CSE 708 - Programming Massively Parallel Systems

Parallel Matrix Multiplication

Name: Ashutosh Dubey
UBIT Name: ashutos2
Person Number: 50479324
Instructor: Prof. Russ Miller

Problem Statement

Develop an efficient parallel matrix multiplication algorithm for hardware accelerators in deep learning, specifically convolutional neural networks (CNNs) used in computer vision.

Practical Applications

- Computer Vision
- Feature Extraction in Machine Learning
- Image Processing

Sequential Approach

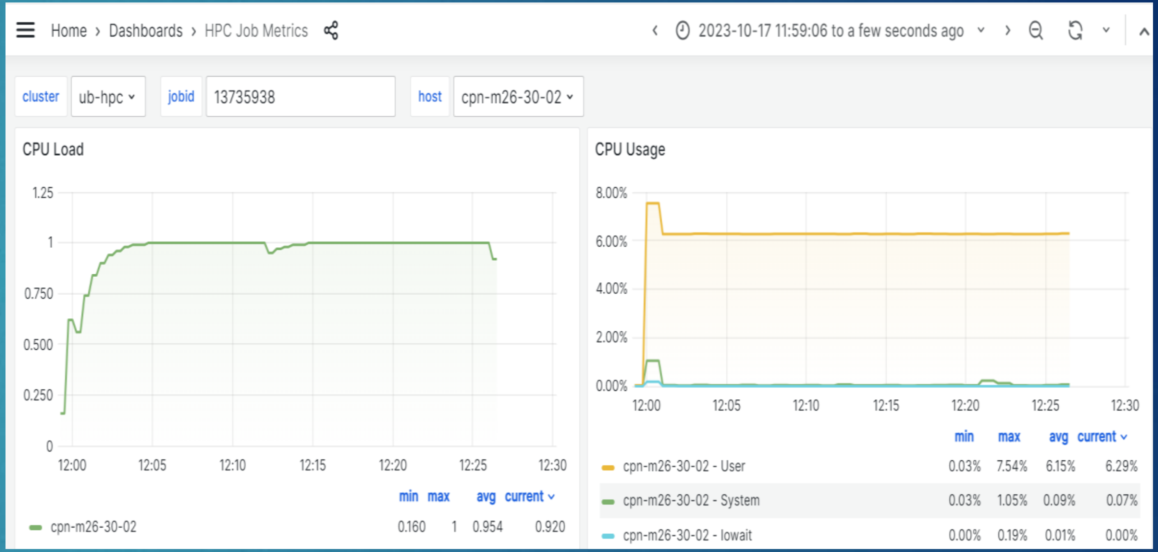
- Ensure the no of cols of first matrix A[n][n] equals the no of rows of second matrix B[m][m].
- Initialize an empty resulting matrix C
- Create three nested for loops, the outermost loop would iterate through no of rows of first matrix
- The second loop would iterate through no of cols of second matrix
- The third loop would iterate through no of rows of second matrix
- Compute element-wise multiplication of A[i][k] and B[k][j] inside the innermost loop and accumulate the products in a temporary variable
- Assign the sum to C[i][j]
- Repeat above steps for all elements in C, with i ranging from 0 to n and j from 0 to m.

Sequential Imp Output

Tabular Data

Processors	Input size	nodes	Cores per node	Time
1	8000*8000	1	1	00:25:35

Graph



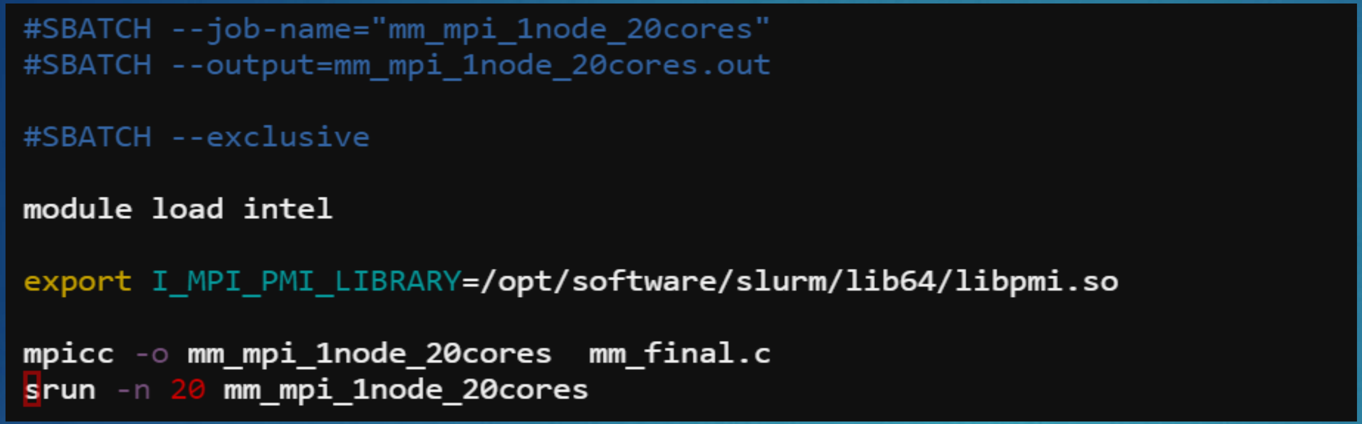
	min	max	avg	current
cpn-m26-30-02	0.160	1	0.954	0.920

	min	max	avg	current
cpn-m26-30-02 - User	0.03%	7.54%	6.15%	6.29%
cpn-m26-30-02 - System	0.03%	1.05%	0.09%	0.07%
cpn-m26-30-02 - Iowait	0.00%	0.19%	0.01%	0.00%

The output for the sequential implementation of matrix multiply program-
For two matrices A and B of size 8000*8000 sequentially, The graphs depicts that the running time was around 25 mins which is quite a lot

The other graph showcases how the CPU utilization changed over time, it was around 8 percent at max and around 6 percent on average which is quite low

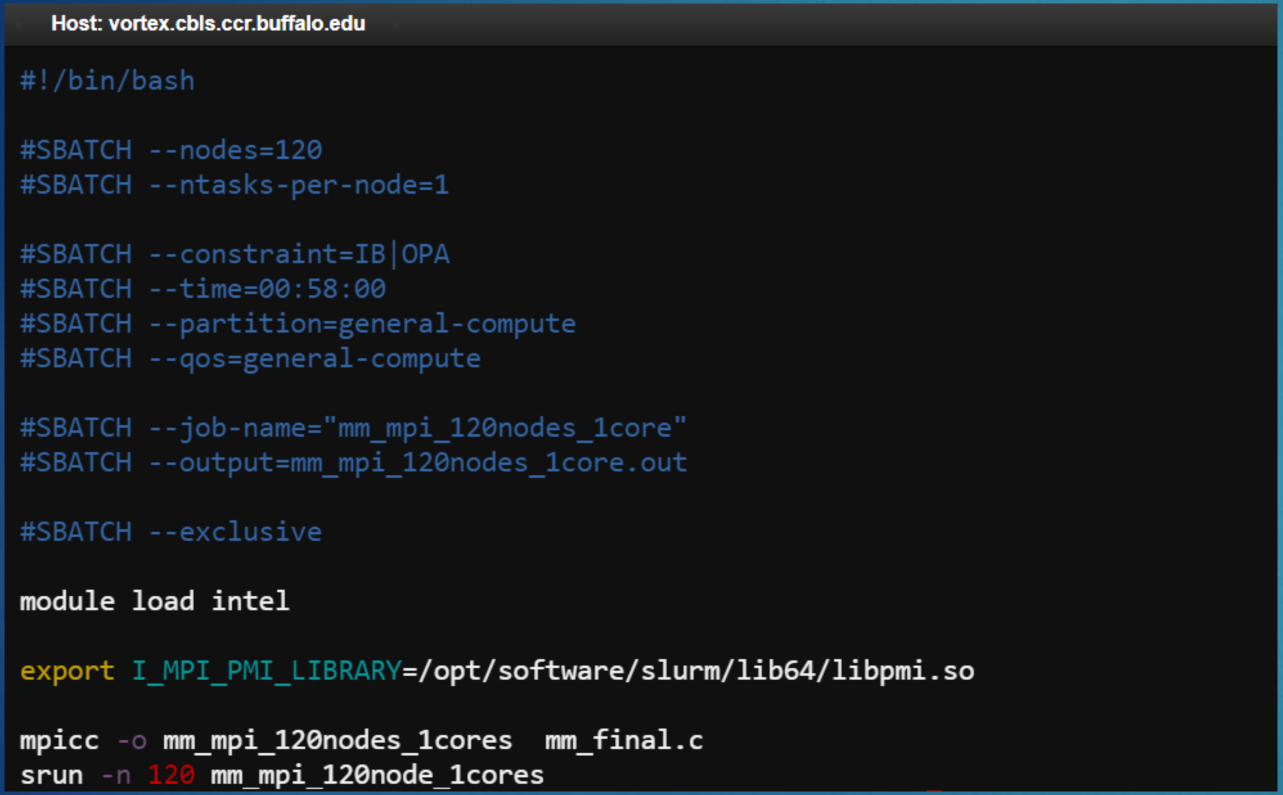
Setup



```
#SBATCH --job-name="mm_mpi_1node_20cores"
#SBATCH --output=mm_mpi_1node_20cores.out

#SBATCH --exclusive

module load intel

export I_MPI_PMI_LIBRARY=/opt/software/slurm/lib64/libpmi.so

mpicc -o mm_mpi_1node_20cores  mm_final.c
srun -n 20 mm_mpi_1node_20cores
```

- We used a special command called '--exclusive' in order to make sure the computer program reserves all the processing power to itself while running
- This command will tell the system to not to share any CPU cores with any other program
- This is how we made sure that no other programs could make use of the cores which were allocated to my program.

Setup



Host: vortex.cbls.ccr.buffalo.edu

```
#!/bin/bash

#SBATCH --nodes=120
#SBATCH --ntasks-per-node=1

#SBATCH --constraint=IB|OPA
#SBATCH --time=00:58:00
#SBATCH --partition=general-compute
#SBATCH --qos=general-compute

#SBATCH --job-name="mm_mpi_120nodes_1core"
#SBATCH --output=mm_mpi_120nodes_1core.out

#SBATCH --exclusive

module load intel

export I_MPI_PMI_LIBRARY=/opt/software/slurm/lib64/libpmi.so

mpicc -o mm_mpi_120nodes_1cores  mm_final.c
srun -n 120 mm_mpi_120node_1cores
```

- The total number of nodes requested over here is 120 which is specified using --nodes flag
- Number of cores per node requested over here is 1 which is specified using –ntasks-per-node flag
- Number of processors requested=(total number of nodes requested) * (number of cores per node)
- Thus, we have specified 120 processors in the srun command

Parallel Approach using MPI(Partially implemented)

- Given Matrix A and B of size N*N, and we have p no of processors
- The workload could be divided such that-
- Each processor is responsible for (N/p) rows of matrix A and (N/p) cols of matrix B.
- Matrix A is partitioned into (N/p) equally-sized vertical strips, and each processor is assigned one of these strips.
- Matrix B is similarly partitioned into (N/p) equally-sized horizontal strips, and each processor is assigned one of these strips.
- Each processor performs local matrix multiplication on its assigned portion of A and B.
- The local results from each processor are gathered and combined to construct the final result matrix C.

MPI Library functions

- MPI_Init(&argc, &argv): MPI library's initialization function. It is the starting point for any MPI program and must be called before any other MPI functions are used.

- MPI_COMM_WORLD: a communicator that allows to perform communication and coordination operations involving all processes in MPI program

- MPI_Comm_rank(MPI_COMM_WORLD, &rank): To retrieve the rank of the calling process within the specified communicator.
- Rank: an integer value ranging from 0 to num cores - 1, where num cores is the total number of cores in the communicator.

- MPI_Comm_size(MPI_COMM_WORLD, &num cores): To retrieve the total number of processes within the specified communicator

MPI Library functions

- MPI_Bcast : Broadcasts data from the process with rank 0 (the root process) to all other processes in the specified communicator

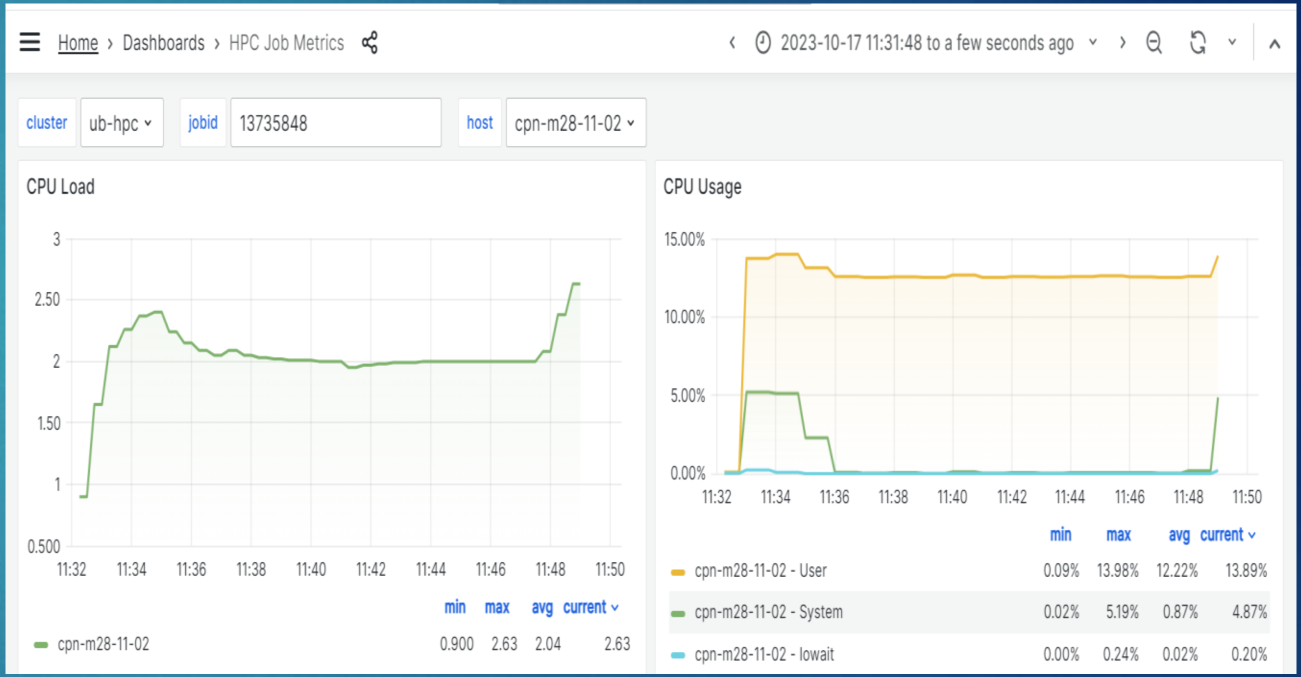
- MPI_Gather: Collects data from each process's local result, specifically the portion determined by starting row, and combines them into the result buffer on process 0 within the MPI_COMM_WORLD communicator.

Parallel 2 processors

Tabular Data

Processors	Input size	nodes	Cores per node	Time
2	8000*8000	1	2	00:16:35

Graph



	min	max	avg	current
cpn-m28-11-02	0.900	2.63	2.04	2.63

	min	max	avg	current
cpn-m28-11-02 - User	0.09%	13.98%	12.22%	13.89%
cpn-m28-11-02 - System	0.02%	5.19%	0.87%	4.87%
cpn-m28-11-02 - Iowait	0.00%	0.24%	0.02%	0.20%

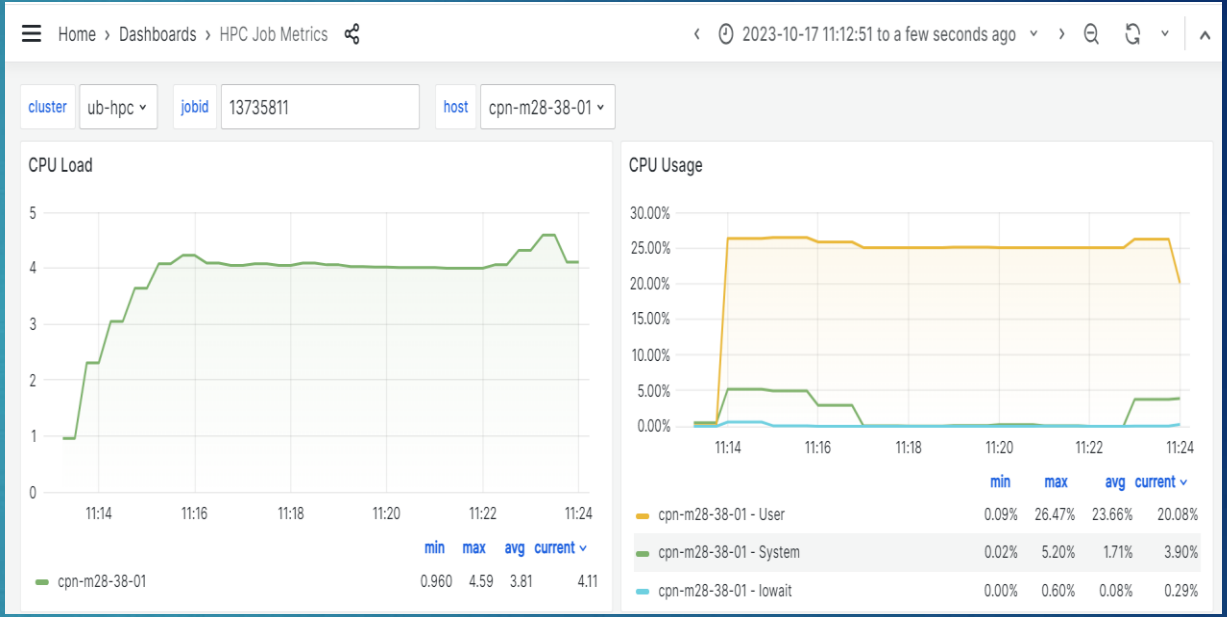
I partially implemented a parallel matrix multiplication algo with two processors on matrices of size 8000x8000. The running time was approximately 17 minutes, a significant improvement over the sequential version. The CPU utilization graph indicates a maximum of 14% and an average of 12%, which is double the utilization observed in the sequential approach.

Parallel 4 processors

Tabular Data

Processors	Input size	nodes	Cores per node	Time
4	8000*8000	1	4	00:10:35

Graph


Home › Dashboards › HPC Job Metrics
2023-10-17 11:12:51 to a few seconds ago
cluster
ub-hpc
jobid
13735811
host
cpn-m28-38-01
CPU Load
CPU Usage

	min	max	avg	current
cpn-m28-38-01	0.960	4.59	3.81	4.11

	min	max	avg	current
cpn-m28-38-01 - User	0.09%	26.47%	23.66%	20.08%
cpn-m28-38-01 - System	0.02%	5.20%	1.71%	3.90%
cpn-m28-38-01 - Iowait	0.00%	0.60%	0.08%	0.29%

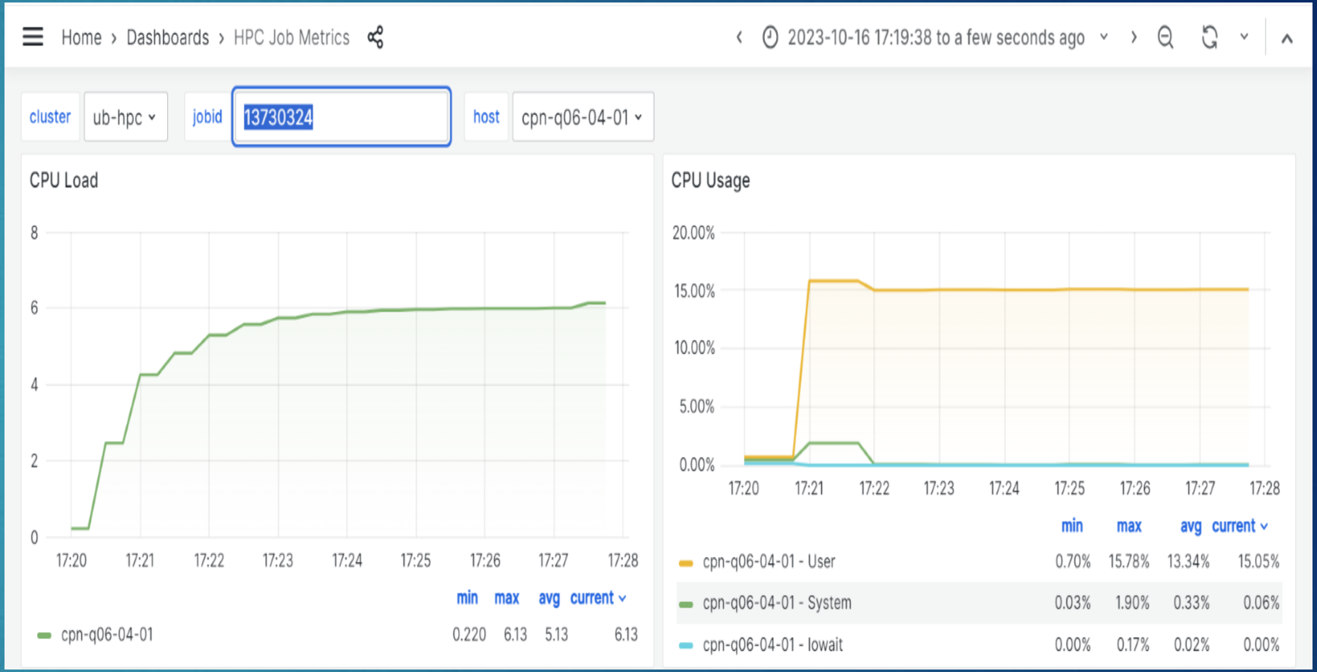
Implemented parallel matrix multiplication algo with 4 processors on 8000x8000 matrices. Running time was approximately 10 minutes, showing improvement over the 2-processors execution. CPU utilization peaked at 26% and averaged around 23%, nearly double that of the 2-processors execution.

Parallel 6 processors

Tabular Data

Processors	Input size	nodes	Cores per node	Time
6	8000*8000	1	6	00:08:35

Graph



Home › Dashboards › HPC Job Metrics

2023-10-16 17:19:38 to a few seconds ago

cluster: ub-hpc
jobid: 13730324
host: cpn-q06-04-01

CPU Load

	min	max	avg	current
cpn-q06-04-01	0.220	6.13	5.13	6.13

CPU Usage

	min	max	avg	current
cpn-q06-04-01 - User	0.70%	15.78%	13.34%	15.05%
cpn-q06-04-01 - System	0.03%	1.90%	0.33%	0.06%
cpn-q06-04-01 - Iowait	0.00%	0.17%	0.02%	0.00%

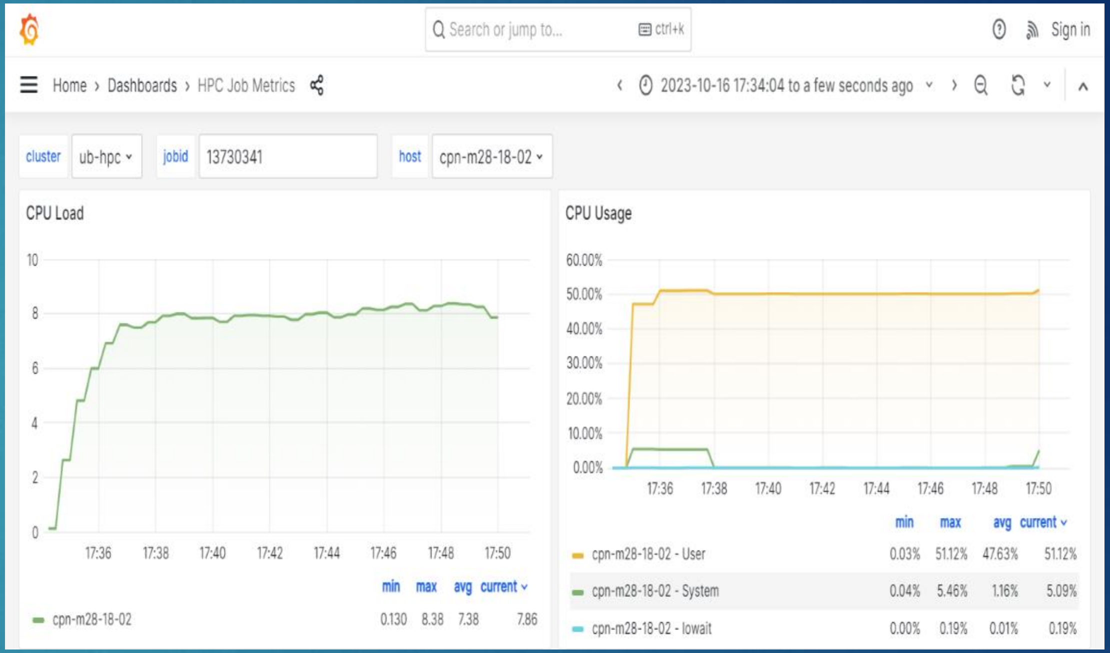
Implemented parallel matrix multiplication with 6 processors on 8000x8000 matrices. Running time was approximately 8 minutes, an improvement over the 4-processors execution. CPU utilization peaked at 15% and averaged around 13% for 6 processors.

Parallel 8 processors

Tabular Data

Processors	Input size	nodes	Cores per node	Time
8	8000*8000	1	8	00:014:35

Graph



2023-10-16 17:34:04 to a few seconds ago

cluster: ub-hpc · jobid: 13730341 · host: cpn-m28-18-02

CPU Load

	min	max	avg	current
cpn-m28-18-02	0.130	8.38	7.38	7.86

CPU Usage

	min	max	avg	current
cpn-m28-18-02 - User	0.03%	51.12%	47.63%	51.12%
cpn-m28-18-02 - System	0.04%	5.46%	1.16%	5.09%
cpn-m28-18-02 - Iowait	0.00%	0.19%	0.01%	0.19%

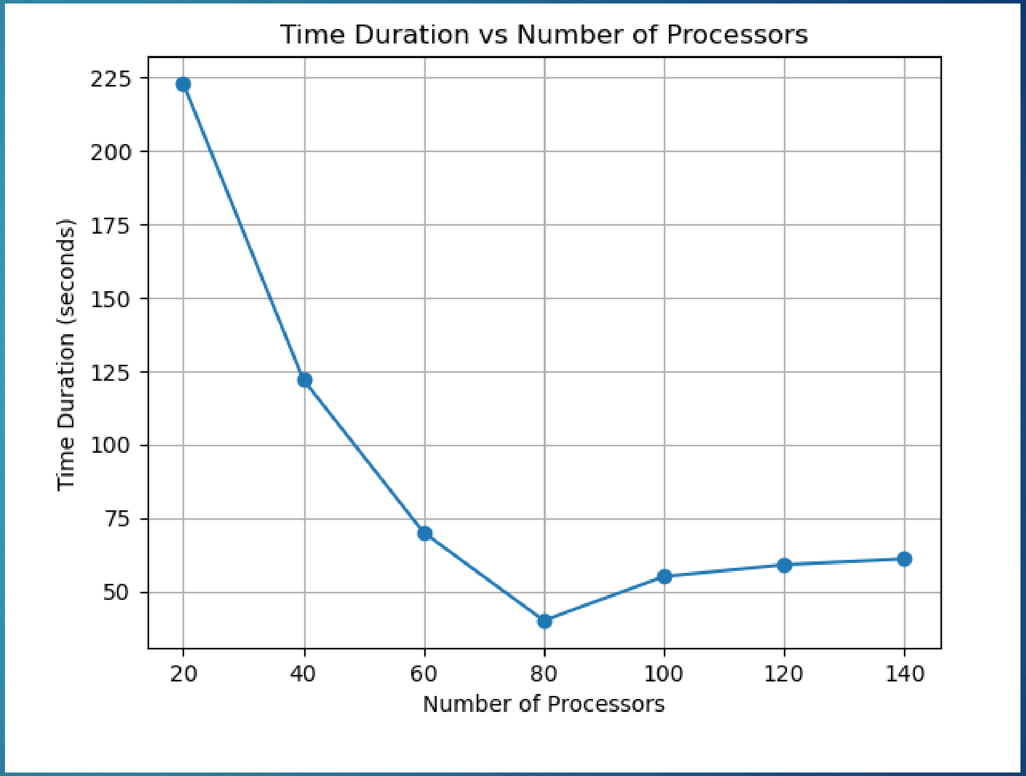
Implemented parallel matrix multiplication with 8 processors on 8000x8000 matrices. Running time was approximately 15 minutes, slightly more than the 4 and 6-processors executions. CPU utilization peaked at 51% and averaged around 47%, likely influenced by inter-process communication overhead.

Setup - 1 Node

Tabular Data

Processors	Input size	nodes	Cores per node	Time
20	16000*16000	1	20	00:03:43
40	16000*16000	1	40	00:02:02
60	16000*16000	1	60	00:01:10
80	16000*16000	1	80	00:00:40
100	16000*16000	1	100	00:00:55
120	16000*16000	1	120	00:00:59
140	16000*16000	1	140	00:01:00

Graph



Implemented parallel matrix multiplication algo from 20 to 140 processors on 16000 x 16000 matrices. the running time decreased with an increasing number of cores up to 80 processors. Beyond this point, the running time increased due to communication overhead. The threshold for optimal performance was identified at 80 processors or cores per node

Key Observations

- Running Time: The running time decreased significantly on the parallel implementation.
- Running Time: The running time kept on decreasing while increasing the no of cores from 2 processors to 4 processors and 4 processors to 6 processors.
- Threshold: As we know that after a certain threshold, parallelism does not help in speedup due to overhead of inter-process communication.
- Threshold point: While running on 8 processors, the running time increased, It kept on increasing further with more cores.
- CPU Utilization: It almost doubled on the parallel Implementation
- CPU Utilization: It increased further on increasing num of cores from 2 to 4.

Next Steps

- As of now, Matrix A is only splitted among various processors, going forward would break down Matrix B too into chunks and allocate it to processors.

Parallel Approach (Fully implemented)

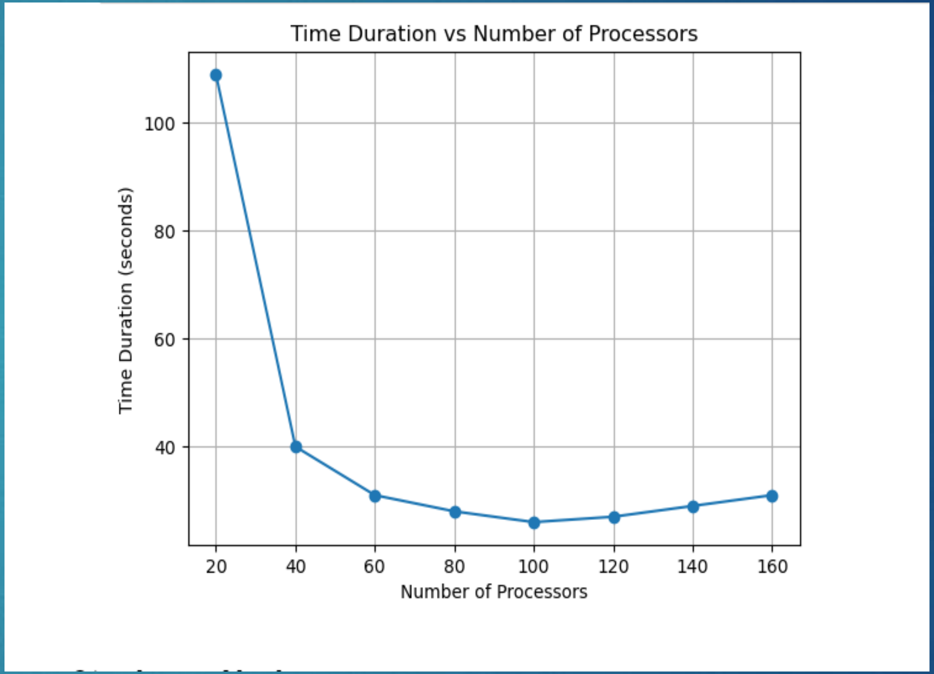
- Given Matrix A and B of size N*N, and we have p no of processors
- The workload could be divided such that-
- Each processor is responsible for (N/p) rows of matrix A and (N/p) cols of matrix B.
- Matrix A is partitioned into (N/p) equally-sized vertical strips, and each processor is assigned one of these strips.
- Matrix B is similarly partitioned into (N/p) equally-sized horizontal strips, and each processor is assigned one of these strips.
- Each processor performs local matrix multiplication on its assigned portion of A and B.
- The local results from each processor are gathered and combined to construct the final result matrix C.

Setup1 - 1 core per Node

Tabular Data

Processors	Input size	nodes	Cores per node	Time
20	16000*16000	20	1	00:01:49
40	16000*16000	40	1	00:00:40
60	16000*16000	60	1	00:00:31
80	16000*16000	80	1	00:00:28
100	16000*16000	100	1	00:00:26
120	16000*16000	120	1	00:00:27
140	16000*16000	140	1	00:00:29
160	16000*16000	160	1	00:00:31

Graph



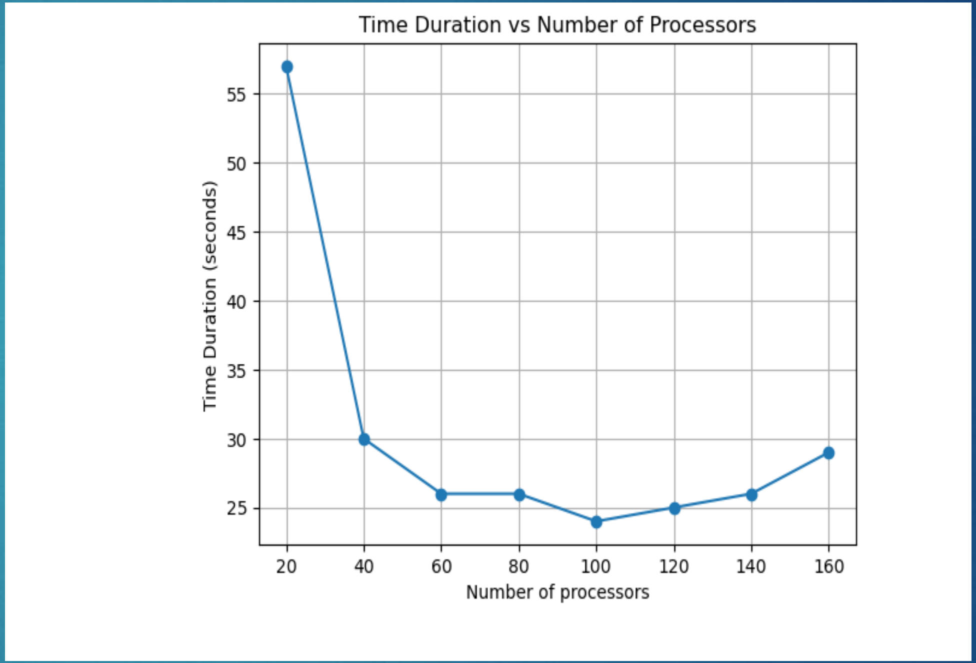
In a single core per node setup, the parallel algorithm was tested with 16000*16000 matrices, the running time decreased with an increasing number of cores up to 80 processors. Beyond this point, the running time increased due to communication overhead. The threshold for optimal performance was identified at 80 processors or cores per node

Setup 2 - 2 cores per Node

Tabular Data

Processors	Input size	nodes	Cores per node	Time
20	16000*16000	10	2	00:00:57
40	16000*16000	20	2	00:00:30
60	16000*16000	30	2	00:00:26
80	16000*16000	40	2	00:00:26
100	16000*16000	50	2	00:00:24
120	16000*16000	60	2	00:00:25
140	16000*16000	70	2	00:00:26
160	16000*16000	80	2	00:00:29

Graph



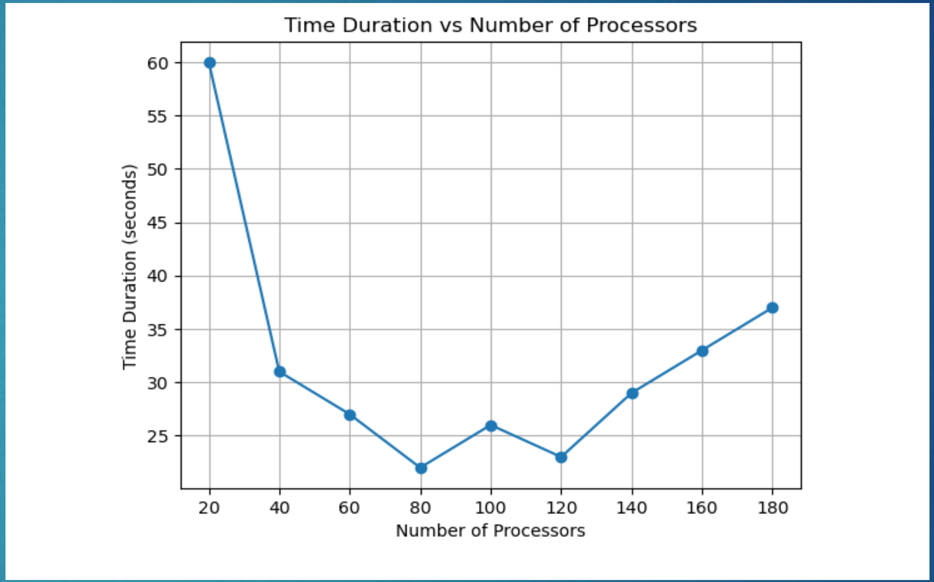
In multi core per node setup, the parallel algorithm was tested with 16000x16000 matrices. The running time decreased with an increasing number of cores up to 100 processors. Beyond this point, the running time increased due to communication overhead. The threshold for optimal performance was identified at 100 processors using 2 cores per node.

Setup 3 - 4 cores per Node

Tabular Data

Processors	Input size	nodes	Cores per node	Time
20	16000*16000	5	4	00:01:00
40	16000*16000	10	4	00:00:31
60	16000*16000	15	4	00:00:27
80	16000*16000	20	4	00:00:22
100	16000*16000	25	4	00:00:26
120	16000*16000	30	4	00:00:23
140	16000*16000	35	4	00:00:29
160	16000*16000	40	4	00:00:33
180	16000*16000	45	4	00:00:37

Graph



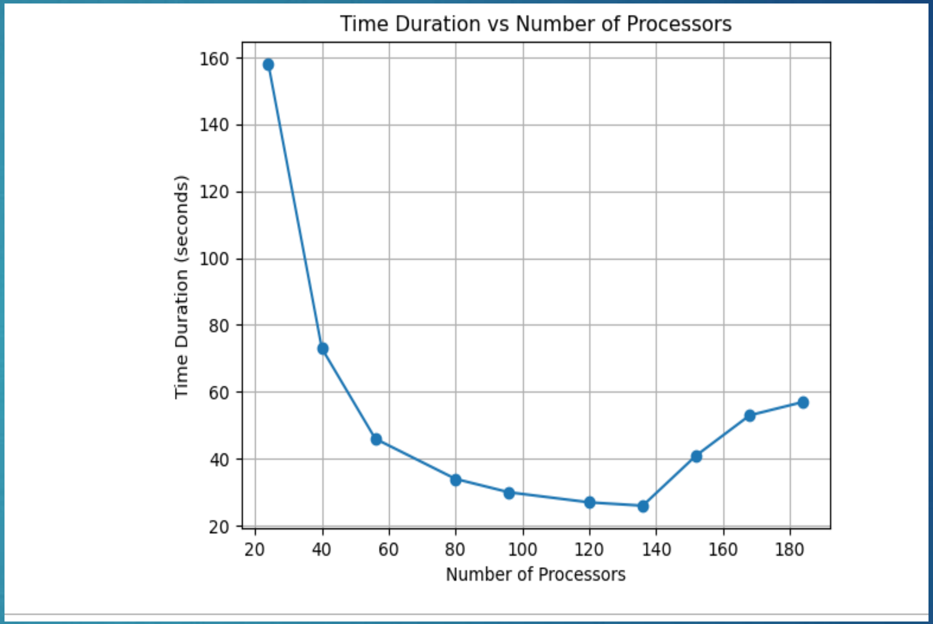
In multi core per node setup, the parallel algorithm was tested with 16000x16000 matrices. The running time decreased with an increasing number of cores up to 80 processors. Beyond this point, the running time increased due to communication overhead. The threshold for optimal performance was identified at 80 processors using 4 cores per node.

Setup 4 - 8 cores per Node

Tabular Data

Processors	Input size	nodes	Cores per node	Time
24	16000*16000	3	8	00:02:38
40	16000*16000	5	8	00:01:13
56	16000*16000	7	8	00:00:46
80	16000*16000	10	8	00:00:34
96	16000*16000	12	8	00:00:30
120	16000*16000	15	8	00:00:27
136	16000*16000	17	8	00:00:26
152	16000*16000	19	8	00:00:41
168	16000*16000	21	8	00:00:53
184	16000*16000	23	8	00:00:57

Graph



In multi core per node setup, the parallel algorithm was tested with 16000x16000 matrices. The running time decreased with an increasing number of cores up to 136 processors. Beyond this point, the running time increased due to communication overhead. The threshold for optimal performance was identified at 136 processors using 8 cores per node.

Key Observations

- Running Time: The running time decreased significantly on the parallel implementation.
- Running Time: The running time decreased significantly while increasing the no of cores per node from 2 to 4 cores and 4 to 6 cores and 6 to 8 cores.
- Threshold: As we saw that after a certain threshold, parallelism does not help in speedup due to the overhead of inter-process communication.
- Threshold point: While running on more than 8 cores per node, the running time increased, It kept on increasing further with more nodes.

References

- MPI Tutorial- https://mpitutorial.com/tutorials/
- MPI Docs- https://www.mpi-forum.org/docs/
- Dr Jones Lectures on MPI
- Matplot lib- https://matplotlib.org/

Thank you!!