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 number of columns of the first matrix $A[n][n]$ equals the number of rows of the second matrix $B[m][m]$.
- Initialize an empty resulting matrix $C$.
- Create three nested for loops, the outermost loop would iterate through the number of rows of the first matrix.
- The second loop would iterate through the number of columns of the second matrix.
- The third loop would iterate through the number of rows of the 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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Input size</th>
<th>nodes</th>
<th>Cores per node</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8000*8000</td>
<td>1</td>
<td>1</td>
<td>00:25:35</td>
</tr>
</tbody>
</table>

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.
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.
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.
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.
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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Input size</th>
<th>Nodes</th>
<th>Cores per node</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>8000x8000</td>
<td>1</td>
<td>6</td>
<td>00:08:35</td>
</tr>
</tbody>
</table>

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.
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

<table>
<thead>
<tr>
<th>Processors</th>
<th>Input size</th>
<th>nodes</th>
<th>Cores per node</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>16000*16000</td>
<td>1</td>
<td>20</td>
<td>00:03:43</td>
</tr>
<tr>
<td>40</td>
<td>16000*16000</td>
<td>1</td>
<td>40</td>
<td>00:02:02</td>
</tr>
<tr>
<td>60</td>
<td>16000*16000</td>
<td>1</td>
<td>60</td>
<td>00:01:10</td>
</tr>
<tr>
<td>80</td>
<td>16000*16000</td>
<td>1</td>
<td>80</td>
<td>00:00:40</td>
</tr>
<tr>
<td>100</td>
<td>16000*16000</td>
<td>1</td>
<td>100</td>
<td>00:00:55</td>
</tr>
<tr>
<td>120</td>
<td>16000*16000</td>
<td>1</td>
<td>120</td>
<td>00:00:59</td>
</tr>
<tr>
<td>140</td>
<td>16000*16000</td>
<td>1</td>
<td>140</td>
<td>00:01:00</td>
</tr>
</tbody>
</table>

**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.
Running Time: The running time decreased significantly on the parallel implementation.

Running Time: The running time kept on decreasing while increasing the number 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 the number of cores from 2 to 4.
Next Steps

- As of now, Matrix A is only split 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.
In a single core per node setup, the parallel algorithm was tested with $16000 \times 16000$ 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.
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.
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

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

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.
Running Time: The running time decreased significantly on the parallel implementation.

Running Time: The running time decreased significantly while increasing the number 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
- Matplotlib- https://matplotlib.org/
Thank you!!