the temperature array is splitted, neighbor box temperature may get placed into a different process, which would make the computation impossible. The splitted 4 input set is distributed using `MPI_Send` function to rank 1-4 MPI processes in 4 separate nodes of 28 cores each for updated DSV computation. Inside each node of the 4 slave processes, the temperature computation is further parallelized by using OpenMP 'for loop' of 28 threads. At the end of each iteration of updated DSV computation, the new DSVs are collected using `MPI_Recv` function and checked for convergence. If it converges the iteration stops and the master communicates with the slave MPI processes to stop by exploiting MPI tag field. Otherwise, the master continues to distribute the updated DSVs to the slave processes for further computation in the next iteration. The timings are measured using the `MPI_Wtime` function. (Detailed output at the end)

Runtime Results: For Input File 'testgrid_400_12206', Affect_rate 0.02, Epsilon 0.02, each converged after 794818 iterations.

Implementation	Best Configuration	Running time (sec)	Speedup factor
Serial	-	200.51	1x
Pthread	Disposable (8 threads)	236.14	0.85x
	Persistent (8 threads)	137.55	1.46x
OpenMP	Disposable (20 threads)	164.37	1.22x
	Persistent (24 threads)	158.72	1.26x
CUDA	96(Block) x 128(TPB)	10.39	19.29x
MPI	MPI(5 Processes) + OpenMP (28 disposable threads)	100.79	1.99x

Comparison of running time:





As can be seen from the above table and figures, CUDA offers the best performance in terms of execution time or performance which is 19 times faster than the serial version. And then MPI,

Pthread_persistent, OpenMP persistent, OpenMP_disposal has the performance in descending order. Interestingly, Pthread implementation with disposal threads is slower (0.85x) than all others including serial one, this is due to the overhead in creating Pthreads in each iteration.

Anomalies/Surprises:

Unlike other methods, I faced difficulties in reading a larger input file as standard input. So, I passed the input file name as the third argument of command line after AFFECT_RATE and EPSILON. And then read the file using freopen(file_name, 'r', stdin) function.

Upto lab 4, I used box as a structure for design advantages. But I find it quite difficult to transfer the structure to slave processes as the structure contains dynamically allocated memory of another structure containing box's neighbor information. As a result, I converted box attributes to separate arrays and splitted accordingly. Communicating complex user-defined struct type is a challenge in MPI.

The MPI implementation is almost two times faster than the serial one but not even close to the last CUDA implementation. I think it is more useful for distributing workload than ensuring parallelization. So it would be useful for running a program with a large workload on multiple hosts even if the hosts are low on resources.

Applicability of the various parallelization APIs:

I think threading is important and useful when there is a considerably large amount of data and we can run our desired operation on them independently.

In terms of execution time, CUDA has the best performance than others. It is good for operations like matrix multiplication. However, it requires costly GPU of NVidia and will not work in a heterogeneous system. Moreover, since all the data has to be copied to the device before an operation, it may not be a suitable option for an application with a large and complex unit of data.

Next best performer is MPI. It can play a good role in the heterogeneous system of different platforms. It is suitable for the data that can be divided into blocks and run operations on them independently.

With respect to the easiness of implementation, OpenMP is the simplest one. It is really useful as it requires very little pragma directive to convert a serial program into a parallel one. It also has a good parallelization mechanism for 'for' loop and 'reduction' operation. However, its disposable implementation performs poorly due to overhead.

In OpenMP, we can only request for a number of threads but can't be sure of having them. However, in Pthread, we can get an exact number of threads we specify. So, if we need such assurance, pthread suites best.

MPI output:

[biplob@o0119 cse5441_lab5]$ make

mpicc -O3 -fopenmp biswas_biplob_lab5_mpi.c -o lab5_mpi

[biplob@o0119 cse5441_lab5]$ mpirun -np 5 -ppn 1 ./lab5_mpi 0.02 0.02 testgrid_400_12206

Sending to process: 1 -> BoxCount: 3051, Starting BoxId: 0, Ending BoxId: 3050
Sending to process: 2 -> BoxCount: 3051, Starting BoxId: 3051, Ending BoxId: 6101
Sending to process: 3 -> BoxCount: 3051, Starting BoxId: 6102, Ending BoxId: 9152
Sending to process: 4 -> BoxCount: 3053, Starting BoxId: 9153, Ending BoxId: 12205

****
dissipation converged in 794818 iterations,
with max DSV = 0.0846372 and min DSV = 0.0829445
affect rate = 0.020000; epsilon = 0.020000

Time taken for MPI operation: 100798.503160 ms

****