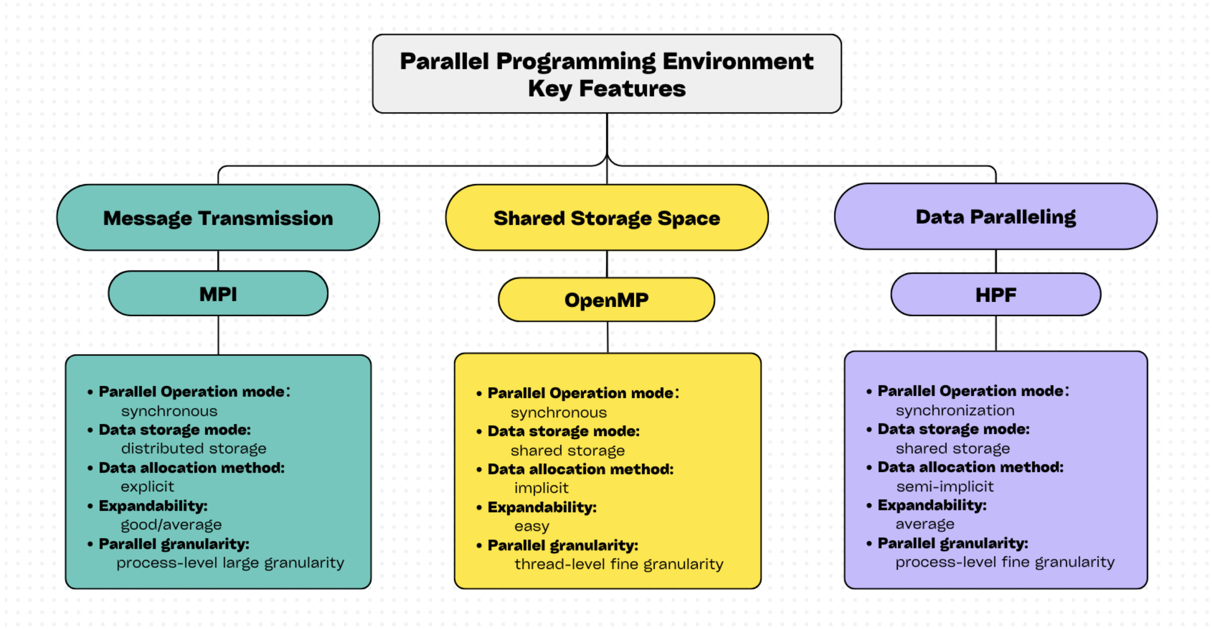
CSCI596 Final Project — Parallel computational analysis of various algorithms

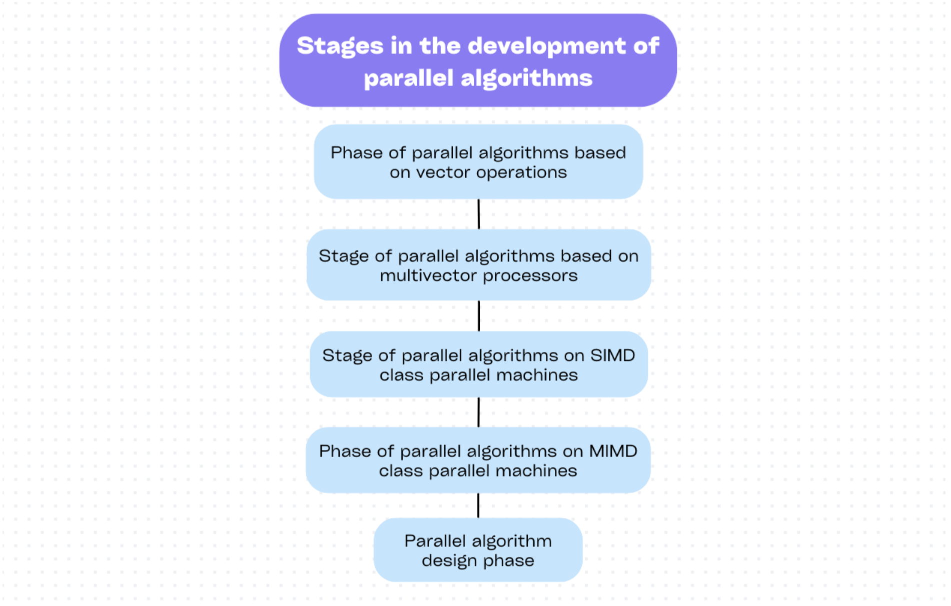
Background Introduction:

Parallel computing has emerged as a pivotal paradigm in addressing the escalating demands for computational power posed by contemporary applications and data. By concurrently executing tasks across multiple processors or computing units, parallelization endeavors to expedite the execution of algorithms and significantly enhance overall performance. In this project, we delve into the parallelization of fundamental algorithms, namely backtracking, dijkstra, matrix operations, and Fourier transformation…, employing prevalent techniques such as MPI, CUDA, and MapReduce…

Current parallel programming environments fall into three main categories: message passing, shared storage and data parallelism.


Parallel Programming Environment Key Features
Message Transmission
MPI
• Parallel Operation mode: synchronous
• Data storage mode: distributed storage
• Data allocation method: explicit
• Expandability: good/average
• Parallel granularity: process-level large granularity
Shared Storage Space
OpenMP
• Parallel Operation mode: synchronous
• Data storage mode: shared storage
• Data allocation method: implicit
• Expandability: easy
• Parallel granularity: thread-level fine granularity
Data Paralleling
HPF
• Parallel Operation mode: synchronization
• Data storage mode: shared storage
• Data allocation method: semi-implicit
• Expandability: average
• Parallel granularity: process-level fine granularity

The development of parallel computers influences the development of parallel algorithms. Stages in the development of parallel algorithms:


Stages in the development of parallel algorithms
Phase of parallel algorithms based on vector operations
Stage of parallel algorithms based on multivector processors
Stage of parallel algorithms on SIMD class parallel machines
Phase of parallel algorithms on MIMD class parallel machines
Parallel algorithm design phase

Importance of Parallelization:

As computational challenges burgeon, traditional sequential algorithms may struggle to meet the requirements of contemporary applications. Parallelizing algorithms allows us to harness the power of parallel architectures, comprising multi-core processors, GPUs, and distributed computing environments. This project aims to assess the efficiency gains achieved by parallelizing algorithms, contributing to a deeper understanding of the role of parallel computing in algorithmic optimization.

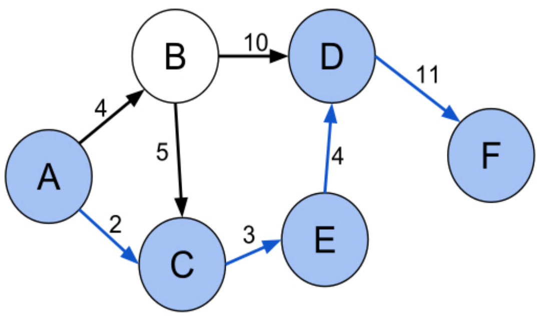
Implementation of Dijkstra’s Algorithm using MPI

Introduction:

Dijkstra’s algorithm is a graph search algorithm that finds the shortest path from a source to every other reachable vertex. Here we compare the performance of Dijkstra’s algorithm when implemented serially versus in parallel using MPI.

Implementation Procedure Overview:

1. Partition and assign vertices to processes
   - For example, we have a graph with 3 vertices and 3 available processes



u, v	A	B	C	D	E	F
A	0	4	2	∞	∞	∞
B	∞	0	5	10	∞	∞
C	∞	∞	0	∞	3	∞
D	∞	∞	∞	0	∞	11
E	∞	∞	∞	4	0	∞
F	∞	∞	∞	∞	∞	0

   - matrix[u][v] represents the distance from vertex u to vertex v, with “infinity” indicating no direct edge
   - **Process 0**: Manages columns 0 and 1 (vertices A and B)
   - **Process 1**: Manages columns 2 and 3 (vertices C and D)
   - **Process 2**: Manages columns 4 and 5 (vertices E and F)

2. Each process identifies its closest vertex to the source vertex
3. Perform a parallel prefix to select the globally closest vertex
4. Broadcast the result to all the processes
5. Each process updates its local distance from the source to this newly found globally closest vertex

Experiment:

Setup

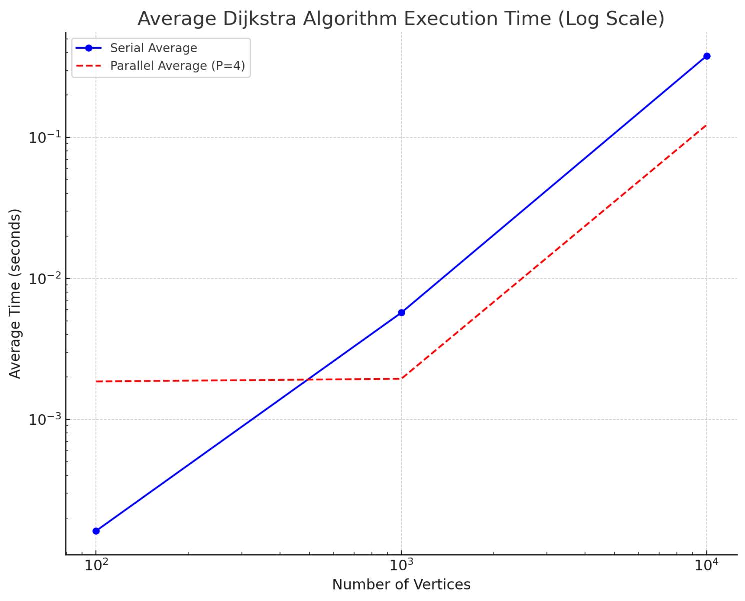
- CPU: Apple M1 Pro (8 physical cores)
- Input: randomly generated weighted directed graphs with 100, 1000, 10000 vertices

Method

1. Run serial Dijkstra’s and parallel Dijkstra’s (using 4 processes) 3 times each version
2. Record the execution time of each run
3. Calculate the average execution time of serial and parallel version

Result:

The parallel version significantly outperformed the serial one for large graphs, particularly at 10,000 vertices. However, for small graphs with 100 vertices, the serial implementation was more efficient. The next step is to scale the implementation on a high-performance computing cluster to run the algorithm with more processes and handle much larger inputs. This scale-up will provide more insight into the topic.



Number of Vertices	Serial Average	Parallel Average (P=4)
100	0.000162	0.001859
1000	0.005716	0.001941
10000	0.378364	0.122562

Implementation of FFT using MPI

Introduction:

A fast Fourier transform is an algorithm that computes the discrete Fourier transform of a sequence, or its inverse. Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa.

Implementation Procedure Overview:

1. Perform bit-reversal permutation on the input sequence
2. Assign N/P numbers to each rank (assume number of ranks must be a power of 2)
3. Each rank performs butterfly operations without inter-rank communication
4. Perform butterfly operations with communication which includes identifying the partner, sending its data and receiving the partner’s data

Experiment (local):

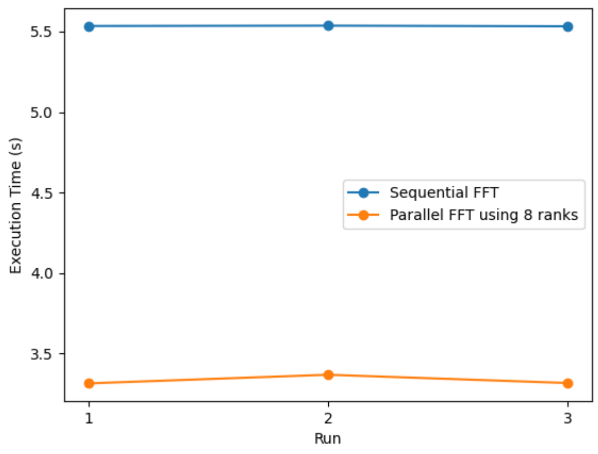
Setup

- CPU: Apple M1 Pro (8 physical cores)
- Input: input20.in (2^{20} = 1048576 numbers)

Method

4. Run sequential FFT and parallel FFT (using 8 ranks) 3 times each version
5. Record the execution time of each run
6. Calculate the average execution time of sequential FFT and parallel FFT

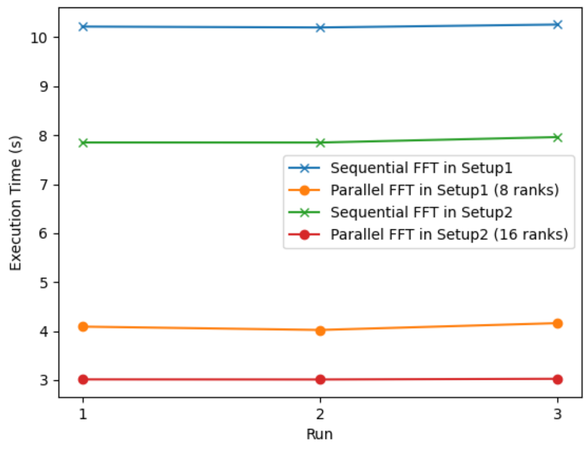
Result: parallel FFT runs faster than sequential FFT in this experimental setup



	Sequential FFT	**Parallel FFT using 8 ranks**
Average Execution Time (Second)	5.534798666666688	3.3318089999999834

<u>Experiment (USC CARC):</u>

- Run sequential FFT and parallel FFT with 8 ranks and 16 ranks



	Sequential FFT	**Parallel FFT using 8 ranks**
Average Execution Time (Second)	10.227008333333334	4.091674557666667

	Sequential FFT	**Parallel FFT using 8 ranks**
Average Execution Time (Second)	7.888008770666666	3.0162059243333332

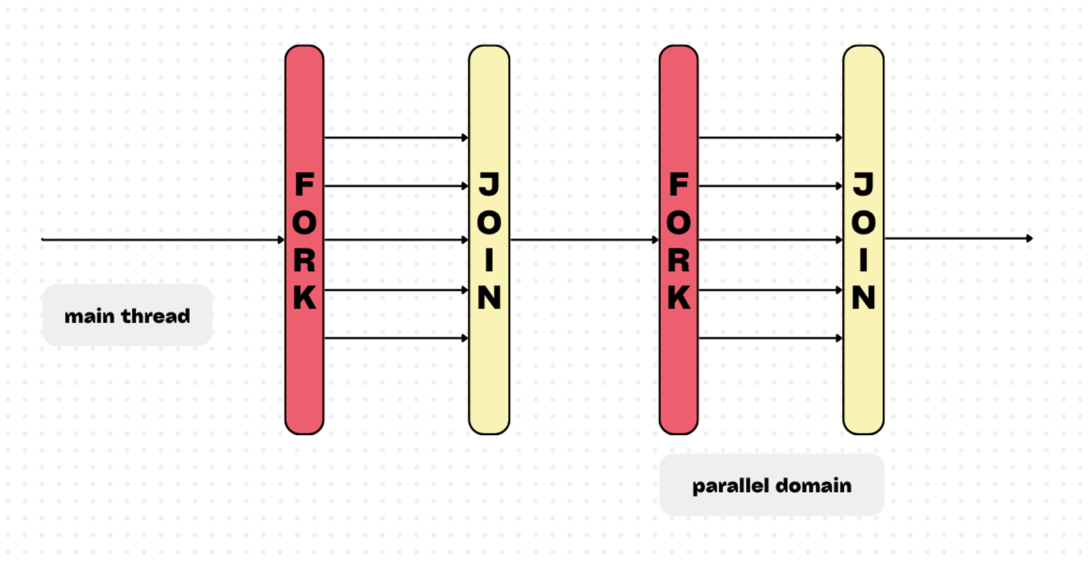
Result:

- Parallel FFT runs faster than sequential FFT
- Parallel FFT (16 ranks) runs faster than parallel FFT (8 ranks)

Implementation of N Queens's Algorithm using OpenMP

Introduction:

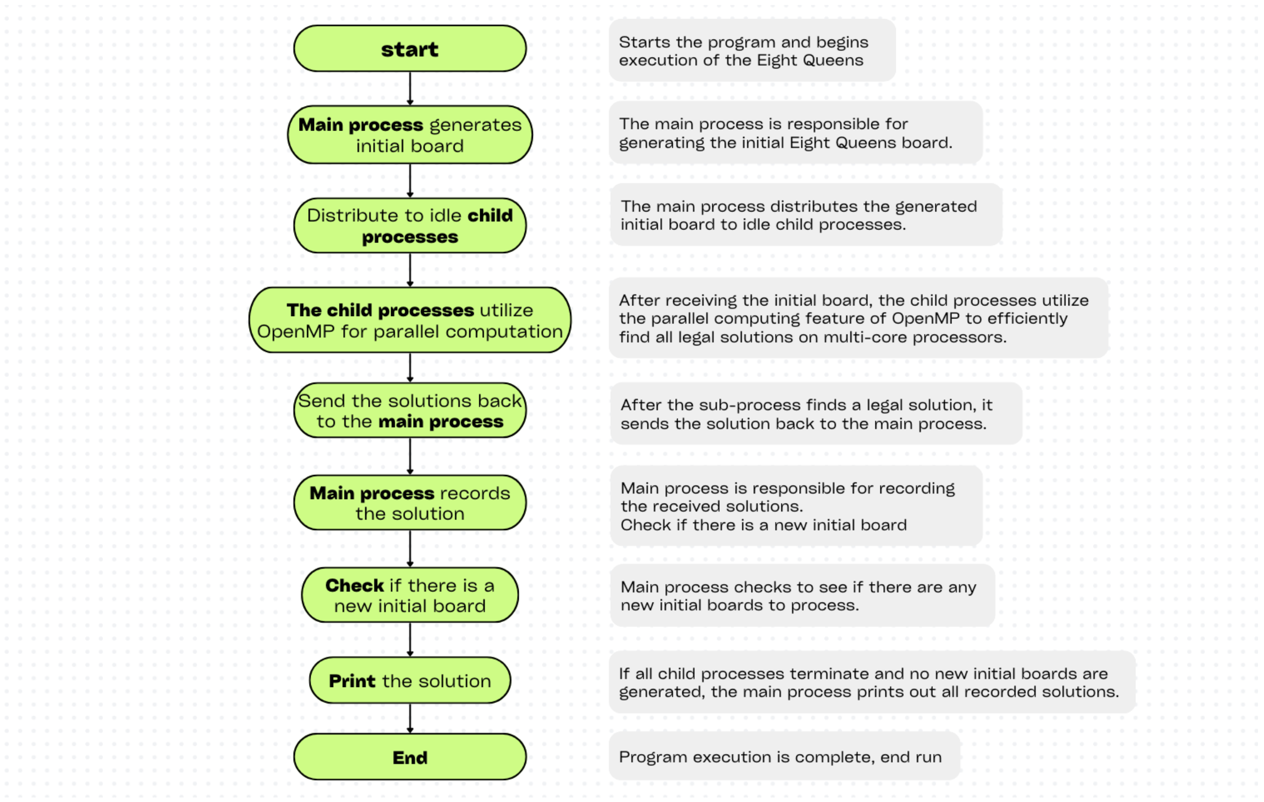
OpenMP uses the Fork-Join parallel execution model:



Implementation Structure Overview:

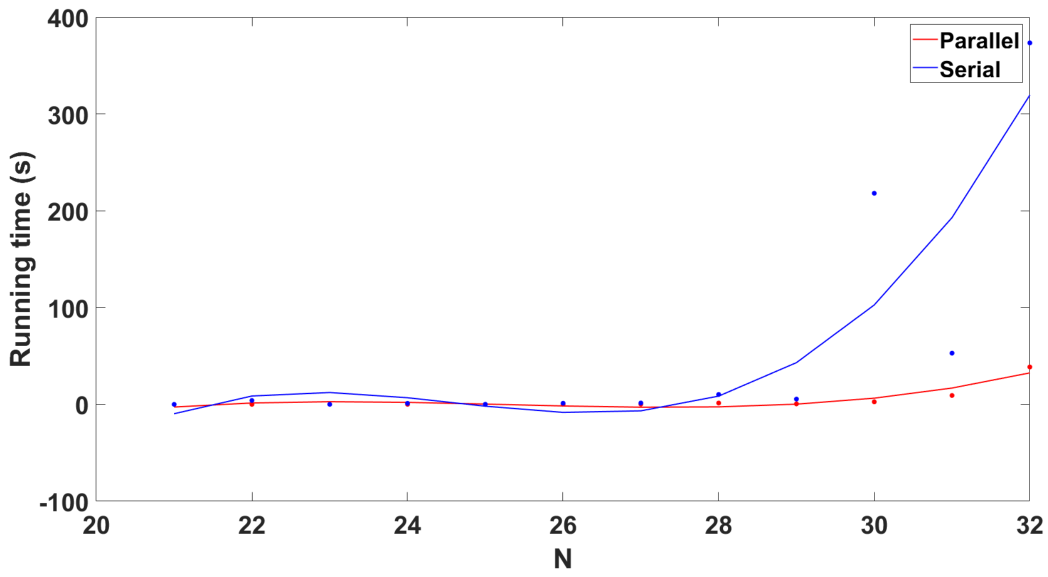
The N Queens Problem aims to place N queens on a n x n chessboard, requiring them not to be in the same row, column, or diagonal, and is a challenging intellectual problem.

Algorithm implementation structure diagram:


start
Main process generates initial board
Distribute to idle child processes
The child processes utilize OpenMP for parallel computation
Send the solutions back to the main process
Main process records the solution
Check if there is a new initial board
Print the solution
End
Starts the program and begins execution of the Eight Queens
The main process is responsible for generating the initial Eight Queens board.
The main process distributes the generated initial board to idle child processes.
After receiving the initial board, the child processes utilize the parallel computing feature of OpenMP to efficiently find all legal solutions on multi-core processors.
After the sub-process finds a legal solution, it sends the solution back to the main process.
Main process is responsible for recording the received solutions.
Check if there is a new initial board
Main process checks to see if there are any new initial boards to process.
If all child processes terminate and no new initial boards are generated, the main process prints out all recorded solutions.
Program execution is complete, end run

Result:

After testing the above serial and parallel algorithms on a 16-core host, fitting the resulting data yields the following figure:



N	21	22	23	24	25	26	27	28	29	30	31	32
Parallel execution time (s)	0.015	0	0.016	0.031	0.031	0.125	0.297	1.422	0.531	2.657	9.218	38.641
Serial execution time (s)	0.015	4.063	0.062	1.094	0.14	1.219	1.453	10.25	5.484	218.05	52.938	373.34

Conclusion:

under different N conditions, especially when N is 29, 30, and 32, the eight queens algorithm with OpenMP parallel computing presents significant advantages compared to serial execution. This also confirms that under certain conditions, the use of parallel computing can effectively improve the performance of the algorithm.

MPI based matrix multiplication parallel computing

Introduction:

Matrix multiplication is a foundational operation with broad applications in diverse fields. Its importance stems from its role in representing, transforming, and analyzing complex systems and relationships, making it a fundamental concept in mathematics and its applications. It is fundamental to many machines learning algorithms and can also be used in computer graphics for transformations and rendering.

Problem Description:

Given two matrices A and B. where A has size i × k and B has size k × j. Now find the product of A and B: C=A×B.

In C++, it can be implemented using a simple multiple loop:

```
int i, j, k;
for (i = 0; i < N; i++) {
```

```
                        for (j = 0; j < N; j++) {
                                res[i][j] = 0;
                                for (k = 0; k < N; k++)
                                        res[i][j] += mat1[i][k] * mat2[k][j];
                        }
                }
```

Parallel Algorithm Solution:

According to the above problems, each row element of the product matrix C is solved independently. Therefore, for cases where A and B are relatively large, parallel computing can be used to solve the rows of C. This can save a lot of time, but it requires sacrificing a certain amount of memory space. If there is enough memory, you can use the C++ MPI interface to implement parallel computing of multiple CPU cores.
Assume that there are np (number of processors) processes solving the problem in parallel at the same time. Since matrices in C++ are stored row-wise, A can be chunked row-wise.

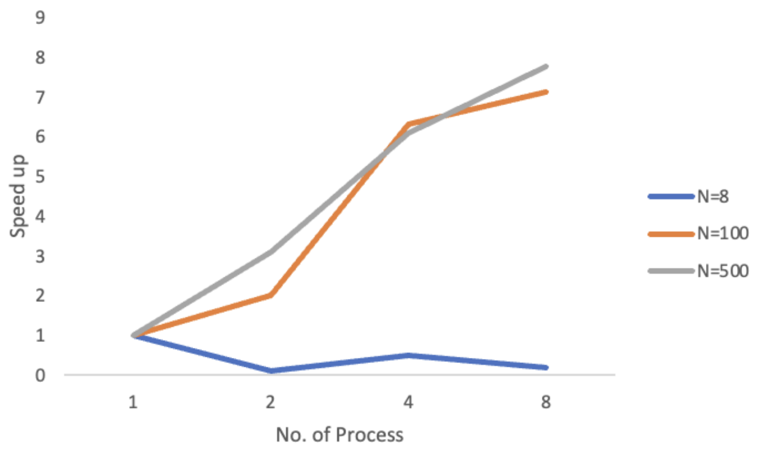
Decomposition steps:

- step 1: The main process divides A into chunks and sends it to all sub-processes;
- step 2: The main process sends B to all sub-processes;
- step 3: All processes calculate their respective matrix products;
- step 4 The main process collects the results.

Result:

- CPU: Apple M1 Pro (8 cores)
- Threads per core: 2

Implemented a simple MPI program to compute the matrix multiplication. Splitting the matrix A row wise, and distribute it to different processes. Comparing the runtime using 1, 2, 4 and 8 processors.



Conclusion:

It can be seen from the obtained data that the larger the size of the matrix, the greater the impact of the number of processes on the matrix calculation speed. For a matrix of the same size, the more processes there are, the faster the calculation processing will be.