Final Project Report

RENYANG GUAN, Shanghai Jiao Tong University, China

ACM Reference Format:
Renyang Guan. 2024. Final Project Report. 1, 1 (May 2024), 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PARALLEL ADI: 3-D SPLIT

1.1 Introduction

The parallel algorithm is designed to improve the performance of a nested loop computation on a 3D array by utilizing multiple processors. The goal is to distribute the workload evenly across four processors and synchronize the computation at necessary steps to ensure correct results. This report explains the design and functioning of the parallel algorithm, highlighting the key steps and communication strategies employed.

1.2 Serial Algorithm Overview

The serial version of the algorithm processes a 3D array `A` with three nested loops, each dependent on different indices (`i`, `j`, and `k`). The computations are as follows:

1. i-dependent loop:

```
for (k = 0; k < N; k++) {
    for (j = 0; j < N; j++) {
        for (i = 1; i < N; i++) {
            A[k][j][i] = A[k][j][i] * 0.4 - A[k][j][i-1] * 0.6;
        }
    }
}
```

2. j-dependent loop:

```
for (k = 0; k < N; k++) {
    for (i = 0 ; i < N; i++) {
        for (j = 1; j < N; j++) {
            A[k][j][i] = A[k][j][i] * 0.5 - A[k][j-1][i] * 0.5;
        }
    }
}
```

Author's address: Renyang Guan, Shanghai Jiao Tong University, Shanghai, China, guanrenyang@sjtu.edu.cn.

3. k-dependent loop:

```
for (j = 0; j < N; j++) {
    for (i = 0; i < N; i++) {
        for (k = 1; k < N; k++) {
            A[k][j][i] = A[k][j][i] * 0.6 - A[k-1][j][i] * 0.4;
        }
    }
}
```

1.3 Parallel Algorithm Design

To leverage parallel processing, the 3D array `A` is divided into subarrays, and each subarray is assigned to one of the four processors. The array `A` is split in three dimensions, resulting in eight subarrays, with each processor handling two subarrays.

1.3.1 Processor Subarray Allocation.

Processor 0: Subarray 1: `A(0~N/2, N/2+1~N-1, 0~N/2)` Subarray 2: `A(N/2+1~N-1, 0~N/2, N/2+1~N-1)`
Processor 1: Subarray 1: `A(0~N/2, 0~N/2, 0~N/2)` Subarray 2: `A(N/2+1~N-1, N/2+1~N-1, N/2+1~N-1)`
Processor 2: Subarray 1: `A(0~N/2, N/2+1~N-1, N/2+1~N-1)` Subarray 2: `A(N/2+1~N-1, 0~N/2, 0~N/2)`
Processor 3: Subarray 1: `A(0~N/2, 0~N/2, N/2+1~N-1)` Subarray 2: `A(N/2+1~N-1, N/2+1~N-1, 0~N/2)`

1.3.2 Computation Steps.

Initialization: Each processor initializes its part of the array and sets the indices for the subarrays it is responsible for. The computation for each dependent loop (`i`, `j`, `k`) is performed in two phases to handle boundary conditions and ensure data consistency.

i-Dependent Loop: Each processor first computes its assigned subarrays for `i` in the range `0~N/2`. After completing this phase, a communication step ensures that boundary values are exchanged with neighboring processors. This synchronization step is crucial for correct results when computing the second phase with `i` in the range `N/2~N`.

j-Dependent Loop: Similar to the `i`-dependent loop, each processor computes its assigned subarrays for `j` in the range `0~N/2`, followed by a communication step to exchange boundary values. The second phase of the computation then proceeds with `j` in the range `N/2~N`.

k-Dependent Loop: The process is repeated for the `k`-dependent loop, with computations performed for `k` in the range `0~N/2`, followed by a communication step and then the second phase for `k` in the range `N/2~N`.

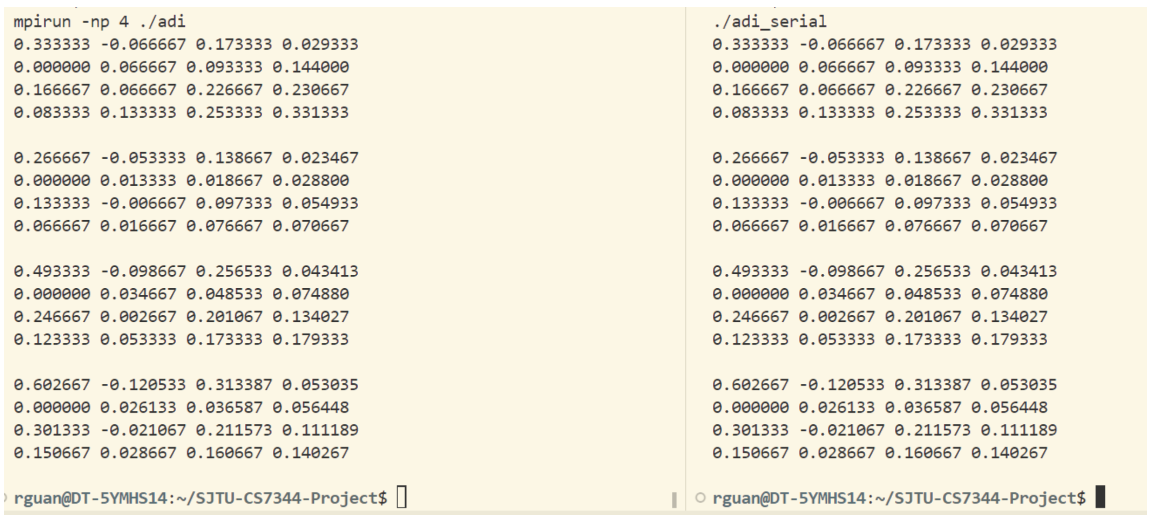
1.3.3 Communication Strategy.

Boundary Value Exchange: After computing the first phase of each loop, processors exchange boundary values with their partner processors. This ensures that the boundary conditions are met for the second phase of the computation.

Final Gathering: After completing all three loops, processor `0` gathers the results from all processors. Each processor sends its computed subarrays to processor `0`, which combines them to form the final array `A`.

1.4 Result

The performance and correctness of the parallel algorithm were evaluated by comparing the results of the serial and parallel computations. Both versions of the algorithm were executed with a 3D array of size N=4, as is shown in 1.



```
mpirun -np 4 ./adi
0.333333 -0.066667 0.173333 0.029333
0.000000 0.066667 0.093333 0.144000
0.166667 0.066667 0.226667 0.230667
0.083333 0.133333 0.253333 0.331333

0.266667 -0.053333 0.138667 0.023467
0.000000 0.013333 0.018667 0.028800
0.133333 -0.006667 0.097333 0.054933
0.066667 0.016667 0.076667 0.070667

0.493333 -0.098667 0.256533 0.043413
0.000000 0.034667 0.048533 0.074880
0.246667 0.002667 0.201067 0.134027
0.123333 0.053333 0.173333 0.179333

0.602667 -0.120533 0.313387 0.053035
0.000000 0.026133 0.036587 0.056448
0.301333 -0.021067 0.211573 0.111189
0.150667 0.028667 0.160667 0.140267

rguan@DT-5YMHS14:~/SJTU-CS7344-Project$
```

```
./adi_serial
0.333333 -0.066667 0.173333 0.029333
0.000000 0.066667 0.093333 0.144000
0.166667 0.066667 0.226667 0.230667
0.083333 0.133333 0.253333 0.331333

0.266667 -0.053333 0.138667 0.023467
0.000000 0.013333 0.018667 0.028800
0.133333 -0.006667 0.097333 0.054933
0.066667 0.016667 0.076667 0.070667

0.493333 -0.098667 0.256533 0.043413
0.000000 0.034667 0.048533 0.074880
0.246667 0.002667 0.201067 0.134027
0.123333 0.053333 0.173333 0.179333

0.602667 -0.120533 0.313387 0.053035
0.000000 0.026133 0.036587 0.056448
0.301333 -0.021067 0.211573 0.111189
0.150667 0.028667 0.160667 0.140267

rguan@DT-5YMHS14:~/SJTU-CS7344-Project$
```

Fig. 1. adi result

2 FLOYD

2.1 Introduction

The Floyd-Warshall algorithm is traditionally used for computing the shortest paths between all pairs of nodes in a weighted graph. Given its cubic runtime complexity, the algorithm becomes computationally intensive with the increase of graph size. Parallelizing the Floyd-Warshall algorithm can significantly reduce computation time by distributing tasks across multiple processors.

2.2 Methodology

2.2.1 System Configuration.

- **Hardware:** Multi-core processors with MPI-compatible networking capabilities.
- **Software:** MPI library and C compiler supporting MPI bindings.

2.2.2 Algorithm Design. The implementation involves initializing MPI, distributing the graph's adjacency matrix among MPI processes, and performing parallel computations to update the shortest path matrix.

Data Distribution. The adjacency matrix is represented as a one-dimensional array to ensure contiguous memory allocation. This matrix is divided row-wise among the processes using `MPI_Scatter`.

Parallel Computation. Each process is responsible for updating its segment of the matrix based on the current intermediate vertex using the relation:

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

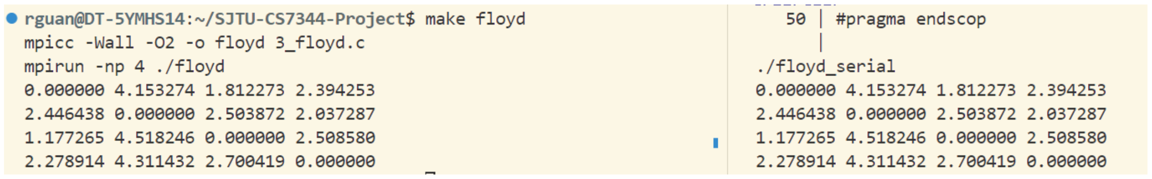
The row corresponding to the intermediate vertex ‘k‘ is broadcast to all processes using `MPI_Bcast`.

Synchronization. Processes synchronize after each update to ensure consistency of the matrix across all nodes.

2.2.3 Code Implementation. The core of the implementation handles the initialization, data distribution, computation, and final gathering of data at the root process. Memory management is carefully handled to prevent leaks.

2.3 Result

The correctness of the parallel algorithm were evaluated by comparing the results of the serial and parallel computations. Both versions of the algorithm were executed with a 3D array of size N=4, as is shown in 2.



```
rguan@DT-5YMHS14:~/SJTU-CS7344-Project$ make floyd
mpicc -Wall -O2 -o floyd 3_floyd.c
mpirun -np 4 ./floyd
0.000000 4.153274 1.812273 2.394253
2.446438 0.000000 2.503872 2.037287
1.177265 4.518246 0.000000 2.508580
2.278914 4.311432 2.700419 0.000000
```

```
   50 | #pragma endscop
      |
./floyd_serial
0.000000 4.153274 1.812273 2.394253
2.446438 0.000000 2.503872 2.037287
1.177265 4.518246 0.000000 2.508580
2.278914 4.311432 2.700419 0.000000
```

Fig. 2. floyd result