CM30225 Parallel Computing Assessed Coursework Assignment 1

October 26, 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

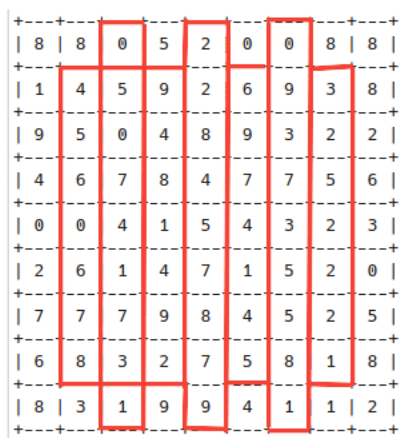
0	4	6	6	6	5	5	5	2	2	2	0
7	1	2	2	8	1	1	6	5	5	3	5
4	0	6	6	0	4	4	9	1	1	0	0

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