Project of CS7344

1 ADI PROGRAM

This program is used to update each element of a three dims matrix $A^{N\times N\times N}$. Algorithm 1 show the computing process.

Algorithm 1 Serial Adi Program

```
1:  **Input:** $N \times N \times N$ matrix $A$
2:  **Output:** Updated $A$
3:  for $i$ from 0 to $N-1$
4:      for $j$ from 0 to $N-1$
5:          for $k$ from 1 to $N-1$
6:              $A[i][j][k] \leftarrow 0.4 \times A[i][j][k] - 0.6 \times A[i][j][k-1]$
7:  for $i$ from 0 to $N-1$
8:      for $j$ from 1 to $N-1$
9:          for $k$ from 0 to $N-1$
10:             $A[i][j][k] \leftarrow 0.5 \times A[i][j][k] - 0.5 \times A[i][j-1][k]$
11: for $i$ from 1 to $N-1$
12:     for $j$ from 0 to $N-1$
13:         for $k$ from 0 to $N-1$
14:             $A[i][j][k] \leftarrow 0.6 \times A[i][j][k] - 0.4 \times A[i-1][j][k]$
```

From Algorithm 1, it is evident that the entire serial code can be segmented into three parts, each corresponding to a different dimension. Specifically, we update matrix A along the third dimension in the first segment, the second dimension in the second segment, and the first dimension in the third segment. Thus, in each segment, we can parallelize the computation across the other two dimensions. To achieve a communication complexity of $O(N^2)$, we partition A along two dimensions. Due to the row-major storage scheme of the matrix, we partition A on the first and second dimensions. Assuming there are p processes, each process manages p blocks, with each block containing $\frac{N}{p} \times \frac{N}{p} \times N$ elements. Figure 1 illustrates the partitioning method when there are four processes.

Now, we can parallelize the three segments of the serial code sequentially. Each segment will undergo p iterations.

In the first segment, updates occur along the third dimension (z-axis) and partitioning occurs on the first and second dimensions. Consequently, there is no need for inter-process communication. Each process independently identifies its blocks and updates all elements during each iteration.

In the second segment, updates proceed along the second dimension (y-axis). Therefore, processes must exchange data during each iteration. Specifically, after a process r updates its block in the current iteration, it should send its updated data to process $r-1$ (or to process $p-1$ if $r=0$) and receive data from process $r+1$ (or from process 0 if $r=p-1$). For instance, as illustrated in Figure 1, after process 0 completes its updates in iteration 0, it should send data to process 3 and receive data from process 1 for the next iteration.

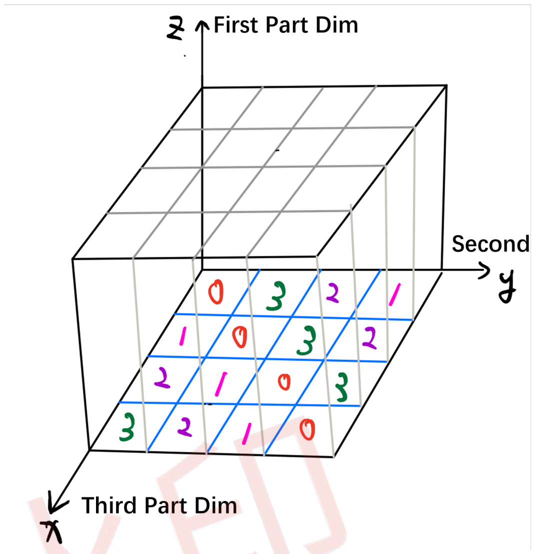


Figure 1: Partitioning the Matrix on the first and second dims with 4 processes

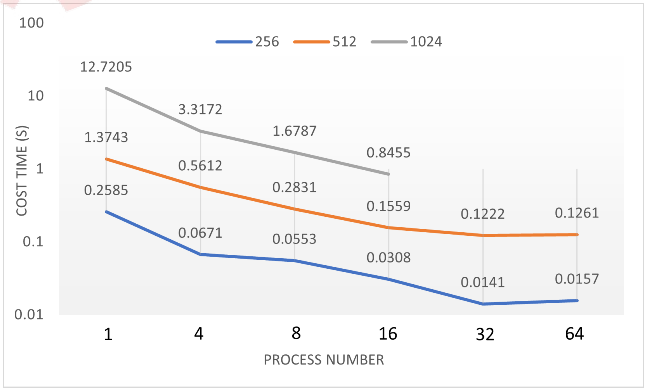


Figure 2: The Cost Time(s) of Algorithm 2 under different number of processes and different N.

In the third segment, updates occur along the first dimension (x-axis). This necessitates data exchanges between processes in each iteration as well. Specifically, after a process r updates its block in the current iteration, it should send its updated data to process $r+1$ (or to process 0 if $r=p-1$) and receive data from process $r-1$ (or from process $p-1$ if $r=0$). For example, as shown in Figure 1, when process 0 completes its updates in iteration 0, it should

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

send data to process 1 and receive data from process 3 for the next iteration.

Algorithm 2 Paralleling Adi Program

```
1: Input: $N \times N \times N$ matrix $A$
2: Output: Updated $A$
3: $r \leftarrow$ MPI_Comm_rank()
4: $p \leftarrow$ MPI_Comm_size()
5: $s \leftarrow \frac{N}{p}$
6: get $r-1$ neighbor rank, left $\leftarrow (r-1+p)\%p$
7: get $r+1$ neighbor rank, right $\leftarrow (r+1)\%p$
8: compute range of the first block for the first part, iterate along $z$ axis, $A[si:ei][sj:ej][0:N]$
9: $si \leftarrow r*s$, $ei \leftarrow si+s$, $sj \leftarrow 0$, $ej \leftarrow sj+s$
10: for $t$ from 0 to $p-1$
11:     for $i$ from $si$ to $ei-1$
12:         for $j$ from $sj$ to $ej-1$
13:             for $k$ from 1 to $N-1$
14:                 $A[i][j][k] \leftarrow 0.4\times A[i][j][k] - 0.6\times A[i][j][k-1]$
15:     move to next block with $si \leftarrow ei\%N$, $ei \leftarrow si+s$, $sj \leftarrow ej$, $ej \leftarrow sj+s$
16: compute range of the first block for the second part, iterate along $y$ axis, $A[si:ei][sj:ej][0:N]$
17: $si \leftarrow r*s$, $ei \leftarrow si+s$, $sj \leftarrow 1$, $ej \leftarrow s$
18: for $t$ from 0 to $p-1$
19:     for $i$ from $si$ to $ei-1$
20:         for $j$ from $sj$ to $ej-1$
21:             for $k$ from 0 to $N-1$
22:                 $A[i][j][k] \leftarrow 0.5\times A[i][j][k] - 0.5\times A[i][j-1][k]$
23:     use MPI_Isend() to send $A[si:ei][ej-1][0:N]$ to process left.
24:     use MPI_Irecv() to receive next block starting data $A[si+s:ei+s][ej-1][0:N]$ from process right.
25:     use MPI_Waitall() to wait receiving.
26:     move to next block with $si \leftarrow ei\%N$, $ei \leftarrow si+s$, $sj \leftarrow ej$, $ej \leftarrow sj+s$
27: compute range of the first block for the third part, iterate along $x$ axis, $A[si:ei][sj:ej][0:N]$
28: $si \leftarrow 1$, $ei \leftarrow s$, $sj \leftarrow ((p-r)\%p)s$, $ej \leftarrow sj+s$
29: for $t$ from 0 to $p-1$
30:     for $i$ from $si$ to $ei-1$
31:         for $j$ from $sj$ to $ej-1$
32:             for $k$ from 0 to $N-1$
33:                 $A[i][j][k] \leftarrow 0.6\times A[i][j][k] - 0.4\times A[i-1][j][k]$
34:     use MPI_Isend() to send $A[ei-1][sj:ej][0:N]$ to process right.
35:     use MPI_Irecv() to receive next block starting data $A[ei-1][sj+s:ej+s][0:N]$ from process left.
36:     use MPI_Waitall() to wait receiving.
37:     move to next block with $si \leftarrow ei$, $ei \leftarrow si+s$, $sj \leftarrow ej\%N$, $ej \leftarrow sj+s$
38: process 0 receive all updated elements from other processes.
39: return Updated $A$
```

Algorithm 3 Serial Floyd Program

```
1: Input: graph matrix $A^{N\times N}$
2: Output: Updated $A$
3: for $k$ from 0 to $N-1$
4:     for $i$ from 0 to $N-1$
5:         for $j$ from 0 to $N-1$
6:             $A[i][j] = \min(A[i][j], A[i][k] + A[k][j])$
7: return $A$
```

With the analysis above, we can effectively parallelize the serial code using Algorithm 2. According to Algorithm 2, the computational complexity is calculated as $3\times p\times \frac{N}{p}\times\frac{N}{p}\times N = O(\frac{N^3}{p})$, and the communication complexity is $2\times p\times 2\times\frac{N}{p}\times N = O(N^2)$.

After implementing Algorithm 2 using *MPI* on a server equipped with 64 CPU cores and 256 GB of memory, we observed the performance outcomes as depicted in Figure 2 across various values of N and p. Notably, results for $N = 10124$ are absent under $p = 32$ and $p = 64$ due to memory constraints; the implementation requires each process to hold the entire matrix, which leads to memory overflow when both the process count and N are large. For clearer visualization of the results, we have set the y-axis to a logarithmic scale. The data reveals that for process counts smaller than 32, all values of N demonstrate significant speedup. However, for a process count of 64, there is no speedup observed for $N = 256$ and $N = 512$. Thus, to achieve speedup with higher numbers of processes, a larger N is required.

2 FLOYD PROGRAM

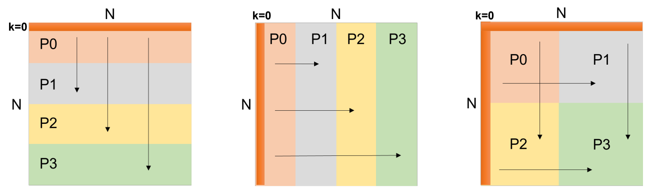
Floyd algorithm is used to solve the all-pairs shortest path problem in a graph. Algorithm 3 shows the computing process when storing graph in the matrix $A^{N\times N}$.

From Algorithm 3, it is evident that the matrix undergoes N update iterations, with each element's update being a basic operation. During the k-th iteration, each element $A[i][j]$ requires access to $A[i][k]$ and $A[k][j]$. As a result, all elements in the k-th column and the k-th row are needed for other computations during this iteration. To efficiently implement parallelism, it is crucial to ensure that the updates to $A[i][j]$ during the k-th iteration do not depend on the concurrent updates of $A[i][k]$ and $A[k][j]$. Fortunately, during the k-th iteration, the values in the k-th column and row remain unchanged, as the update operations do not alter their values. The following equations clarify the unchanged property.

$$A[i][k] = \min(A[i][k], A[i][k] + A[k][k]) = A[i][k]$$
$$A[k][j] = \min(A[k][j], A[k][k] + A[k][j]) = A[k][j]$$

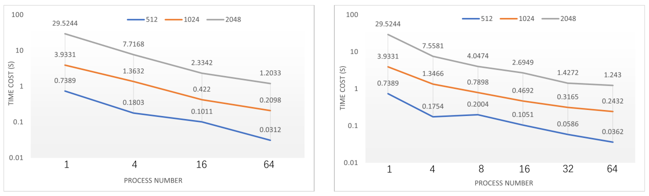
Therefore, during the k-th iteration, we can directly transmit the values of the k-th column and the k-th row to other tasks without needing to update them first. Since matrix A is two-dimensional, it can be partitioned in three different ways: row-wise partitioning, column-wise partitioning, and block-wise partitioning. These partitioning schemes are illustrated in Figure 3.

For row-wise partitioning, the broadcast among primitive tasks within the same row is unnecessary because all data values are local to the task. Thus, in each iteration k, the process owning the



(a) row-wise (b) column-wise (c) block-wise

Figure 3: Different partitioning schemes



(a) block-wise partitioning (b) row-wise partitioning

Figure 4: The Cost Time(s) of Algorithm 4 and Algorithm 5 under different number of processes and different N.

k-th row must send this row to all other processes. As depicted in Figure 3-(a), when $k = 0$, process 0 sends the first row to other processes. The communication complexity per iteration is $\log p(\lambda + \frac{N}{\beta})$, and the computation complexity is $\frac{N}{p}N\chi$. The total complexity is $\frac{N^3}{p}\chi + N\log p(\lambda + \frac{N}{\beta})$.

For column-wise partitioning, broadcasts within the same column are eliminated. Similarly, the process owning the k-th column must distribute this column to all other processes during each iteration k. As shown in Figure 3-(b), for $k = 0$, process 0 sends the first column. The communication and computation complexities match those of row-wise partitioning.

Block-wise partitioning requires data exchanges among processes in the same row or column. Each iteration k necessitates that $\sqrt{p}$ processes owning the k-th row send data across their column, and $\sqrt{p}$ processes owning the k-th column do the same across their row. For instance, in Figure 3-(c) when $k = 0$, processes 0 and 1 send the first row to processes 2 and 3, while processes 0 and 2 send the first column to processes 1 and 3. The communication complexity for each iteration is $\log\sqrt{p}(\lambda + \frac{N}{\sqrt{p}\beta}) + \log\sqrt{p}(\lambda + \frac{N}{\sqrt{p}\beta})$. The computation complexity is $\frac{N}{\sqrt{p}}\frac{N}{\sqrt{p}}\chi$. The overall complexity becomes $\frac{N^3}{p}\chi + N\log p(\lambda + \frac{N}{\sqrt{p}\beta})$.

Analysis reveals that block-wise partitioning has a lower communication complexity compared to the other schemes. Despite this, row-wise partitioning, which shares the same communication complexity as column-wise partitioning, tends to perform better due to the matrix's row-major storage format. Therefore, we implement row-wise partitioning in Algorithm 4 and block-wise partitioning in Algorithm 5.

After implementing Algorithm 4 and Algorithm 5 using *MPI* on a server with 64 CPU cores and 256 GB of memory, we analyzed the performance outcomes illustrated in Figure 4 for various values of N and p. Since p must be a square number for block-wise partitioning, values like 8 and 32 are omitted. The results show that

Algorithm 4 Row-wise Paralleling Floyd Program

```
1:  Input: graph matrix $A^{N\times N}$
2:  Output: Updated $A$
3:  $r \leftarrow MPI\_Comm\_rank()$
4:  $p \leftarrow MPI\_Comm\_size()$
5:  $s \leftarrow \frac{N}{p}$
6:  compute the start row index $sr \leftarrow r \times s$, the end row index $er \leftarrow (r+1) \times s$
7:  set temp array $T[N]$ to store required row for each iteration
8:  for $k$ from 0 to $N-1$
9:      compute the owner of the $k$-th row, $rt \leftarrow \frac{k}{s}$
10:     if $r == rt$
11:         store $k$-th row to array $T$
12:     use $MPI\_Bcast()$ to broadcast $T$ among all processes
13:     for $i$ from $sr$ to $er-1$
14:         for $j$ from 0 to $N-1$
15:             $A[i][j] = \min(A[i][j], A[i][k] + T[j])$
16: return $A$
```

Algorithm 5 Block-wise Paralleling Floyd Program

```
1:  Input: graph matrix $A^{N\times N}$
2:  Output: Updated $A$
3:  $r \leftarrow MPI\_Comm\_rank()$
4:  $p \leftarrow MPI\_Comm\_size()$
5:  compute the square root of p, $sp \leftarrow \sqrt{p}$
6:  compute the side-length of the block, $s \leftarrow \frac{N}{sp}$
7:  compute the row index in the partitioning, $px = \frac{r}{sp}$
8:  compute the column index in the partitioning, $py = r\%sp$
9:  compute the start row index of the block data, $sr \leftarrow px \times s$; the end row index $er \leftarrow (px+1) \times s$; the start column index $sc \leftarrow py \times s$; the end column index $ec \leftarrow (py+1) \times s$
10: set temp array $TR[s]$ to store required row for each iteration
11: set temp array $TC[s]$ to store required column for each iteration

12: use $MPI\_Comm\_split()$ to split the whole communicator into rows and columns such that processes with same $px$ are in the same row communicator $ROW\_COMM$ and processes with same $py$ are in the same column communicator $COLUMN\_COMM$
13: for $k$ from 0 to $N-1$
14:     compute the owners of the $k$-th row, $rr \leftarrow \frac{k}{s}$
15:     compute the owners of the $k$-th column, $rc \leftarrow \frac{k}{s}$
16:     if $px == rr$
17:         store $k$-th row in data block to array $TR$
18:     if $py == rc$
19:         store $k$-th column in data block to array $TC$
20:     use $MPI\_Bcast()$ to broadcast $TR$ among all processes with communicator $COLUMN\_COMM$
21:     use $MPI\_Bcast()$ to broadcast $TC$ among all processes with communicator $ROW\_COMM$
22:     for $i$ from $sr$ to $er-1$
23:         for $j$ from $sc$ to $ec-1$
24:             $A[i][j] = \min(A[i][j], TC[i-sr] + TR[j-sc])$
25: return $A$
```

the speedup is substantial when the number of processes is less than 32. However, with 64 processes, the speedup remains favorable only at $N = 2048$. Notably, block-wise partitioning outperforms row-wise partitioning with larger numbers of processes, supporting our earlier analysis that block-wise partitioning has a smaller communication complexity. Based on these findings, block-wise partitioning is recommended for scenarios involving large numbers of processes and problem sizes.