CM30225 Parallel Computing Assessed Coursework Assignment 1

November 8, 2021

1 Algorithm Design

My initial design for the algorithm was possibly too naiive; the plan was for all threads to work on the same given matrix, and simply lock the cells that they needed for the calculation whilst in use.

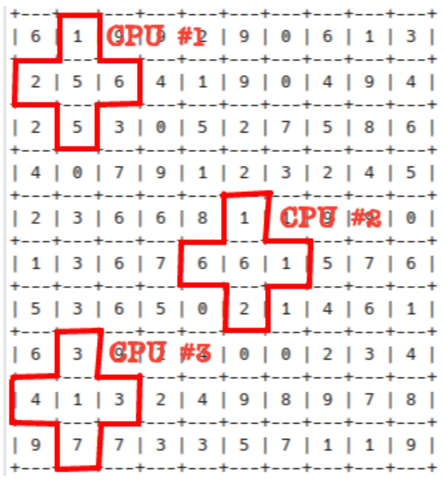


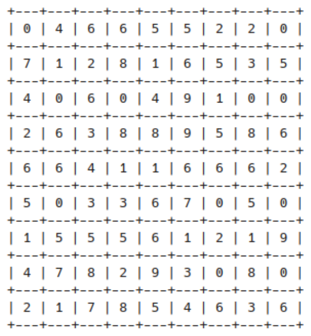
Figure 1: Naiive Appoach

As can be seen in Figure 1, a thread would lock the current cell, and the cells required to decide the new average, until the calculation was complete. The thread then releases the lock and finds a new cell. Initially, this appears like it would be an acceptable solution, however on reflection, I realized that more time would be spent on overhead - locking cells, checking the status of cells, finding new unlocked cells - than would actually be spent on the main calculation. Thus I decided to find a new algorithm with higher work efficiency, and what follows is the solution I came to.

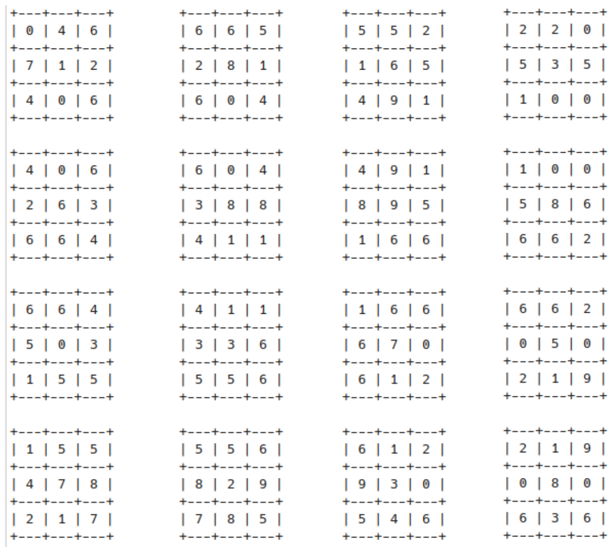
In Figure 2, the matrix is split into p sub-matrices, where p is the number of processors the program is to run on, and each sub-matrices boundary overlaps its neighbours (In Figure 2, it is assumed there are 16 processors to run on). The process runs as expected, with each thread computing the inner values of its matrix to the desired precision, and leaving the boundaries untouched. Then sub-matrices will be merged, and their boundaries computed. If the initial matrix is of an odd size (i.e. 9x9), their boundaries will be computed as a $3 * n$ array, where n is the size of the $n * n$ square array provided. The first pass will be columns, and then second pass is rows (See Figure 3). For even arrays, one set of sub-matrices will have an extra column/row, so computation will be marginally slower for even arrays.

As seen in Figure 3, The arrays passed to each processor will overlap, however since the shared values have already been computed to the desired precision and won't be changed, this shouldn't cause any issues.

This is my initial consideration of the algorithm design, pre-development. There is obviously room for improvement; for example, if the number of processors p does not evenly fit into the $(n+1)^2$ sub-matrices generated (as in Figure 2), what should be done? Clearly, the division of the matrix into an efficient number of sub-matrices can be improved, however I maintain that this principle of overlapping sub-matrices and then subsequently computing boundaries along columns/rows appears to be a good approach.

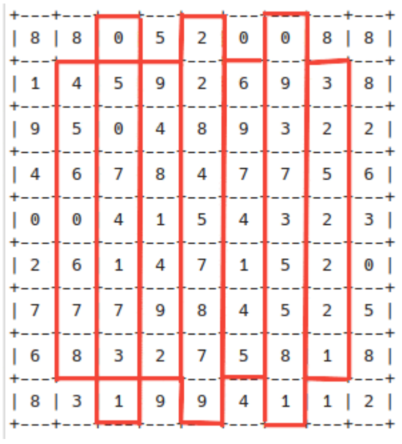


```
+---+---+---+---+---+---+---+---+---+
| 0 | 4 | 6 | 6 | 5 | 5 | 2 | 2 | 0 |
+---+---+---+---+---+---+---+---+---+
| 7 | 1 | 2 | 8 | 1 | 6 | 5 | 3 | 5 |
+---+---+---+---+---+---+---+---+---+
| 4 | 0 | 6 | 0 | 4 | 9 | 1 | 0 | 0 |
+---+---+---+---+---+---+---+---+---+
| 2 | 6 | 3 | 8 | 8 | 9 | 5 | 8 | 6 |
+---+---+---+---+---+---+---+---+---+
| 6 | 6 | 4 | 1 | 1 | 6 | 6 | 6 | 2 |
+---+---+---+---+---+---+---+---+---+
| 5 | 0 | 3 | 3 | 6 | 7 | 0 | 5 | 0 |
+---+---+---+---+---+---+---+---+---+
| 1 | 5 | 5 | 5 | 6 | 1 | 2 | 1 | 9 |
+---+---+---+---+---+---+---+---+---+
| 4 | 7 | 8 | 2 | 9 | 3 | 0 | 8 | 0 |
+---+---+---+---+---+---+---+---+---+
| 2 | 1 | 7 | 8 | 5 | 4 | 6 | 3 | 6 |
+---+---+---+---+---+---+---+---+---+
```



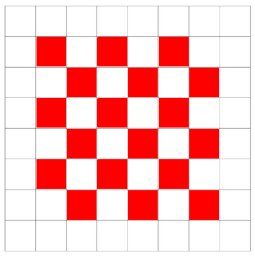
```
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 0 | 4 | 6 |    | 6 | 6 | 5 |    | 5 | 5 | 2 |    | 2 | 2 | 0 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 7 | 1 | 2 |    | 2 | 8 | 1 |    | 1 | 6 | 5 |    | 5 | 3 | 5 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 4 | 0 | 6 |    | 6 | 0 | 4 |    | 4 | 9 | 1 |    | 1 | 0 | 0 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+

+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 4 | 0 | 6 |    | 6 | 0 | 4 |    | 4 | 9 | 1 |    | 1 | 0 | 0 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 2 | 6 | 3 |    | 3 | 8 | 8 |    | 8 | 9 | 5 |    | 5 | 8 | 6 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 6 | 6 | 4 |    | 4 | 1 | 1 |    | 1 | 6 | 6 |    | 6 | 6 | 2 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+

+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 6 | 6 | 4 |    | 4 | 1 | 1 |    | 1 | 6 | 6 |    | 6 | 6 | 2 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 5 | 0 | 3 |    | 3 | 3 | 6 |    | 6 | 7 | 0 |    | 0 | 5 | 0 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 1 | 5 | 5 |    | 5 | 5 | 6 |    | 6 | 1 | 2 |    | 2 | 1 | 9 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+

+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 1 | 5 | 5 |    | 5 | 5 | 6 |    | 6 | 1 | 2 |    | 2 | 1 | 9 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 4 | 7 | 8 |    | 8 | 2 | 9 |    | 9 | 3 | 0 |    | 0 | 8 | 0 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
| 2 | 1 | 7 |    | 7 | 8 | 5 |    | 5 | 4 | 6 |    | 6 | 3 | 6 |
+---+---+---+    +---+---+---+    +---+---+---+    +---+---+---+
```

Figure 2: Scalable Approach

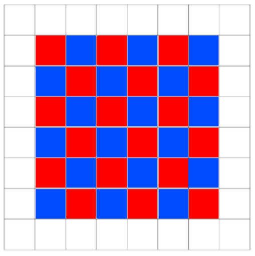


```
+---+---+---+---+---+---+---+---+---+
| 8 | 8 | 0 | 5 | 2 | 0 | 0 | 8 | 8 |
+---+---+---+---+---+---+---+---+---+
| 1 | 4 | 5 | 9 | 2 | 6 | 9 | 3 | 8 |
+---+---+---+---+---+---+---+---+---+
| 9 | 5 | 0 | 4 | 8 | 9 | 3 | 2 | 2 |
+---+---+---+---+---+---+---+---+---+
| 4 | 6 | 7 | 8 | 4 | 7 | 7 | 5 | 6 |
+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 4 | 1 | 5 | 4 | 3 | 2 | 3 |
+---+---+---+---+---+---+---+---+---+
| 2 | 6 | 1 | 4 | 7 | 1 | 5 | 2 | 0 |
+---+---+---+---+---+---+---+---+---+
| 7 | 7 | 7 | 9 | 8 | 4 | 5 | 2 | 5 |
+---+---+---+---+---+---+---+---+---+
| 6 | 8 | 3 | 2 | 7 | 5 | 8 | 1 | 8 |
+---+---+---+---+---+---+---+---+---+
| 8 | 3 | 1 | 9 | 9 | 4 | 1 | 1 | 2 |
+---+---+---+---+---+---+---+---+---+
```

Figure 3: Boundary computation: First columns, then rows



(a) First Pass



(b) Second Pass

Figure 4: Matrix Passes of the Algorithm

1.0.1 Final Algorithm

After much painstaking deliberation, what I believe to be the optimal algorithm has been found. It is a halfway compromise between the initial random-find-and-lock algorithm and the overly complicated divide-and-conquer algorithm.

As can be seen in figure 4, If the diagonals of the matrix are taken, and the red cells have their values calculated, there is no interference between threads. In the second pass, the opposite diagonals are calculated, and the computation is complete. The intuitive way to perform this calculation is as follows:

1. The diagonal cells in Figure 4a are identified; this is Pass #1
2. The diagonal cells in Figure 4b are identified. this is Pass #2
3. The number of items in Pass #1 is divided by the number of threads for the process to run on. This gives us the number of cells each thread must process, n.
4. The same is done for Pass #2, m.
5. Each thread is randomly assigned n exclusive items from Pass #1, and m items for Pass #2.
6. The threads perform their calculations.
7. Threads will check if there is any change in the auxillary cells (i.e. the cells required to calculate the average) of the Pass #2 cells. If there is not, they set a flag.
8. Once all threads have completed Pass #1, they are free to move on to Pass #2.
9. The computation for Pass #2 is performed.
10. The threads check again, but this time for Pass #1 cells. If the flag was set from step 7, and this check is also successful, the calculation is complete and the thread marks itself as finished. If not, the process repeats from step 6.

Figure 5: Algorithm Flow

This algorithm at first glance appears to elegantly avoid all race conditions and maximise the work throughput of threads.

2 Implementation

This section is best read as an accompanying document when reading the source code.

The interface of the program is as follows:

```
Usage: main [options]
Options:
    -P      double; the desired precision to work to.
    -p      int; number of threads to run on. (Note: one thread is a control thread
            only p-1 threads will run the computation).
    -n      int; the dimension of the provided array (must be square).
    -a      space-separated list of integers or doubles; the array to be computed
            integers will be converted to doubles.
    -af     file; the array provided in a file. Each element must have its own line.
```

Figure 6: Program Interface

For convenience, a simple Makefile was used to compile the program with the required libraries and subsequently run the program with some set of parameters. Behaviour for incorrect parameters, or an insufficient number of parameters, is not defined (as it did not appear to be the purpose of the coursework). Some notes on the code:

2.1 main function: compute()

compute() is the main function of the program. Its interface is as follows:

```
double **compute(int p, double P, int n, double **a)
```

which corresponds to the parameters defined in Figure 6. Without going into excessive detail, this function:

1. Identifies the cells in Pass #1 and #2 from Figure 4
2. Divides the work between the threads
3. Creates threads
4. Checks if threads are done; upon completion it returns array a.

The function that each thread is given to execute is multi-parameter, so a struct **ARGT** is created with the necessary values, which is then passed through **pthread_create()**.

2.2 thread function: compute0()

compute0() is the function that the slave threads (in contrast to the control thread which runs **compute()**) run. compute1() is the sequential analogue of this function that will be used to test correctness and to evaluate efficiency later on. compute0() implements 3 thread barriers, for which each of the slave threads must reach before they can continue to execute. These are; *check*, *passa*, and *passb*. The purpose of *passa* and *passb* should be fairly self explanatory from the description in Figure 5, and the *check* barrier is used to make sure all passes are finished before the check is done. compute0 is an infinite loop, where threads only have the ability to set a flag signalling their status and not cancel themselves. This was because of a technicality where, if a given thread cancelled itself before others, the barrier could not be passed and the threads would hang. As a result of this, the control thread now performs all thread management.

2.3 Auxiliary Functions and Structures

2.3.1 structs

CELL	The cell information for each thread of which cells to compute - a linked list whose values contain the (x,y) coordinates, the current value, the previous value, and the previous 4 values of its neighbours.
TSTAT	Holds the status of each thread; whether is has finished its computation, and the number of passes it made.
ARGT	The argument object passed to compute() because of the one-argument restriction on **pthread_create()**. Contains P, a (the same as the values from the interface in Figure 6), and two **CELL**s, for Pass #1 and Pass #2 respectively.

2.3.2 functions

new_tstat() update_tstat() find_tstat()	functions to create a **TSTAT**, for a thread to mark itself complete and to find a tstat by thread id respectively.
new_workt()	Returns a group of work for a thread. Takes the linked list of **CELL**s a, and returns the first n elements of the list, where n is the work for a given thread.
check_done()	Checks if a thread is done. Takes the absolute value of the difference between the current and previous values of the auxillary cells for each cell in a threads workload, and checks if their sum is less than or equal to $P*4$, since when averaged out they would not change the value more than the desired precision.

There are a few more functions in the program, however they are all self explanatory and are not worth mentioning; the above should be sufficient for understanding the function of the program.

3 Analysis

3.1 Scalability Investigation

3.1.1 Efficency

Running on 44 threads with an matrix of dimension $n = 1000$, the program completes in ~ 63 seconds. The same size matrix completes sequentially in 12 minutes 35.5 seconds[1]. This gives us an efficiency of:

$$E_p = \frac{S_p}{p} = \frac{755.5}{44*63}(seconds) = 30.1\%$$

Which is pretty poor. I suspect this is to do with the data structure I have used; each thread is given a linked list of cells, and when splitting the work between threads, copies of sections of the linked list are made and given to each thread, which I suspect is a slow operation. Specifically I believe this section:

[1]all times in this section are given on average, over 10 runs.

```
while(i<p){
    if(extraa){
        workta = new_workt(aa->next,worka+1);
        extraa--;
        aa = find_next(aa, worka+1);
    }
    else{
         workta = new_workt(aa->next,worka);
         aa = find_next(aa, worka);
    }
    if(extrab){
        worktb = new_workt(bb->next,workb+1);
        extrab--;
        bb = find_next(bb, workb+1);
    }
    else{
         worktb = new_workt(bb->next,workb);
         bb = find_next(bb, workb);
    }
    args = new_args(a,workta,worktb,P);
    int err = pthread_create(&threads[i],NULL,&compute0,args);
    if(err){
        printf("Thread Creation Error, exiting..\n");
    }
    i++;
}
```

Figure 7: Initial cell assignment.

Was the main cause of the inefficiency. Evidence for this was the fact that on the same $n = 1000$ matrix, when run on 22 threads, the runtime was approximately the same as for 44; thus there is some fixed amount of work happening in both, which is likely this work done in the control thread. I attempted to optimize and improve this by moving the work discovery process (as is done in Figure 7) into individual threads, but all I accomplished was further confusion and a waste of 2 days. I decided to leave the algorithm as is as it is correct and predictable, albeit inefficent. The efficiency seems to be depending on finding an optimal number of processors for a given size array; a certain number of threads can have high efficiency. For example, running on the same $n = 1000$ matrix (this time with different values), 17 threads (that is, 1 control thread and 16 workers), finishes in $\sim$ 96 seconds, giving an efficency of $\sim$ 61%(against a sequential time of 16m44s on the same dataset), a non-trivial improvement. While the code may not be maximally efficent, by Gustafsson's law, we should see an improvement in efficiency as n gets larger; that is, the sequential aspect of the code shrinks in comparison to the computations done related to the large value of n. When the program is run with a matrix of over 10 million entries ($n = 3172$), the

3.2 Correctness Testing