ECE/CSC 506/406: Architecture of Parallel Computers

Project 1: OpenMP, MPI and Hybrid Parallel Programming

Version 1.2

Due Date: 16 October 2022, 11:59 PM

1. Objectives

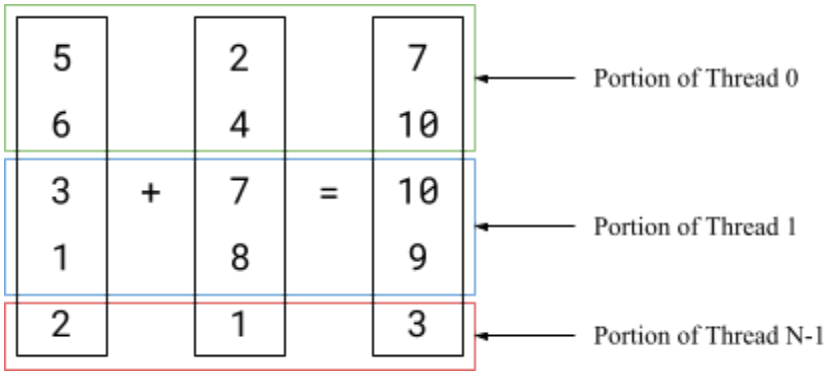
- Get hands on experience with OpenMP and MPI libraries in HPC system
- Understand limits and overheads of parallelization

In this project, you will work on parallelizing three basic linear algebra algorithms. You will implement them using two Parallel Programming approaches as explained in class: Shared Memory Model (OpenMP) and Message Passing Interface (MPI). A comparative study will provide insights on the performance of various parallelization approaches.

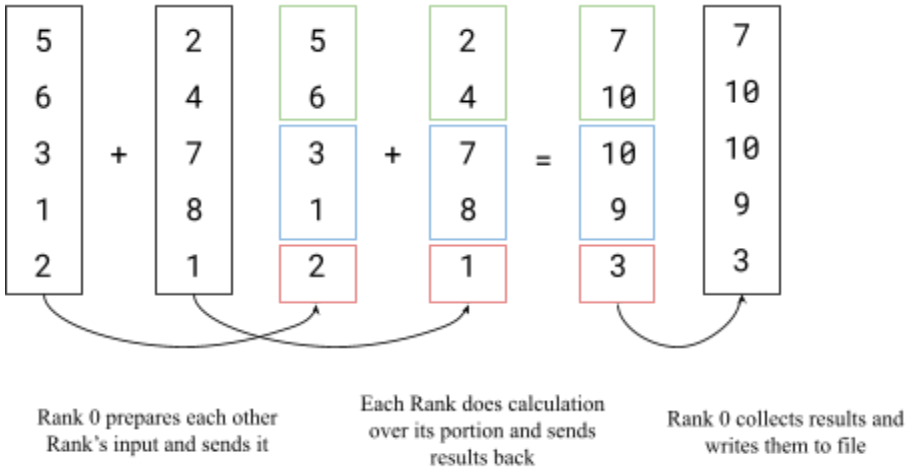
2. Algorithms

For every algorithm, see input portion of the start code, dataset and output for format.

i. Vector addition. Vector addition is N by 1 matrix addition.



Shared Memory Model: Given two vectors, one can divide work to some number of threads. Each thread will add its portion and write results into the respective portion of the output array.

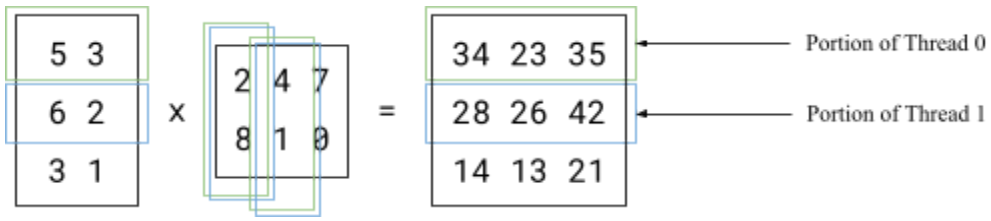


Message Passing Interface: Different from the shared memory approach, ranks can't access input data unless it is sent to them by rank 0. Please note that it is not necessary to send the whole vector to each rank, and it reduces overhead.

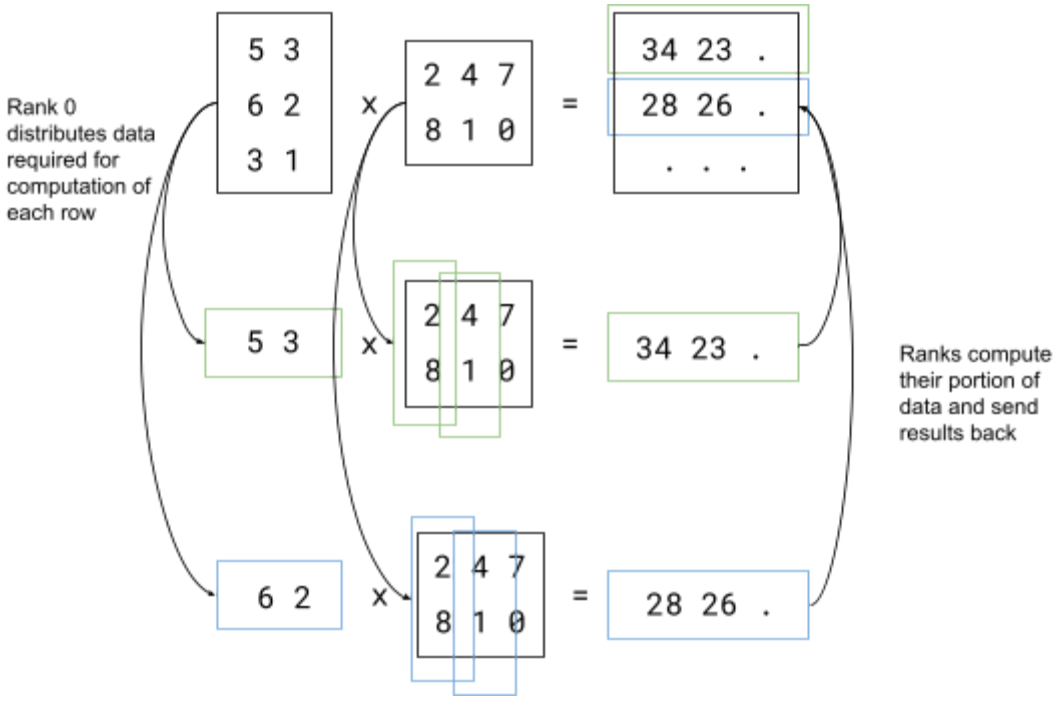
ii. Matrix addition

Matrix addition is processed the same way as vector addition, but different decisions could be made on how to partition input data across threads/ranks.

iii. Matrix multiplication



Shared Memory Model: Matrix multiplication is more interesting, since there are many ways of partitioning the work. On the figure, an example of partitioning by row and data accessed for calculating the first two elements of each row is shown. Each thread calculates one row of the answer, by accessing a row and a column from two input matrices. Note that all data is kept in one node's memory and each thread can access any desired element of input matrices.



For this problem, we don't require any advanced matrix multiplication algorithm. Please implement a solution which you feel is efficient enough.

Message Passing Interface: As it was said earlier, we can try different partitioning. On the figure row partitioning is shown. With that approach, ranks need one row and all columns of first and second matrix operand, respectively. Note that this is more efficient than sending both matrices in full.

Implement serial, shared memory (OpenMP) and message passing (MPI) solutions. Since algorithms are straightforward, the exact way to program them is up to you. *You can divide data between any number of tasks in any way you see it feasible*, but don't forget - parallel programming abstractions are not free!

3. Datasets

You are provided with a matrix generator script. It requires seed value, X and Y values as input and prints the 2^X by 2^Y matrix. Seed value is required for grading purposes. Please always use the seed value as given. You can redirect stdout to file to save the matrix. The following code will create two files with 8x1 vectors.

```
python gen_matrix.py 406 3 0 > vec_8_1_a.txt

python gen_matrix.py 506 3 0 > vec_8_1_b.txt
```

For reporting, please generate datasets with following parameters:

Vector Addition

Name	Matrix A & B size	Matrix A/Matrix B parameters to generator
Debug	16x1	406 4 0
		506 4 0
Small	8,192x1	406 13 0
		506 13 0
Medium	131,072x1	406 17 0
		506 17 0
Large	134,217,728x1	406 27 0
		506 27 0

Matrix Addition

Name	Matrix A & B size	Matrix A/Matrix B parameters to generator
Debug	16x16	406 4 4
		506 4 4
Small	64x128	406 6 7
		506 6 7
Medium	256x512	406 8 9
		506 8 9
Large	8,192x16,384	406 13 14
		506 13 14

Matrix Multiplication

Name	Matrix A/Matrix B size	Matrix A/Matrix B parameters to generator
Debug	8x16	406 3 4
	16x8	506 4 3
Small	128x64	406 7 6
	64x128	506 6 7
Medium	256x512	406 8 9
	256x512	506 9 8
Large	16,384x8,192	406 14 13
	8,192x16,384	506 13 14

Handling inputs with sizes of power of two inputs is enough for the project.

4. Infrastructure

Starter code with inputs being already read, compilation and run routines are provided in the project repository. We tried to minimize your burden and let you focus on the most important points by providing those.

You should modify files `src/{serial,omp,mpi}_{vecadd,matadd,matmul}.c`. Each file is standalone and has all library headers required for development. Makefile contains targets which will compile these sources with appropriate compiler and library.

The report requires data for 16 parallel running tasks. An average laptop can run only up to 8 parallel tasks. So to collect your data, you will need to run your implementation on a cluster which has server-grade processors. Details on how to connect and run code will be provided later.

5. Initial Setup

- Start modifying Vector Addition implementation in `src/serial_vecadd.c`
- Compile the code
    - `make serial`

    - Binaries which were able to be build, will be placed in `bin/`
- Study targets in Makefile and update appropriate variables
- Generate test data
    - `make gen_matrix`
- Run the code
    - `make run_serial`

After you are done with OpenMP and MPI versions, repeat the cycle choosing appropriate Makefile targets.

In addition, we encourage you to expand `run` targets for batch running with different input sizes.

Data Structure

data_struct is defined in data.h and it is a dynamic data structure which stores matrix as a double pointer array. It is filled in get_data_struct() implemented in data_handling.c.

For storing answer you can reuse the same data structures d_1 or d_2, but probably will need to create one for the multiplication. You can get some inspiration from lines 8-10 and 39-41 in data_handling.c

Required Software

Cluster nodes will have all required packages.

6. General Guidelines

- **Start as early as possible**. It will help you understand the infrastructure early and use the rest of the time for coding.
- Implement serial version of algorithms
- Think of:
    - how you can divide problem into multiple parallel tasks
    - how you will distribute data
    - how you will gather and print the answer
- Implement the shared memory version (OpenMP). It is easier because communication between tasks is implicit. *Please start with the defaults scheduling policy, chunk size, etc., get working parallel code, record timing on large inputs and only then try to use different scheduling policies, comparing timings.*
- Vary number of threads in OpenMP implementation and prepare report.
- Implement the message passing version (MPI). Here, you need to explicitly send data between ranks. You will have the same source file/executable for master and worker

ranks, but they will run in isolation (possibly, on different machines), and hence, have access only to data which was explicitly sent!

- Vary number of ranks in MPI implementation and prepare report.

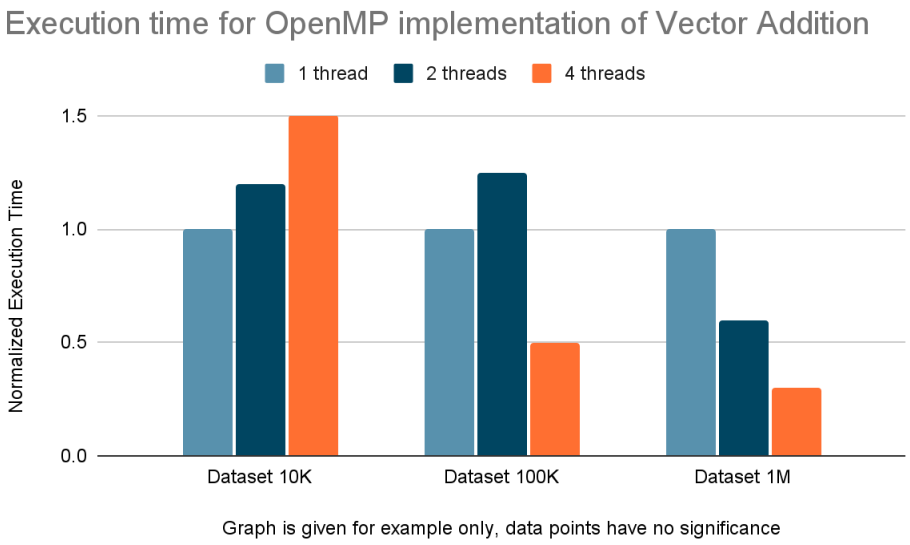
7. Report

1. Provide a short explanation of your implementations. This should include how you partitioned work across threads/ranks.

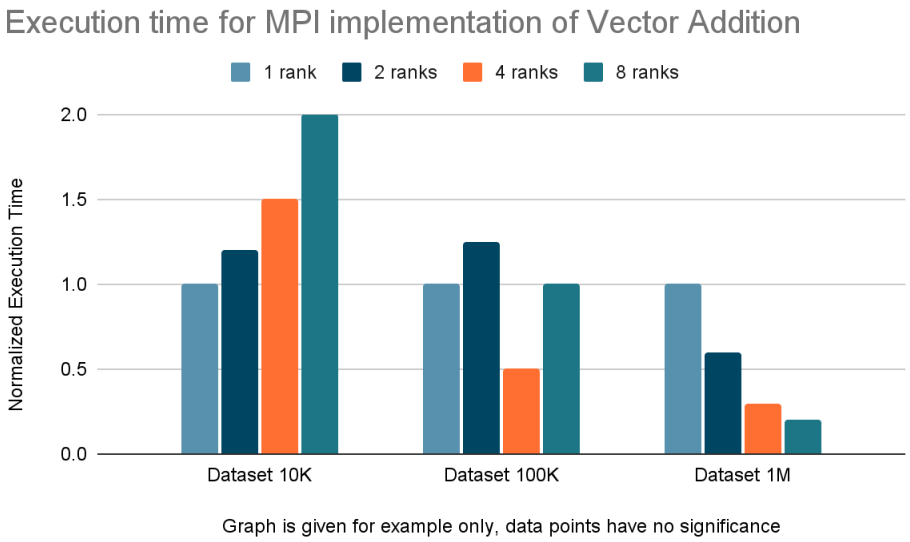
2. Plot following graphs. For each graph, run the required workload three times and use average. Comparing only time spent for calculation would eliminate fluctuation in reading/printing and will give you a better idea on where time is spent (program has I/O, memory allocation, deallocation and for large inputs they are all significant). For MPI, start timer just before sending data and stop right after rank 0 has received all data (i.e., include communication cost).

Provide raw numerical data in spreadsheet.

2a. For OpenMP implementation of each algorithm, prepare one graph showing execution times for different configurations. On the X axis vary the number of threads: {1, 2, 4, 8, 16}. On the Y axis show time normalized to 1 thread. This will total in 3 graphs. Briefly explain the results you got.

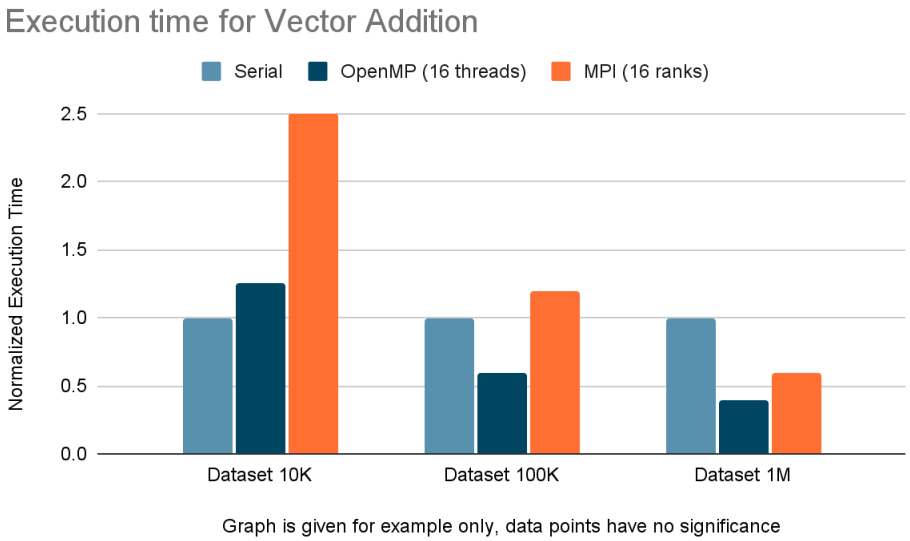


2b. For MPI implementation of each algorithm, prepare one graph showing execution times for different configurations. On the X axis vary the number of ranks: {1, 2, 4, 8, 16}. On the Y axis show time normalized to 1 rank. This will total in 3 graphs. Briefly explain the results you got.



2c. For each problem, prepare one graph showing execution times for different implementations. On the X axis vary serial, OpenMP and MPI for each dataset. On the Y axis show time normalized to serial. For OpenMP use 16 threads, for MPI use 16 ranks. This will total in 3 graphs. Briefly explain the results you got.

565 students: Add Hybrid (4 ranks with 4 threads each) bar.



8. Submission Guidelines

Submit three source files for each algorithm. More details will be posted on GradeScope.

9. Grading

ECE/CSC-406

30%: Your code compiles successfully. Partial credit is given for substantial effort.
30%: All three implementations generate correct output matching exactly (points will be equally distributed).
40%: Report. Credit will be given on the statistics shown and the discussion presented.

ECE/CSC-506

30%: Your code compiles successfully. Partial credit is given for substantial effort.
30%: All three implementations generate correct output matching exactly (points will be equally distributed).
30%: Report. Credit will be given on the statistics shown and the discussion presented.
10%: Implementing OpenMP-MPI hybrid solution to improve performance wherever applicable.

10. Self-grading

For your convenience, Gradescope Self-grader is provided. It tests only functional correctness of your code, and not parallelization efforts. Grader will compile your code, run it and compare output with `numdiff` tool.

To submit your code, use a new target `make pack.`

Output format for resulting matrix: one row per line, using whitespace as a separator. Print output to `stdout`. Extra output should be removed or printed to `stderr`.

We encourage you to submit to Gradescope as early as possible, that way you will fix all compilation and output issues early. We believe that output is self-explanatory to some extent.

Currently, custom Makefile is not supported. Your code will be compiled using the given `make {omp,mpi,hybrid}_{vecadd,matadd,matmul}` targets.

11. Hybrid Implementation

The main idea for Hybrid implementation is shared memory (OpenMP) inside message passing (MPI). Each MPI rank has its own disjoint memory, and computation is done serially (on many nodes simultaneously). You can enhance that with introducing shared memory parallelism inside

each rank, thus utilizing all resources in that node (though, outside this project you can achieve the same thing by running multiple ranks on that node, too).