# CM30225 Parallel Computing Coursework 2

Lei Zeng

01 January 2015

# Contents

**1 How to use** **1**

**2 Code Description** **2**

2.1 Basic idea . . . . . 2

2.2 Job Splitting . . . . . 2

2.3 End Calculation . . . . . 2

2.4 Pseudo code . . . . . 3

**3 Correctness Testing** **4**

3.1 Testing basic calculation . . . . . 4

3.2 Further Testing . . . . . 5

3.3 Parallelism Testing . . . . . 5

3.3.1 Testing job splitting . . . . . 5

3.3.2 Testing Averaging . . . . . 6

**4 Scalability Investigation** **9**

4.1 Time spent . . . . . 9

4.2 Speedup . . . . . 10

4.3 Efficiency . . . . . 10

4.4 Compare with shared memory . . . . . 11

# Chapter 1

# How to use

Compile cw2.c:

```
mpicc -std=c99 -Wall cw2.c -o cw2
```

This program takes 2 inputs:

1. dimension: Type: integer, the size of square array.
2. precision: Type: double, the accuracy you require from the final result.

e.g.:

```
./cw2 10 0.1
```

This will randomly generate a 10 x 10 square array and keep iterating until all new values differ from their old values by less than the precision 0.1.

This program will take use of all cores in each node. Specify numbers of cores and nodes you want to use in the job script.

# Chapter 2

# Code Description

## 2.1 Basic idea

All processors run the same code except *Root* processor. *Root* ($rank_0$, in my program) broadcasts the square array to the rest processors. Then all processors average their part of rows with regard to rank number. After average, the rest processors send the results of their rows back to *Root*. *Root* receives all new values from other processors and replaces the square array with new values.

## 2.2 Job Splitting

In an $n*n$ square array with $i$ processors, we have: $avg = \frac{n-2}{i}$ and $extra = (n-2) \mod i$.

All processors with $rank < extra$ will averaging $avg + 1$ rows, and the rest will averaging $avg$ rows.

E.g. a $15 * 15$ square array with 5 processors:

$avg = \frac{15-2}{5} = 2$ and $extra = (15-2) \mod 5 = 3$, so from $rank_0$ to $rank_4$, they will average $3, 3, 3, 2, 2$ rows.

## 2.3 End Calculation

The is a flag *isEnd*. Before replacing old values with new one, set *isEnd* to *true*. During replacing, if $|old - new| > precision$ occurs, *isEnd* will be set to $false$. At the end of replacing, *Root* broadcasts *isEnd*. Ig *isEnd* is not *true*, the averaging step will continue.

## 2.4 Pseudo code

```
main()
{
    init MPI;

    calculate rows to average with regard to rank;

    if (rank == ROOT) init array;

    while(!isEnd)
    {
        MPI_Bcast array;

        averaging;

        if (rank == ROOT)
        {
            MPI_Recv;
            repacing array;
        }
        else
        {
            MPI_Send;
        }

        MPI_Barrier;

        MPI_Bcast isEnd;
    }
}
```

# Chapter 3

# Correctness Testing

## 3.1 Testing basic calculation

Test program in single processor to check the results of each calculation with regard to specification. The following sample using 4 x 4 square array with *precision* = 1.0, which is the same as the one I used in *coursework 1*, so it is easier to check results:

| | | | |
|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 |
| 91.164736 | 19.755137 | 33.522276 | 76.822959 |
| 27.777471 | 55.396996 | 47.739705 | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 |

Each calculation will average 4 neighbours (the old values of top-right-bottom-left cells):

| | | | |
|---|---|---|---|
| 84.018772 | **39.438293** | 78.309922 | 79.844003 |
| **91.164736** | **54.880575** | **33.522276** | 76.822959 |
| 27.777471 | **55.396996** | 47.739705 | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 |

Each step will update all cells except boundaries. The result of first update:

| | | | |
|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 |
| 91.164736 | **54.880575** | **55.656931** | 76.822959 |
| 27.777471 | **36.653101** | **61.757334** | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 |

The difference between new value and old value of cell[2][2] is 54.880575 − 19.755137 = 35.125438 which is larger than precision (1.0), so the averaging step will be continued. In this example, it takes 6 steps to end: the new values of all cells differ from their previous values by less than the precision:

| | | | |
|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 |
| 91.164736 | **63.671773** | **72.311806** | 76.822959 |
| 27.777471 | **53.307976** | **70.548532** | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 |

## 3.2 Further Testing

Due to the large outputs, it is very difficult to test when the dimension increases. I try to compare the result of each step with my friend who uses different method to accomplish this coursework. I randomly generate a square array with 100 x 100 dimension, pass the file to my friend. We both write the results of each step to an output file. Here is how I compare the output file:

**Using git to compare results:** Firstly I push my result file to a git repository, and then push my friend's output file as update version of my result. Git will help us to check those two files and highlight the differences.

It turns out that two results are exactly the same, so I will say that both of our programs work as what we expected.

## 3.3 Parallelism Testing

Here I use another sample to test multi-threading. In my program, the algorithm for splitting work is described as following:

In an n x n square array with i threads, each thread (except the last one) deals with $\frac{n-2}{i}$ row(s), and the last thread will deal with the rest.

Each thread use the same algorithm which I have tested in 3.1 and 3.2, so I will focus on testing the part of splitting job and parallelism.

### 3.3.1 Testing job splitting

The out put of testing different dimensions and different number of processors:

1. 20 x 20 with 4 threads:

```
rank 0 in node006: 1 - 5
rank 3 in node006: 15 - 18
rank 1 in node006: 6 - 10
rank 2 in node006: 11 - 14
```

2. 50 x 50 with 8 threads:

```
rank 3 in node006: 19 - 24
rank 1 in node006: 7 - 12
rank 0 in node006: 1 - 6
rank 6 in node006: 37 - 42
rank 5 in node006: 31 - 36
rank 2 in node006: 13 - 18
rank 4 in node006: 25 - 30
rank 7 in node006: 43 - 48
```

3. 100 x 100 with 16 threads:

```
rank 3 in node007: 21 - 26
rank 7 in node007: 45 - 50
rank 0 in node007: 1 - 7
rank 1 in node007: 8 - 14
rank 6 in node007: 39 - 44
rank 5 in node007: 33 - 38
rank 14 in node008: 87 - 92
rank 2 in node007: 15 - 20
rank 4 in node007: 27 - 32
rank 9 in node008: 57 - 62
rank 8 in node008: 51 - 56
rank 10 in node008: 63 - 68
rank 12 in node008: 75 - 80
rank 15 in node008: 93 - 98
rank 11 in node008: 69 - 74
rank 13 in node008: 81 - 86
```

All test cases give the result as expected. The splitting part should work fine.

### 3.3.2 Testing Averaging

Firstly, I will test a small array: 6 x 6 with 3 processors, so $rank_0$ computes *row 1-2*, $rank_1$ computes *row 3* and $rank_2$ compute *row 4*.

| | | | | | |
|---|---|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 | 91.164736 | 19.755137 |
| 33.522276 | 76.822959 | 27.777471 | 55.396996 | 47.739705 | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 | 63.571173 | 71.729693 |
| 14.160256 | 60.696888 | 1.630057 | 24.288677 | 13.723158 | 80.417675 |
| 15.667909 | 40.094439 | 12.979045 | 10.880880 | 99.892452 | 21.825691 |
| 51.293239 | 83.911223 | 61.263983 | 29.603162 | 63.755227 | 52.428719 |

**The outputs:**

```
rank 2 in node051: 4 - 4
rank 0 in node051: 1 - 2
rank 1 in node051: 3 - 3
Round 1 starts:

rank 2 finished averaging:
```

| | | | | | |
|---|---|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 | 91.164736 | 19.755137 |
| 33.522276 | 76.822959 | 27.777471 | 55.396996 | 47.739705 | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 | 63.571173 | 71.729693 |
| 14.160256 | 60.696888 | 1.630057 | 24.288677 | 13.723158 | 80.417675 |
| 15.667909 | **43.313766** | **28.467340** | **41.690834** | **27.546239** | 21.825691 |
| 51.293239 | 83.911223 | 61.263983 | 29.603162 | 63.755227 | 52.428719 |

```
rank 0 finished averaging:
```

| | | | | | |
|---|---|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 | 91.164736 | 19.755137 |
| 33.522276 | **38.019533** | **76.438212** | **61.745172** | **68.254999** | 62.887092 |
| 36.478447 | **67.305317** | **43.091782** | **59.619954** | **56.203016** | 71.729693 |
| 14.160256 | 60.696888 | 1.630057 | 24.288677 | 13.723158 | 80.417675 |
| 15.667909 | 40.094439 | 12.979045 | 10.880880 | 99.892452 | 21.825691 |
| 51.293239 | 83.911223 | 61.263983 | 29.603162 | 63.755227 | 52.428719 |

```
rank 0 finished averaging:
```

| | | | | | |
|---|---|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 | 91.164736 | 19.755137 |
| 33.522276 | 76.822959 | 27.777471 | 55.396996 | 47.739705 | 62.887092 |
| 36.478447 | 51.340091 | 95.222973 | 91.619507 | 63.571173 | 71.729693 |
| 14.160256 | **26.806211** | **48.296895** | **29.463400** | **67.042494** | 80.417675 |
| 15.667909 | 40.094439 | 12.979045 | 10.880880 | 99.892452 | 21.825691 |
| 51.293239 | 83.911223 | 61.263983 | 29.603162 | 63.755227 | 52.428719 |

```
Thread 2 is waiting for signal
Thread 1 is waiting for signal
Thread 0 is broadcasting signals
Round 2 starts now:
...
Total rounds: 18
Result:
```

| | | | | | |
|---|---|---|---|---|---|
| 84.018772 | 39.438293 | 78.309922 | 79.844003 | 91.164736 | 19.755137 |
| 33.522276 | 44.443627 | 60.947406 | 68.739927 | 72.209024 | 62.887092 |
| 36.478447 | 42.971455 | 53.945278 | 60.512557 | 67.063370 | 71.729693 |
| 14.160256 | 38.668366 | 49.007878 | 54.970752 | 62.356437 | 80.417675 |
| 15.667909 | 47.086171 | 51.117751 | 45.666212 | 48.624188 | 21.825691 |
| 51.293239 | 83.911223 | 61.263983 | 29.603162 | 63.755227 | 52.428719 |

Check final result with non-parallel version, they are the same.

It is very difficult to test more threads or larger dimensions. The output are huge. What I did is random the input array once, output to a file. Then run non-parallel version and this program on the same array, compare the result of each round using the same method which is described in 3.2. The testing results show that my output are all correct.

# Chapter 4

# Scalability Investigation

I use three square arrays with different dimensions to do the scalability investigation:

1. 100 x 100 array with precision 0.01
2. 200 x 200 array with precision 0.1
3. 200 x 200 array with precision 0.1

## 4.1 Time spent

Figure 4.1 shows time spent with different threads. In general, using distribute memory increase the time needed drastically.

From Table 4.1, we could see that using processors in different nodes does not effect time spent on program very much. This may be because it uses distribute memory, so even processors in same node need to store messages to their own local memory. Therefore, both processors in same node and in different nodes doing the same thing.

| | 100x100 - 0.01 | 100x100 - 0.1 | 200x200 - 0.1 |
|---|---|---|---|
| 1 node, 6 processors each | 30s | 18s | 3m 11s |
| 2 node, 3 processors each | 37s | 17s | 3m 6s |
| 3 node, 2 processors each | 38s | 21s | 3m 6s |

Table 4.1: Time spent for 6 processors

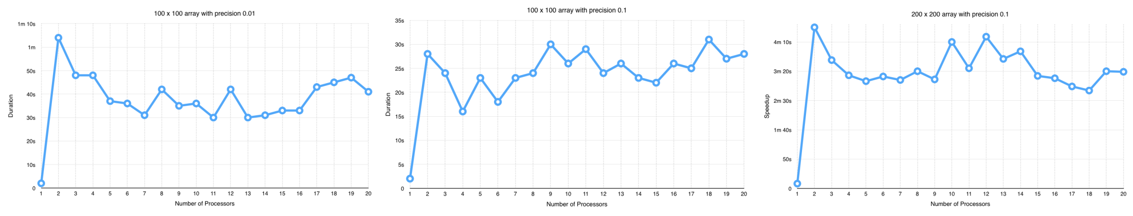

Figure 4.1: Time spent

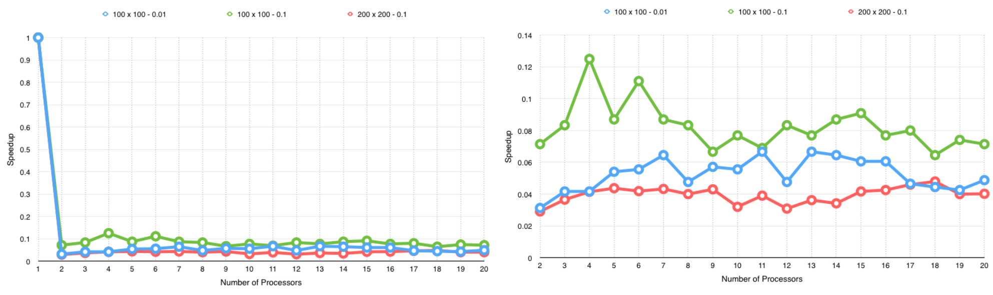

Figure 4.2: Speedup

## 4.2 Speedup

Figure 4.2 shows the parallel algorithm in comparison with the corresponding sequential algorithm. The maximum speedup is 0.125 only. The reason is that the problem to solve is too small compared with the overhead of creating a message, sending, waiting, reading the reply. In my program, broadcasting array and flag ('isEnd'), sending and receiving calculating results are huge overhead comparing with the calculation. This kind of simple problem does not worth to use distributed memory.

## 4.3 Efficiency

Figure 4.3 shows efficiency of this program run in parallel. In general, the efficiency decreases as the number of processors increase. The highest efficiency is only 0.036. This means we are not even using 3.6% of each processor. This is a huge waste since processors spent most of their time on waiting, sending and reciving messages rather than computation.

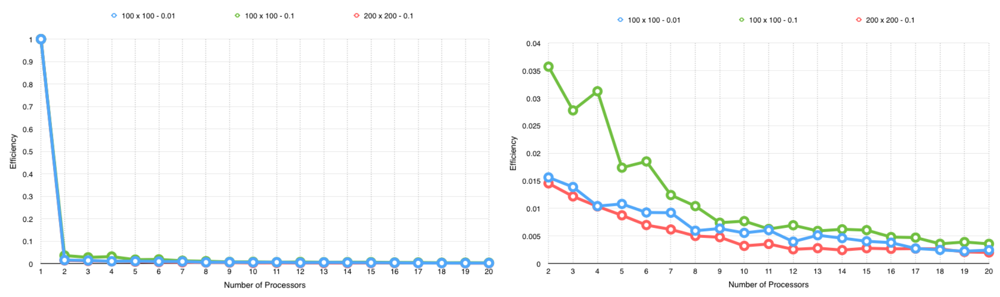

Figure 4.3: Efficiency

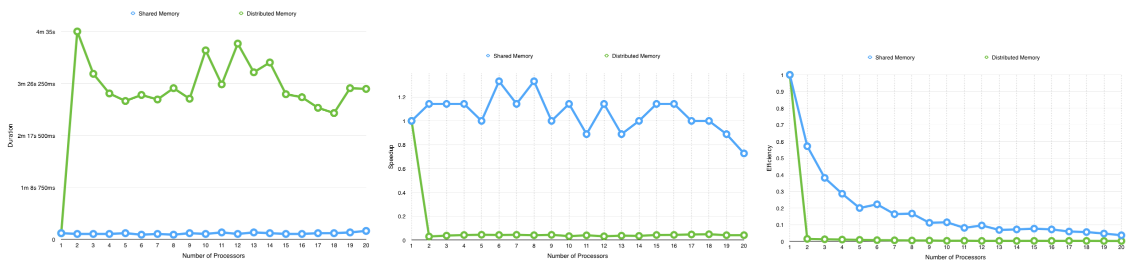

Figure 4.4: Time spent on 200x200 array with 0.1 precision

## 4.4 Compare with shared memory

From Figure 4.4 we can see that even through the overall speedup and efficiency are not good, program using shared memory are a lot better than the one using distributed memory. In shared memory, all thread could read array directly from shared memory after main thread update the values, after calculation, each thread write the new value back to shared memory, all threads do this one by one. While in distributed memory, all other processors have to wait for ROOT to send updated array, store it in their own memory. After calculation, all other processors need to send new values back to ROOT, ROOT will have to receive all those new values and update array, and then broadcast new array. Read from shared memory is much more faster then broadcast to all other processors. Even through write back to shared memory can not be done simultaneously, it is still a lot faster than sending and receiving (receiving can not be done simultaneously in this case neither). Those would be the main reason why shared memory is much faster than distributed memory to this program.