David Rohweder
Parallelization Analysis
Penn State University
State College, PA, USA
Djr6005@psu.edu
April 29, 2022

1 Introduction

For this research we will attempt to reduce the execution time of large-scale matrix-matrix multiplication utilizing a distributed memory processing. To begin our analysis, we will first need to give the specifics of the equipment and tools used in the experiment. First, the study was executed on the Penn State Roar supercomputer using the ACI-b Standard Core resource profile and queued batch jobs to fully utilize open allocation on the cluster. The allocation for this setting is as follows: 20 available cores, InfiniBand interconnect, and 256 GB total RAM (for open allocation users). More specifically, the VM is using an Intel Xeon CPU E5-2680 v3 @ 2.5GHZ with 32KB L1 cache, 256KB L2 cache, and 25600KB L3 cache. Moreover, the program was run on several dozen ACI-b nodes throughout the trials of the code. It is important to note that this experiment utilizes a single core for program execution because we aim to study the effects distributed memory can have on code performance. Next, we will discuss the aims of the study by analyzing distributed memory processing. The first scenario the study will be exploring is the implementation of the algorithm, followed by an analysis of how the programs functionality was verified. After, the code performance will be analyzed in comparison to other types of parallel programming techniques such as threaded environments and programs using the OpenMP standard. Finally, we will conduct a stress test on the program to see the limits of MPI and distributed memory processing.

2 Analysis

Implementation

When implementing Cannon's algorithm in MPI there were several factors to consider. The first factor was that each of the nodes was processing the algorithm at the same time. Therefore, how/ when large matrices were declaring, when barriers were used, and appropriate communication of data was critical to ensuring success.

To begin, the major matrices A, B, and C were created of size NxN and along with them sub-matrices A',B' and C' were created of size $N/\sqrt{P}$. The first challenge that was overcome during the experiment was dividing the major matrices into the sub-matrices and equally diving each node the correct corresponding matrix for A and B since the C' was local to each. To accomplish this, MPI_Scatterv was used to send each A' and B' to each node performing the code execution. Next, creating an efficient communication schema was needed to implement Cannon's algorithm. This was accomplished by using the MPI_Cart topology and creating a 2-dimensional grid of the processing nodes. The cartesian topology allows for an efficient communication schema to not only to perform circular rotations of the nodes, but also efficiently send data blocks to other nodes. This is because the Cart topology makes finding neighboring nodes in the 2D grid easy as well as shifting nodes and shifting matrices by coordinate offsets. The skew phase for A and B (A shifts by row number, B shifts by column number) was done with simple MPI_Cart_Shifts and the MPI_Sendrev_Replace functions. Also, the local C' matrix multiplications were easy and was implemented using a serial matrix-matrix multiplication function since threads were not considered. After the local C' matrices were computed, the matrices were replaced with neighboring matrices as A moved circular left and B moved circular upwards. After the local computations were performed for $\sqrt{P}$ times the original A and B sub-matrices were restored in the nodes by setting back to the default buffer pointers.

After the MPI Cannon's Algorithm was executed, on the root node (rank = 0), a serial computation was performed on the A, B, and C matrices. Then the output C of MPI Cannon was compared element by element to the serial matrix C to test validity. The output was tested on several P and N sizes and was confirmed correct each instance.

Relative Performance

The introduction of MPI served as a different way of processing data concurrently. Other methods seen, compared, and tested include standard threads (also known as pthreads) and OpenMP (shared memory processing). Since MPI works on the basis of communication over a fast network to compute nodes in an SIMD manner over something like OpenMP with SMP the implementations are much different and with that have different performance Results.

In parallel algorithms such as standard threads or OpenMP we see initialization penalty's that are more impactful for smaller datasets but become less relevant for medium to large datasets as the application is scaled. However, these types of parallel environments normally are executed on a singular node which has resource limitations such as RAM. When the code is scaled too large, a crash is inevitable as there are limited resources. This is where MPI has better performance. While MPI has far more communication overhead compared to OpenMP and standard threads it can handle much larger concurrent applications because the processing is computed on several nodes or environments with separate resources. Since the data is distributed however and uses the network to send data it is far less efficient for small to medium workloads. There are comparable results seen in the parallel methods on large workloads as the overhead begins to even out more evenly.

Overall Performance

Since in Relative Performance, it was discovered that MPI is best suited for heavily intensive workloads, the question arises of how much can it handle? Or simply, what are the limits of MPI? This question was tested during the code analysis of Cannon's algorithm. At limits the ACI-b become unresponsive when ran at P = 100 and N = 250,000. This response is interesting because when ran at N = 250,000 the actual matrix size is 62.5B and the total operations is N^3 or 1.56E16. No shared memory system such as OpenMP would be able to ensure such feats, again showing the need and niche where MPI lies. Since 100 tasks were all processing the data in tandem the task was able to be accomplished without completely crashing under the intensive load.

3 Conclusion

While MPI is a niche consideration when implementing algorithms, it has shown that it holds the best under massive workloads and increases with performance up to that point. At large, but not overwhelming workloads MPI can still hold its own against parallel techniques such as standard threading or OpenMP; however, the initialization penalty also seen by those algorithms is far less than what is seen in MPI. Furthermore, MPI, while maximizing efficiency of communication is still at smaller workloads outclassed by faster, closer hardware to the CPU as MPI data is sent over network. When demonstrating Cannon's algorithm, it was showed that MPI was able to hold under pressure and compute massive workloads not possible by OpenMP or standard threading. When considering what parallel technique to use it is clear that the size and difficulty of the problem are essential to maximizing performance as several factors play into how the data is processed and what can be most efficient for different workloads.