Project 3 Report for Probelm 3.1

Zhu Liang

October 25, 2023

1 Project Description

The primary objective of this project is to implement matrix multiplication using the Ring (a.k.a. Cannon) algorithm. The performance of this method is then evaluated across varying matrix sizes and numbers of processors. The program is implemented in C and compiled by the shell script `test.sh`.

The initial results from the main segment hints at potential cache effects influencing the performance metrics. To offer a clearer and unbiased perspective on the algorithm's performance, additional tests were conducted, aiming to minimize this cache influence. These supplementary tests and their outcomes are detailed in Appendix.

2 Algorithm Description

Algorithm 1 Ring Algorithm for Matrix Multiplication

```
 1: procedure RingMatrixMultiply($N, P, A, B$)
 2:     Initialize MPI with rank and size $P$
 3:     Determine local matrix sizes based on $N$ and $P$
 4:     Transpose matrix $B$
 5:     Scatter matrix $A$ and $B$ to all processor from root
 6:     for each rotation, all processors do
 7:         Perform local matrix multiplication of $A$ and $B$
 8:         Store current results on local $C$ matrix
 9:         Shift local $B$ matrix to the left by one processor
10:     end for
11:     Gather all local results to root
12:     if rank is root then
13:         Formulate the final result matrix and return
14:     else
15:         Return NULL
16:     end if
17: end procedure
```

Matrix multiplication is a cornerstone in numerous computational tasks. Efficiently parallelizing this operation can considerably expedite computations, particularly with large matrices. The ring algorithm offers a parallel approach for this. At its core, the algorithm scatters matrices across multiple processors. During each iteration, each processor computes a local multiplication, stores the result, and then shifts its portion of matrix to the neighboring processor. This rotation strategy ensures that each processor multiplies with all parts of the other matrix over several iterations. The pesudocode for this algorithm is shown in Algorithm 1.

It's worth noting that this specific implementation slightly deviates from what's typically taught in class. Instead of rotating matrix A, we opt to rotate matrix B. This choice was driven by the convenience it provides in gathering the final result matrix using the `MPI_Gather` function under our costomized matrix data structure.

This description is intentionally streamlined, omitting many details and supporting functions for brevity. More details can be gleaned from the source code file `functions.c`.

3 Results

3.1 Performance

	Ring Algorithm			Naive Algorithm
(in seconds)	$P = 2^2$	$P = 2^4$	$P = 2^6$	$P = 1$
$N = 2^8$	0.020182	0.006294	0.003478	0.090545
$N = 2^{10}$	1.163175	0.310421	0.095925	6.119090
$N = 2^{12}$	73.364430	18.900791	5.574106	707.989130

Table 1: Performance of the Ring and Naive methods.

As demonstrated in Table 1, the performance of the Ring and Naive matrix multiplication methods is outlined for different matrix sizes (N) and varying number of processors (P). The results indicate the time taken by each method to complete the matrix multiplication for the specified configurations.

3.2 Speedup

Speedup is a metric that quantifies the performance improvement of a parallel algorithm compared to its sequential counterpart. The speedup S for P processors for size N problem is defined as:

$$S(P, N) = \frac{T(1, N)}{T(P, N)} \tag{1}$$

where $T(1, N)$ is the execution time of the sequential algorithm and $T(P, N)$ is the execution time of the parallel algorithm using P processors.

Using the data from Table 1, we can compute the speedup for matrix multiplication for different matrix sizes N and varying number of processors P. The speedup values for each N are plotted in the following Figure 1.

For $N = 2^8$, the curve exhibits a normal speedup where $S(P, N) < P$, lying below the ideal diagonal. For $N = 2^{10}$, the speedup closely follows the ideal diagonal, indicating an almost perfect linear speedup. Interestingly, for $N = 2^{12}$, the curve is above the diagonal, suggesting a superlinear speedup, which is an rarel phenomenon.

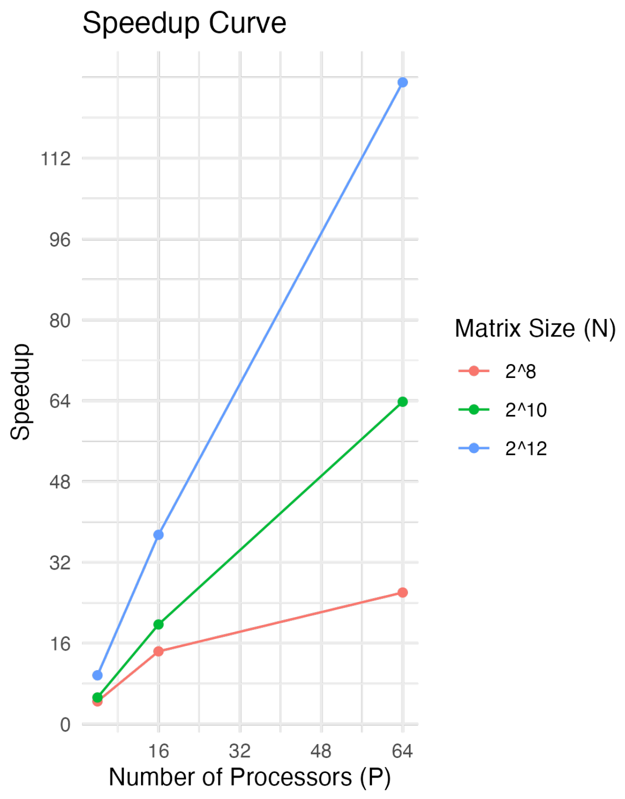

Figure 1: Speedup Curve for Matrix Sizes N

4 Analysis

Overall, the parallel acceleration is remarkably significant. Interestingly, the speedup characteristics are not consistent across the three matrix sizes N. One intriguing observation is the superlinear speedup exhibited when the matrix size is large ($N = 2^{12}$).

One hypothesis for this phenomenon is the cache effect. When N is relatively small, such as 2^8, assuming each element of the matrices A and B occupies 4 bytes as a floating-point number, the combined space required for matrices would be $2 \times 2^8 \times 2^8 \times 4$ bytes, or 512 KB, which is not substantial. Such data could potentially be handled directly within the cache. As a result, the parallel computation exhibits a typical speedup, with communication overheads significantly affecting performance.

However, for $N = 2^{12}$, the combined space for matrices A and B becomes $2 \times 2^{12} \times 2^{12} \times 4$ bytes, or 128 MB. This is a substantial amount of data, and excesses the cache size of most modern processors. The total space rockets up, potentially preventing direct cache handling when computed with a single core. This would necessitate direct memory access, significantly slowing down computation.

Consequently, a nonlinear performance drop is observed for single-core execution when transitioning from $N = 2^{10}$ to $N = 2^{12}$. In contrast, during parallel computation, each core only deals with a fraction of the matrix at a time. This fraction might still fit within the cache, avoiding direct memory reads and substantially boosting speed. The drag due to communication overhead becomes relatively minimal in this scenario.

In conclusion, the observed performance benefits from the ring algorithm, especially in the $N = 2^{12}$ case, highlight the potential impacts of faster memory hierarchies in parallel computing.

A Supplementary Tests

To mitigate the influence of cache effects on the speedup results, we redefined our baseline to $P = 4$ and gathered results using a different set of P values to plot the speedup curve.

A.1 Performance

	Ring Algorithm				
(in seconds)	$P = 2^2$	$P = 2^3$	$P = 2^4$	$P = 2^5$	$P = 2^6$
$N = 2^8$	0.039433	0.020771	0.011038	0.009600	0.006572
$N = 2^{10}$	2.123300	1.203297	0.635084	0.353849	0.175946
$N = 2^{12}$	73.409207	37.168124	20.105683	10.941110	6.431338

Table 2: Updated performance of the Ring method.

Note that when we take $P = 4$ as the baseline, the maxima data size for each processor shrink by a factor of 4, which is 32MB. This is a more reasonable size for the cache, and the performance is not expected to drop significantly as N increases.

A.2 Updated Speedup

With the updated performance data, we can compute the updated speedup values for each N based on the modified defination in Equation 2 with the performance results in Table 2.

$$S(P, N) = 4 \times \frac{T(P, N)}{T(4, N)} \tag{2}$$

The updated speedup values for each N are plotted in the following Figure 2.

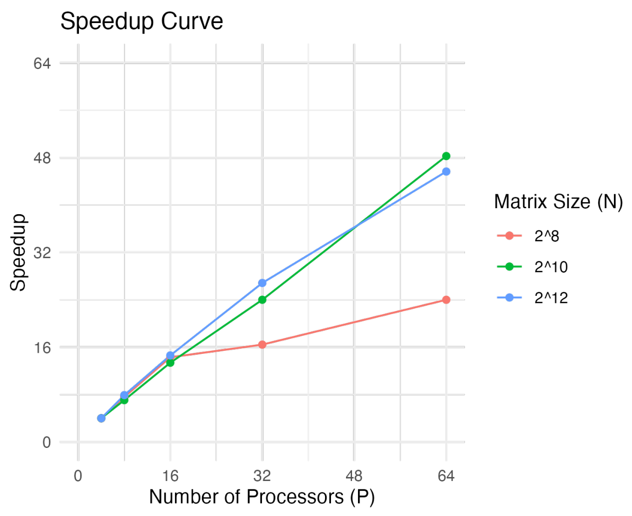

Figure 2: Updated Speedup Curve for Matrix Sizes N

A.3 Analysis

The updated speedup curve is much more consistent with regular speedup curves. All of the speedup values are below the ideal diagonal, with the speedup for $N = 2^{12}$ and $P = 2^{10}$ being above the speedup for $N = 2^8$.

This suggests that when the matrix size is small, the overhead of communication is significant, and the parallel computation becomes more effeicient when the matrix size increases.

In summary, while the updated speedup curve does exhibit benefits from parallel processing, especially for larger matrices, there remains a balance between computation and communication that determines the overall performance gains.

File Notes

The source code of the program is in the `project3` folder. For more details, please refer to the `README.md` file in the folder.