Parallel Matrix Multiplication Algorithm Comparison

Adam Ross
adro4510@colorado.edu

Notice: I have run out of time and there is literally nothing I can do about it besides prove the correctness of my serial and fox algorithms.

I have wasted too much time on the all gather, and it is still not running. I have the sbatch jobs ready to go for serial and fox but not time. I will accept the failing grade.

Abstract

This paper is written to examine parallel and serial implementations of matrix multiplication algorithms, the timings and speed up associated with each algorithm or computation method, communicator organization in MPI, and the general purpose of derived data types. I will first implement and prove the implementation for matrix multiplication via serial straightforward row-column dot products, then a parallel dot product method using allgather, and finally Fox's parallel matrix multiplication algorithm. These will then be subjected to timing of ranging data sizes and then analysis.

1. Introduction

Often in parallel computing your data set will follow particular patterns and when decomposing your data between processes it is useful to define MPI communicators and possibly custom or derived data types to make process communication as efficient as possible. To examine and reflect upon the communicator and data type implementations we compare serial and parallel matrix multiplication algorithms. We will do this by implementing a serial SMMA(square matrix multiplication algorithm), a parallel allgather SMMA, and Fox's SMMA. We will then run each program one either comet or stampede, consistently, with the following NxN matrix sizes; 144, 576, 1152 and 2304 with varying processor counts; 4, 16, 36, 64, and 144. This data will then be analyzed to point toward the usefulness of communicators and derived data types.

2. Overview

The serial algorithm was implemented following basic matrix multiplication and timed by the MPI library at np = 1. Figure 2.1 explains the multiplication implementation with basic code and Figure 2.2 is the output from the serial program running on a 9x9 matrix.

```
          while(cont) { // continue running while our confidence interval error is
too large
              start = MPI_Wtime(); // start timing
              for (i = 0; i < N; i++) { // Multiply Matrices
                 for (j = 0; j < N; j++) {
                    sum = 0.0;
                    for (k = 0; k < N; k++) {
                       sum = sum + A[i][k] * B[k][j];
                    }
```

```
                    C[i][j] = sum;
                }
            }

            finish = MPI_Wtime(); // finish timing
            timing_data[n] = (finish - start); // collect timing data to calculate
confidence intervals

            cont = Calc_Confidence_Interval_stop(timing_data, n); // calculate
intervals and determine if we should continue

            n++;
        }
```

Figure 2.1 - Serial matrix implementation with timing and confidence intervals

The following can be worked out by hand or reputable calculator to get the same values. I used Wolfram Alpha to confirm.

```
mpiexec -np 1 RossAdam_serial_HW6
Start. Matrix A
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80

Start. Matrix B
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80

0.00000227999999999999     0.0000004490  0.0000001760  7.719630     25
Result Matrix C
1836 1872 1908 1944 1980 2016 2052 2088 2124
4752 4869 4986 5103 5220 5337 5454 5571 5688
7668 7866 8064 8262 8460 8658 8856 9054 9252
10584 10863 11142 11421 11700 11979 12258 12537 12816
13500 13860 14220 14580 14940 15300 15660 16020 16380
16416 16857 17298 17739 18180 18621 19062 19503 19944
```

```
19332 19854 20376 20898 21420 21942 22464 22986 23508
22248 22851 23454 24057 24660 25263 25866 26469 27072
25164 25848 26532 27216 27900 28584 29268 29952 30636
```

Figure 2.2 - Serial matrix output from a 9x9 matrix where the input matrices increment each new entry by 1

Below in figure 2.3 is the main lifting work from my Fox implementation in which we do the necessary shifts, broadcasts, and process communication using row and column communicators defined in Figure 2.4. Finally 2.5 is sample output from running the program with 9 processes on two 9x9 matrix where each consecutive entry is incremented by 1.

```
MPI_Barrier(MPI_COMM_WORLD);
start = MPI_Wtime();

for (i = 0; i < grid->grid_order; i++) {
    root = (grid->current_row + i) % grid->grid_order;
    if (root == grid->current_col) {
        MPI_Bcast(*local_A, local_n * local_n,
                  MPI_DOUBLE, root, grid->row_comm);
        Local_Matrix_Product(local_A, local_B, local_C, local_n);
    } else {
        MPI_Bcast(*temp_matrix, local_n * local_n,
                  MPI_DOUBLE, root, grid->row_comm);
        Local_Matrix_Product(temp_matrix, local_B, local_C, local_n);
     }
     // The circular shift and replacement of local_B values
     MPI_Sendrecv_replace(*local_B, local_n * local_n,
                          MPI_DOUBLE, dest, 0, source, 0,
                          grid->col_comm, &status);
 }

 finish = MPI_Wtime();
```

Figure 2.3 - Fox implementation snippet with timing

```
grid->grid_order = (int) sqrt((double)p);

local_ns[0] = local_ns[1] = grid->grid_order;
periods[0] = periods[1] = 1;

// create communicator for the process grid
MPI_Cart_create(MPI_COMM_WORLD, 2, local_ns, periods, 1,
&(grid->grid_comm));

// retrieve the process rank in the grid Communicator
// and the process coordinates in the cartesian topology
MPI_Comm_rank(grid->grid_comm, &(grid->grid_comm_rank));
MPI_Cart_coords(grid->grid_comm, grid->grid_comm_rank, 2, grid_coords);
```

```
    grid->current_row = grid_coords[0];
    grid->current_col = grid_coords[1];

    // setup row communicators
    sub_coords[0] = 0;
    sub_coords[1] = 1;
    MPI_Cart_sub (grid->grid_comm, sub_coords, &(grid->row_comm));

    // setup column communicators
    sub_coords[0] = 1;
    sub_coords[1] = 0;
    MPI_Cart_sub(grid->grid_comm, sub_coords, &(grid->col_comm));
```

Figure 2.4 - Fox cartesian communicator implementation

```
Start. Matrix A
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80

Start. Matrix B
0 1 2 3 4 5 6 7 8
9 10 11 12 13 14 15 16 17
18 19 20 21 22 23 24 25 26
27 28 29 30 31 32 33 34 35
36 37 38 39 40 41 42 43 44
45 46 47 48 49 50 51 52 53
54 55 56 57 58 59 60 61 62
63 64 65 66 67 68 69 70 71
72 73 74 75 76 77 78 79 80

Local A - rank 0
0 1 2
9 10 11
18 19 20

Local B - rank 0
0 1 2
9 10 11
18 19 20

0.000073959999999958     0.0000614638  0.0000240938  32.576810     25
```

```
Result Matrix C
1836 1872 1908 1944 1980 2016 2052 2088 2124
4752 4869 4986 5103 5220 5337 5454 5571 5688
7668 7866 8064 8262 8460 8658 8856 9054 9252
10584 10863 11142 11421 11700 11979 12258 12537 12816
13500 13860 14220 14580 14940 15300 15660 16020 16380
16416 16857 17298 17739 18180 18621 19062 19503 19944
19332 19854 20376 20898 21420 21942 22464 22986 23508
22248 22851 23454 24057 24660 25263 25866 26469 27072
25164 25848 26532 27216 27900 28584 29268 29952 30636
```

Figure 2.5 - Fox's matrix output from a 9x9 matrix where the input matrices increment each new entry by 1

3. Algorithm Verification

This section would have been awesome if my code had been completed. Oh man. You don't even know.

4. Conclusion

Find more time to work on homework outside of my 50hr/week job while maintaining relationships.