Project 4 Report for Probelm 4.1

Zhu Liang

November 16, 2023

1 Project Description

The primary objective of this project is the implementation of the parallel Strassen algorithm for matrix multiplication across three levels (1, 2, and 3), focusing on minimizing memory usage and communication costs.

Our approach utilizes a hierarchical multi master-slave tree (or leader-worker tree) structure. Each layer comprises a single leader and seven workers, where workers can serve as leaders for the subsequent layer, facilitating the next level of the Strassen algorithm. This design allows the leader to distribute specific submatrices to its workers, who later return their computed results. By limiting data exchange to only necessary data between leaders and workers, our implementation effectively reduces communication overhead and optimizes memory consumption.

2 Algorithm Description

The parallel Strassen algorithm employs a divide-and-conquer approach to decompose matrix multiplication into smaller, manageable subproblems. These subproblems are distributed across multiple processors for parallel computation. After processing, the results are aggregated and synthesized into the final product. This recursive method continues until the subproblems reach a size conducive to sequential computation.

Our MPI-based implementation is a three-step process within a master-slave tree structure:

1. Distribute submatrices to worker processors, descending to the targeted level.
2. Perform matrix multiplication at each worker processor on the targeted level.
3. Relay results back up to the leader processor for aggregation, culminating at the *root*.

The following details the pseudocode for our implementation, beginning with an overview of the algorithm, followed by the data distribution and result collection functions.

Algorithm 1 Strassen Multiply Parallel

```
1: procedure StrassenMultiplyParallel(A, B, N, max_level)
2:     for level ← 1 to max_level do
3:         DistributeData(level, rank)
4:     end for
5:     Perform matrix multiplication on all workers.
6:     for level ← max_level down to 1 do
7:         CollectResults(level, rank)
8:     end for
9:     return final result on ROOT
10: end procedure
```

2.1 Data Distribution

The `distribute_data` function assigns submatrices to worker processors. This process starts at the *root* processor and continues recursively to the target level. The leader processor generates pairs of submatrices (M_i^A and M_i^B) and distributes them to its workers. The submatrices are defined as follows:

M_i^A	M_i^B	M_i	Assignment
$M_1^A = (A_{11} + A_{22})$	$M_1^B = (B_{11} + B_{22})$	$M_1 = M_1^A \times M_1^B$	$\rightarrow$ Worker 1
$M_2^A = (A_{21} + A_{22})$	$M_2^B = B_{11}$	$M_2 = M_2^A \times M_2^B$	$\rightarrow$ Worker 2
$M_3^A = A_{11}$	$M_3^B = (B_{12} - B_{22})$	$M_3 = M_3^A \times M_3^B$	$\rightarrow$ Worker 3
$M_4^A = A_{22}$	$M_4^B = (B_{21} - B_{11})$	$M_4 = M_4^A \times M_4^B$	$\rightarrow$ Worker 4
$M_5^A = (A_{11} + A_{12})$	$M_5^B = B_{22}$	$M_5 = M_5^A \times M_5^B$	$\rightarrow$ Worker 5
$M_6^A = (A_{21} - A_{11})$	$M_6^B = (B_{11} + B_{12})$	$M_6 = M_6^A \times M_6^B$	$\rightarrow$ Worker 6
$M_7^A = (A_{12} - A_{22})$	$M_7^B = (B_{21} + B_{22})$	$M_7 = M_7^A \times M_7^B$	$\rightarrow$ Worker 7

Table 1: Submatrices Distributed to Each Worker

All M_i^A and M_i^B submatrices are of the same size and are sent to seven cores for computation. The generation and calculation of each M_i^A and M_i^B occur at the leader core. This approach ensures uniformity in the format of the distributed data and facilitates further distribution. Although it leads to a non-parallel computation of certain addition and subtraction operations, these operations are computationally less intensive compared to multiplication. Hence, we believe the benefits outweigh the costs.

Algorithm 2 Distribute Data

```
1: procedure DistributeData(level, rank)
2:     if rank is leader under level then
3:         Send all M_i^A, M_i^B to its worker i
4:     end if
5:     if rank is worker under level then
6:         Receive M_i^A, M_i^B from its leader
7:     end if
8: end procedure
```

2.2 Result Collection

The `collect_results` function is responsible for gathering computation outcomes from worker processors and assimilating them into a unified matrix. This reverse hierarchical procedure commences at the designated target level and ascends to the *root* processor. Each leader processor acquires partial results (M_1 to M_7) from its subordinates and integrates them to construct the final matrix. The integration is delineated as follows:

Submatrix	Aggregated Components
C_{11}	$M_1 + M_4 - M_5 + M_7$
C_{12}	$M_3 + M_5$
C_{21}	$M_2 + M_4$
C_{22}	$M_1 - M_2 + M_3 + M_6$

Table 2: Aggregation of Results from Worker Processors

This table succinctly illustrates the process of assembling the final matrix from the individual components calculated by the workers.

Algorithm 3 Collect Results

```
1: procedure CollectResults(level, rank)
2:     if rank is leader under level then
3:         Receive all M_i from its workers
4:         Aggregate M_i to C
5:     end if
6:     if rank is worker under level then
7:         Send M_i to its leader
8:     end if
9: end procedure
```

As outlined in Algorithms 2 and 3, establishing the leader-worker relationships for each processor at different levels is crucial. This is achieved through a recursive tree structure, where the relationships are dictated by the processors' ranks. These relationships remain consistent across all levels. For an in-depth understanding of this tree structure, please refer to the Appendix A.

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

Notice that the level 1, 2, 3 of the parallel Strassen algorithm requires 7, 49, 343 processors respectively to fully utilize the parallelism. We will run the parallel Strassen algorithm with 7, 49, 343 processors respectively.

(in seconds)	Parallel Strassen Algorithm		
	level 1 (cores = 7)	level 2 (cores = 49)	level 3 (cores = 343)
$N = 2^8$	0.029430	0.010317	0.040137
$N = 2^{10}$	1.757423	0.284435	0.274502
$N = 2^{12}$	63.811983	9.049598	2.902417

Table 3: Performance of the Parallel Strassen Algorithm at different levels.

3.2 Speedup

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

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

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.

However, in our case, the sequential algorithm on single core is not implemented. Instead, we use the performance of the parallel algorithm with $P = 7$ as the baseline. The updated speedup S is defined as:

$$S(P, N) = 7 \times \frac{T(7, N)}{T(P, N)} \tag{1}$$

Using the data from Table 3, we can compute the speedup for matrix multiplication for different matrix sizes N and varying number of processors P (under level 1, 2, 3). Since the baseline is the performance of the parallel algorithm with $P = 7$, the speedup values are computed using Equation 1. The speedup values for each N (under level 2 and level 3) are plotted in the following Figure 1.

4 Analysis

The speedup curve, as shown in Figure 1, demonstrates the scalability of our parallel Strassen algorithm with varying numbers of processors. An ideal linear speedup is indicated by a dashed line, while the solid lines represent the actual speedup for different matrix sizes. The observed trends can be summarized as follows:

For $N = 2^8$, the speedup decreases with additional processors, suggesting that the overhead of communication dominates the benefits of parallel computation due to the small size of the matrices.

When $N = 2^{10}$, the speedup curve plateaus, indicating that the computational load and communication overhead reach a balance.

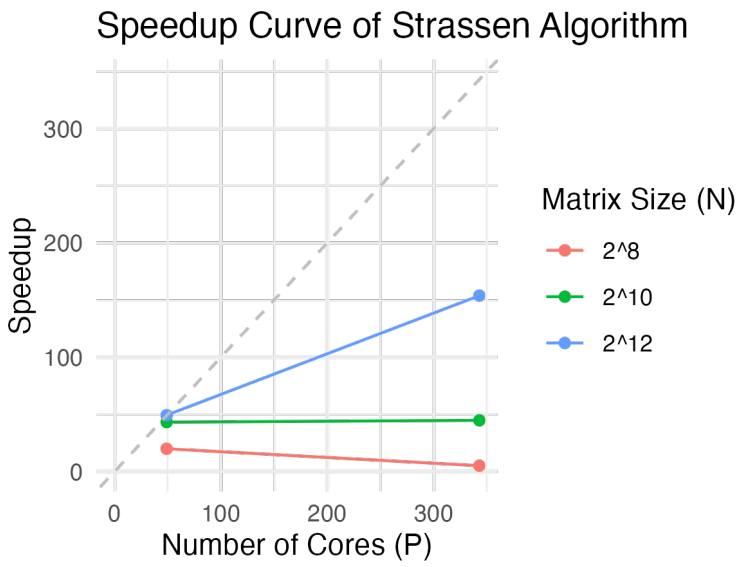

Figure 1: Speedup Curve for Matrix Sizes N

For $N = 2^{12}$, there is a noticeable increase in speedup, implying that the computation benefits substantially from parallel execution.

The underlying causes for these trends can be attributed to two main factors:

- **Communication Overhead:** As the number of processors increases, the communication overhead can become a bottleneck, especially for smaller matrices where the communication cost outweighs the computation time.
- **Computation to Communication Ratio:** For larger matrices, the computation to communication ratio improves, allowing the algorithm to better exploit the parallel architecture. This results in a higher speedup as the computation workload per processor increases, which is more pronounced for $N = 2^{12}$.

This analysis highlights the critical balance between computation and communication in parallel algorithms and will guide our efforts in optimization, focusing on minimizing communication costs and improving load distribution among processors.

A Tree Structure for Leader-Worker Relationships

The leader-worker relationships within the parallel Strassen algorithm are organized in a hierarchical tree structure. This structure is pivotal for determining the flow of data distribution and result collection across different levels of computation. The tree structure is recursive, with each leader processor being connected to a set of worker processors. The following table illustrates these relationships:

Leader Rank	**Worker Ranks**
0	[0, 1, 2, 3, 4, 5, 6]
1	[7, 8, 9, 10, 11, 12, 13]
2	[14, 15, 16, 17, 18, 19, 20]
3	[21, 22, 23, 24, 25, 26, 27]
4	[28, 29, 30, 31, 32, 33, 34]
5	[35, 36, 37, 38, 39, 40, 41]
6	[42, 43, 44, 45, 46, 47, 48]
...	...

Table 4: Leader to Worker Relationships in the Tree Structure

Each processor's rank in the tree structure determines its specific role, ensuring consistent leader-worker assignments throughout the parallel computation of the Strassen algorithm. This design reuses leaders from one level as workers in the subsequent level, facilitating the implementation of levels 1, 2, and 3 with 7, 49, and 343 processors, respectively. The hierarchy is pre-established and remains unchanged throughout the process.

To identify the leader and worker ranks at any given level, we employ the `get_leade_rank` and `get_worker_rank` functions. Additionally, the functions `is_leader` and `is_worker` determine whether a processor serves as a leader or a worker at a particular level. For a detailed understanding of these functionalities, please refer to the `functions.c` file in the source code.

A.1 Special Case: *root*

As evident from the leader-worker relationships detailed in , the *root* processor (rank 0) uniquely acts as both sender and receiver simultaneously. This dual role could lead to deadlock situations when employing standard MPI communication methods like `MPI_Isend` and `MPI_Irecv`.

To circumvent this issue, a special handling mechanism is implemented specifically for the *root* processor. Instead of using MPI functions for sending and receiving data, the *root* processor directly writes the relevant submatrices into its memory buffer and later reads from this buffer for further processing. This approach not only prevents deadlock scenarios but also reduces communication overhead, as it eliminates the need for additional message-passing tasks for the *root* processor. By directly manipulating the memory, we maintain the efficiency and fluidity of the algorithm's execution, especially at the top level of the tree structure.

File Notes

The source code of the program is in the `project4` folder. For more details, please refer to the `README.md` file in the folder.