**CMPE 275**

**Grand Challenge-2 Report**

**Team Achievers**

**TEAM MEMBERS**

Sushmitha Dhummi Thrilochana - 016043755

Sai Sri Harshini Kosuri - 016005912

Thirumala Sai Krishna Kaaja - 016033758

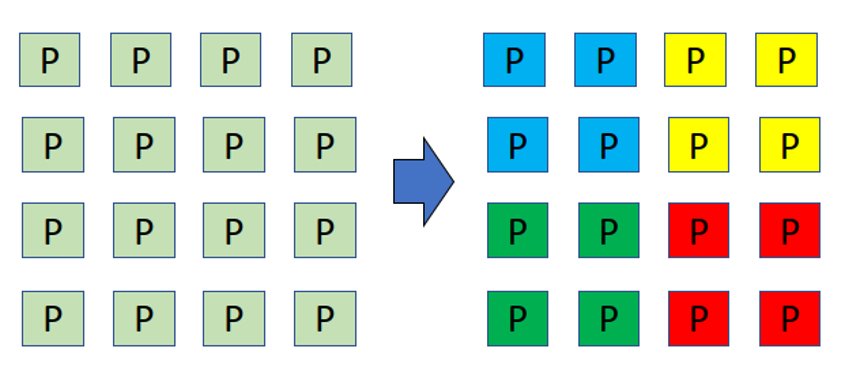
Santhosh Bodla - 016002454

Abhiram Yenugadhati - 015962128

Sai Subhash Reddy Gangireddygari - 016003403

**PROBLEM STATEMENT**

* To create partitioned compute space - Blue, Yellow, Green, and red that works independently.
* Share or exchange information along shared edges.



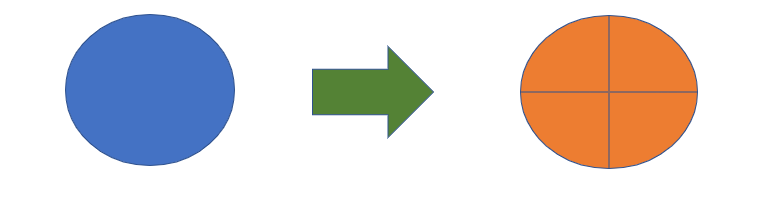
## **Multithreading vs. multiprocessing**

* Multiprocessing increases computing power by using two or more CPUs, whereas multithreading increases computing power by using a single process with multiple code segments.
* Multithreading is concerned with creating computing threads from a single process, whereas multiprocessing is concerned with increasing computing power by adding CPUs.
* Multiprocessing is used to make a more reliable system, whereas multithreading is used to create parallel threads.
* Multithreading is easy to set up and requires few resources, whereas multiprocessing takes a long time and specific resources to set up.
* Multiprocessing executes multiple processes at the same time, whereas multithreading executes multiple threads at the same time.
* Multiprocessing creates a separate address space for each process, whereas multithreading uses a shared address space for all threads.

**Fundamentals of Parallel Processing**

Generally, these fundamentals are broken into 4 major steps:

1. Decomposition
2. Assignment
3. Orchestration
4. Mapping



MPI

**MPI** is a directory of C++ programs which illustrate the use of the Message Passing Interface for parallel programming.

MPI allows a user to write a program in a familiar language, such as C, C++, FORTRAN, or Python, and carry out a computation in parallel on an arbitrary number of cooperating computers.

**MPI** program is written so that one computer supervises the work, creating data, issuing it to the worker computers, and gathering and printing the results at the end.

**Open MPI (Message Passing Interface)​**

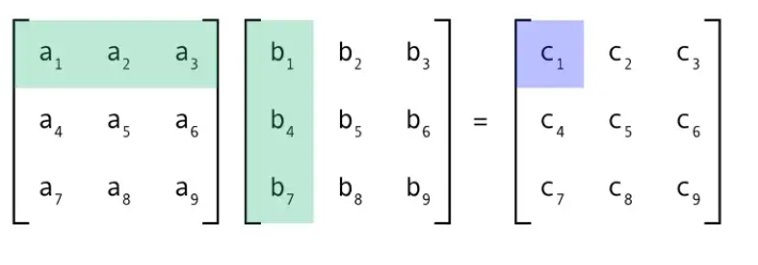
A High Performance Message Passing Library​

* Useful for parallel computing in distributed memory model.​
* Here, message passing is used to synchronize distribution of computation and collect results between nodes .

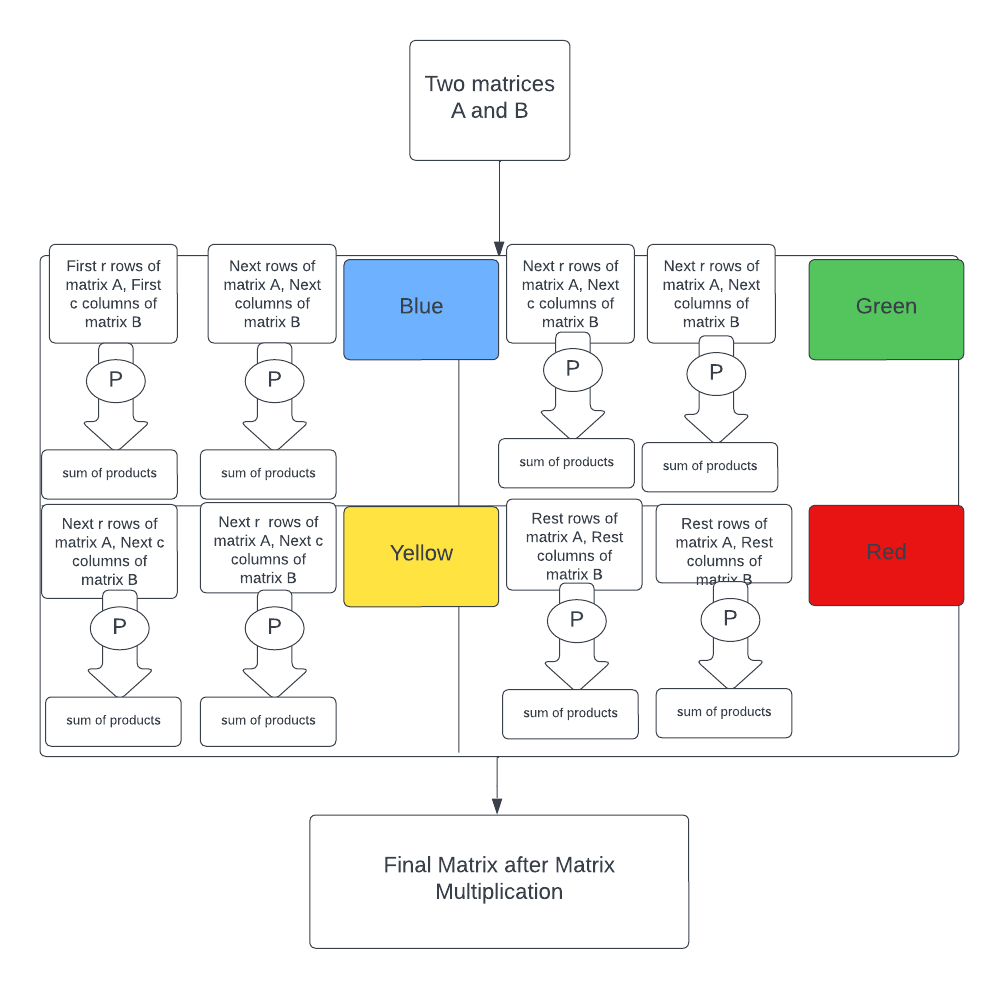
**Here we have picked matrix multiplication as a problem to be solved using open MPI and analyzed it.**



1. Here we are using the above discussed parallel programming concepts and dividing the rows and columns between different processes.​
2. Each process will compute its work independently and return back the response through message passing​
3. Here we have initialized two matrices and used a random function to auto fill the numbers automatically.
4. We changed the size of the matrix for the testing purpose.



**Architecture Diagram:**



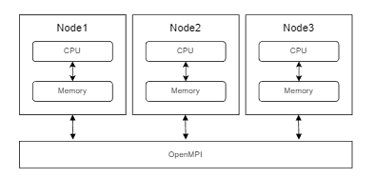
**Our Approach:**

**Basic functionalities:**

* Here we have taken matrix multiplication for testing parallel computing with the help of open MPI.
* We have tested the time taken for calculating matrices on different matrix sizes as well as with different numbers of processes.

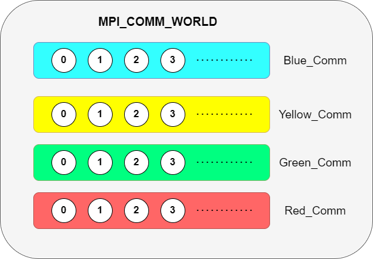
**Our Core Components:**

* Communicators – Set of processes allowed to communicate among themselves. ​
* Ranks – Each process is assigned a unique rank.​
* Messages are predefined data types ​
* ex: MPI\_INT, MPI\_FLOAT, MPI\_CHAR​
* Point-to-point communication (Blocking/ Non-Blocking)​
* ex: MPI\_Send(), MPI\_Recv()​
* Collective communication​
* ex: MPI\_Bcast(), MPI\_Scatter(), MPI\_Gather()

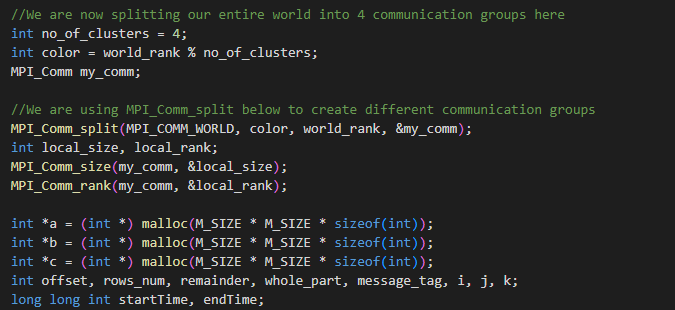


**Steps Performed:**

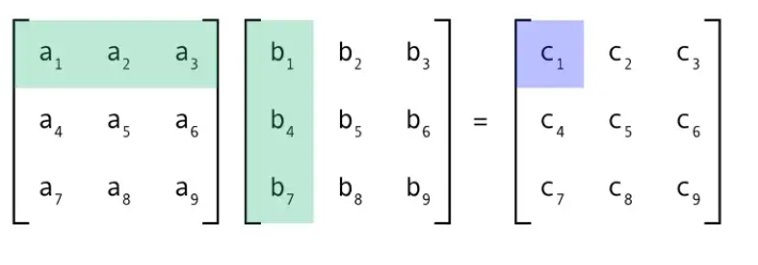
1. We have set up 4 different communicators with the help of reminder logic.



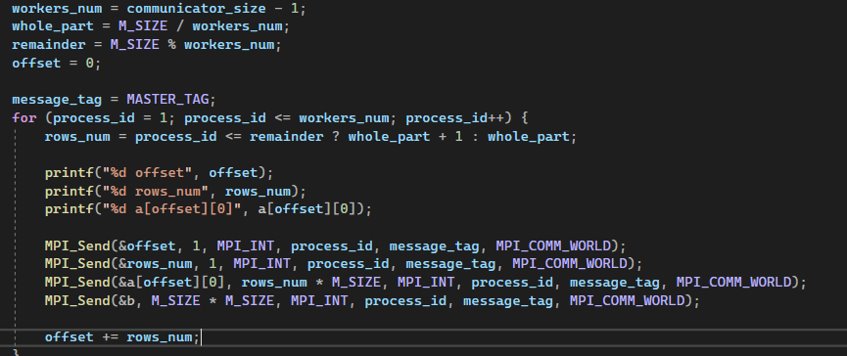
1. After that, each process is assigned a unique rank.
2. We have used MPI\_send() and MPI\_Recv() for communication between communicators.
3. Within the communicator, we used MPI\_Scatter() instead of MPI\_Bcast() because we are dividing the rows based on offset so in every turn we are getting different rows of the matrix rather than in MPI\_Bcast we will be sending the same data to different processes.



1. So basically we are dividing the number of rows based on the number of processes in one communicator.
2. There will be a leader in every communicator and it will assign the work equally to all other processes with MPI\_Scatter() and get responses from them with the help of MPI\_Gather().
3. For example, if we have 3\*3 matrix, 3 worker processes and one leader process, the main leader process will distribute the first row of the first matrix to process 1 and so on.



1. After the assignment of work these worker processes will solve the sub problem independently and send back responses to the leader.



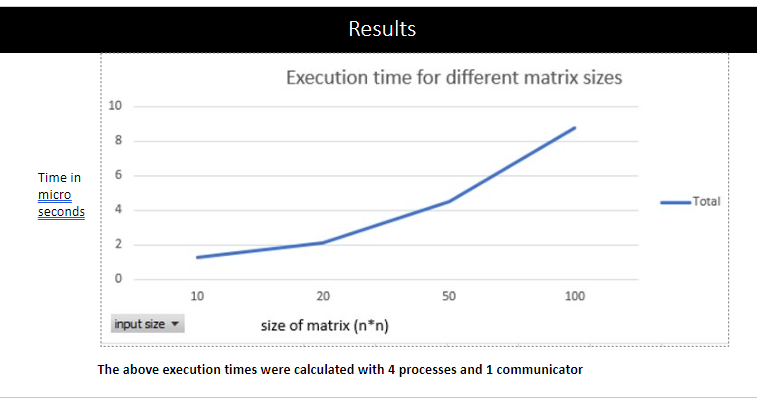
**PROS AND CONS OF Open MPI:**

**PROS:-**

* Simple: need not deal with message passing as MPI does
* Data layout and decomposition are handled automatically by directives.
* Incremental parallelism: can work on one part of the program at one time, no dramatic change to code is needed.
* Unified code for both serial and parallel applications: OpenMP constructs are treated as comments when sequential compilers are used.

**CONS:-**

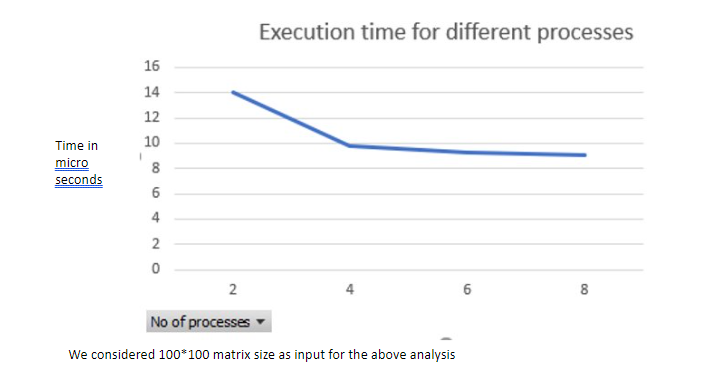
* Requires a compiler that supports OpenMP.
* Scalability is limited by memory architecture.
* no support for compare-and-swap
* Reliable error handling is missing.
* Lacks fine-grained mechanisms to control thread-processor mapping.
* Can't be used on GPU
* High chance of accidentally writing false sharing code
* Multithreaded Executables often incur longer startup times so, they can actually run much slower than if compiled single-threaded, so, there needs to be a benefit to being multi-threaded.



**The above execution times were calculated with 4 processes and 1 communicator​**

**Conclusions:**

1. In the first graph as the size of the matrix increases the execution time is also gradually increased.
2. The above execution times were calculated with 4 processes and 1 communicator.



**We considered 100\*100 matrix size as input for the above analysis​**

**Conclusions:**

1. In the graph above as the number of processors increases, execution time decreases.
2. In the second graph, we used 4 core machine, we can see the clear difference after 4 processors we see the execution time is significantly same for all the processors

**REFERENCES:**

<https://www.open-mpi.org/doc/v4.1/>

<https://www.eecis.udel.edu/~cavazos/cisc879/Lecture-03.pdf>

<http://compphysics.github.io/ComputationalPhysics2/doc/LectureNotes/_build/html/parallelization.html>

<https://stackoverflow.com/questions/41575243/matrix-multiplication-using-mpi-scatter-and-mpi-gather>

<https://people.math.sc.edu/Burkardt/cpp_src/mpi/mpi.html#:~:text=MPI%20is%20a%20directory%20of,arbitrary%20number%20of%20cooperating%20computers.>

<https://www.open-mpi.org/faq/?category=running>

**The End**