CM30225 Parallel Computing
Coursework 1

Lei Zeng

23 Novenber 2014

Contents

Chapter 1

How to use

This program takes 3 inputs:

1. dimension: Type: integer, the size of square array.
2. precision: Type: double, the accuracy you require from the final result.
3. number of thread: Type: integer, the thread you want to use

e.g.:

```
./cw1 10 0.1 5
```

This will randomly generate a 10 x 10 square array, run with 5 threads, and keep iterating until all new values differ from their old values by less than the precision 0.1.

Chapter 2

Code Description

2.1 Basic idea

There are four low-level primitives used to make a parallel solution:

1. **pthread_create**: create threads according to user input
2. **barriers**: at the end of each loop, all thread need to wait for others, until everyone is finished, then copy the new array to its temporary array, and then start next loop, which is averaging four nearest neighbours.
3. **mutexes**: to avoid race condition, lock variables like 'isEnd' which indicates the end of loop, and 'count' which counts how many steps needed to end.
4. **condition variables**: a way to make sure all threads start together. At the beginning of each loop, need to set 'isEnd' to true, because each averaging result will compare with the old value to get precision, if one of them is not valid, set 'isEnd' to false, so the loop continues. We could not let all threads to modify 'isEnd' at the beginning: other threads may read this value, and if anyone set it to true before others entering the loop, then later threads will stop looping. So the best way is to let all threads wait inside loop, and the last thread changes 'isEnd' to true, and broadcast messages, so others will start averaging again.

2.2 Pseudo code

```
main()
{
    for(numThreads)
    {
        /* create threads */
        pthread_create
    }
    pthread_join
}

function averaging
{
    while(!isEnd)
    {
        mutex_lock
        if( last_thread)
        {
            isEnd = true
            broadcast signal
        }
        else
        {
            wait for signal
        }
        mutex_unlock

        /* do averaging */
        /* set isEnd = false if any result > precision */
        barrier_wait
    }
}
```

Chapter 3

Correctness Testing

3.1 Testing basic calculation

Test program in single thread to check the results of each calculation with regard to specification. The following sample using 5 x 5 square array with *precision* = 1.0 to illustrate:

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 Testing parallelism

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 threads (thread number and row number start with 0):

1. 10 x 10 with 3 threads:

```
Thread 0 will average row 1 to 2
Thread 1 will average row 3 to 4
Thread 2 will average row 5 to 8
```

2. 20 x 20 with 3 threads:

```
Thread 0 will average row 1 to 6
Thread 1 will average row 7 to 12
Thread 2 will average row 13 to 18
```

3. 20 x 20 with 5 threads:

```
Thread 0 will average row 1 to 4
Thread 1 will average row 5 to 8
Thread 2 will average row 9 to 12
Thread 3 will average row 13 to 18
```

4. 100 x 100 with 10 threads:

```
Thread 0 will average row 1 to 9
Thread 1 will average row 10 to 18
Thread 2 will average row 19 to 27
Thread 3 will average row 28 to 36
Thread 4 will average row 37 to 45
Thread 5 will average row 46 to 54
Thread 6 will average row 55 to 63
Thread 7 will average row 64 to 72
Thread 8 will average row 73 to 81
Thread 9 will average row 82 to 98
```

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

3.3.2 Testing parallelism

Firstly, I will test a small array: 6 x 6 with 3 threads, so *Thread 0* computes *row 1*, *Thread 1* computes *row 2*, and *Thread 2* computes *row 3* & *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:

```
Thread 0 will average row 1 to 1
Thread 2 will average row 3 to 4
Thread 1 will average row 2 to 2

Thread 0 is waiting for signal
Thread 2 is waiting for signal
Thread 1 is broadcasting signals
Step 1 starts now:

Thread 1 is done:
```

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	**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

```
Thread 0 is done:
```

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	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

```
Thread 2 is done:
```

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	**43.313766**	**28.467340**	**41.690834**	**27.546239**	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
Step 2 starts now:
...
Total steps: 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 messages are huge. I use the same method as described in 3.2: compare results with non-parallel version, step by step (only compare the whole array when all threads are done in each step). This works fine and the results are all correct. I also try to run program with same dimension and same number of threads several time and compare result in more detail like what I am showing above, but the method in 3.2 is no more applicable: threads are not always running in the same sequence, hence the output results may not print in the same order.

Chapter 4

Scalability Investigation

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

1. 200 x 200 array with precision 0.001
2. 300 x 300 array with precision 0.001
3. 400 x 400 array with precision 0.001

4.1 Time spent

Figure 4.1 shows time spent with different threads. In general, using parallel could help to finish the program faster, but too many threads will make duration even longer than non-parallel. We can see that programs runs in 8 threads produce results fastest, so if we want to finish job faster and do not care about efficiency, we shall use 8 threads. Durations increase drastically from 8 threads to 9 threads.

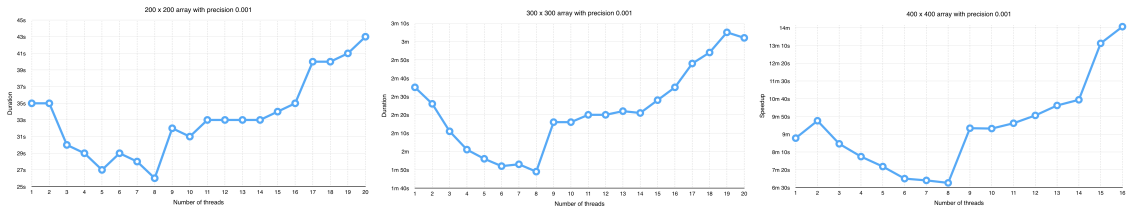

Figure 4.1: Time spent

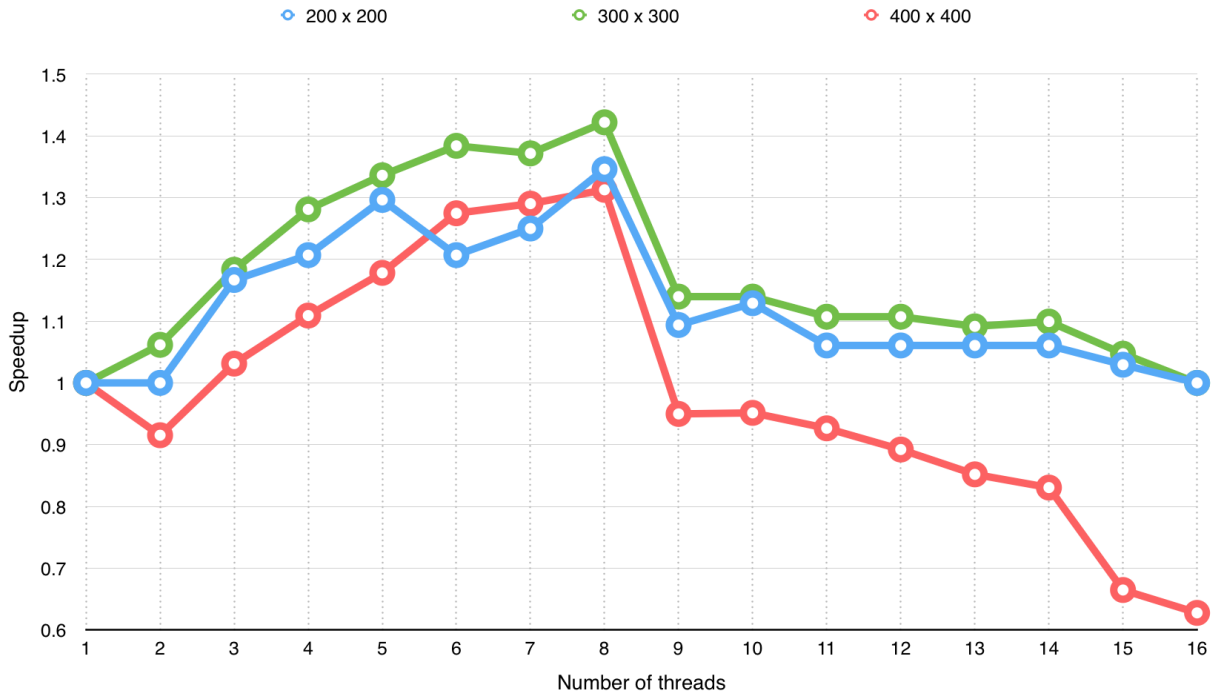

Figure 4.2: Speedup

4.2 Speedup

Figure 4.2 shows the parallel algorithm in comparison with the corresponding sequential algorithm. The maximum speedup is when using 8 threads. From Figure 4.1 we can see that 8 threads produce result fastest, and the speedup is the highest, but it is only $S_8 = 1.42$ which is much smaller than the number of thread used. The reason is that the overheads of communications between threads, such as reading 'isEnd', copy array, wait for others, etc. The cost of those communications are small for shares memory, but it is quite large compare to the computation in each thread, which is adding four elements and calculating the average. The way to increase speedup is to make the computation more complex, so the overheads will be relatively small.

4.3 Efficiency

Figure 4.3 shows efficiency of this program run in parallel. The efficiency decreases as the number of threads increase. The efficiency drops drastically from non-parallel to parallel. The highest efficiency for parallel is $E_2 = 0.53$. This means in the best case, we use only

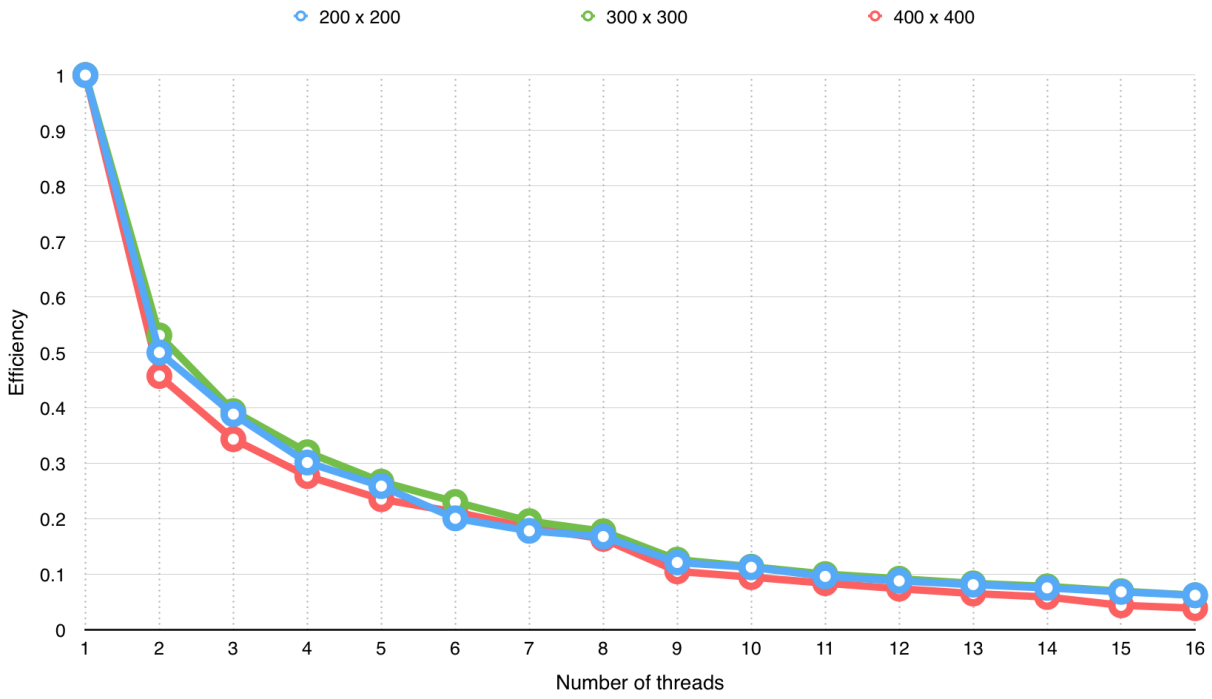

Figure 4.3: Efficiency

half of the processors; capabilities. The lowest one has $E_{16} = 0.04$, this is totally a waste. The efficiency of using 8 threads, which produce the result fasted, is $E_8 = 0.18$. So even through using 8 threads is the fastest way but the efficiency is very low, only use 8 threads when you do not care about the cost.