CM30225 Parallel Computing
Assessed Coursework Assignment 2

Contents

Compiling and Running

My program should be compiled using:

```
mpicc main.c -lm
```

Then run with:

```
mpirun -np $SLURM_NTASKS a.out [source] [print?] [precision]
              [threads] [dimension] [filename]
```

Where:

- string [source] relates to the implementation you wish to run
    - ‘seq’ for sequential
    - ‘shared’ for shared memory
    - ‘dist’ for distributed memory

- char [print?] is for printing result array
    - ‘y’ for printing

- double [precision] is the precision to work to
    - e.g. 0.001, 0.5, or 2

- int [threads] is the number of threads to use
    - Only used when using the shared memory implementation

- int [dimension] is the width or height of the square array provided

- string [filename] is the name of the file in which the initial array is stored
    - File must contain integer array values less than 10 on one line, delimited by any character
        - E.g. “9 5 4 1 7 8 2 4 6” for a 3x3 array delimited by spaces

Example running distributed algorithm:

```
mpirun -np $SLURM_NTASKS a.out dist n 1 1 1000 data/1000x1000.txt
```

You can also run a set of predefined tests with:

```
mpirun -np $SLURM_NTASKS a.out
```

Approach

Method

The method I used to solve the relaxation problem in a distributed memory architecture is as follows:

1. Split the processors so that there is a maximum of 1 process per row
2. Assign each process an equal, or near equal, number of rows to compute
3. Have each process calculate averages for its first and last row
4. Asynchronously send/receive the averages to/from the process handling the rows above and below
5. Calculate averages for the rest of the rows
6. Have each process check whether its rows are settled to below the precision
7. Wait on the asynchronous sends and receives, if they have not already completed
8. If any process has not settled, then repeat from step 3
9. If all processors are settled, then gather the rows from all processors to the base process

Asynchronous Messaging

If I were to use synchronous/blocking messaging (`MPI_Send(), MPI_Recv()`), then each process would have to wait on each message to successfully send or receive, meaning that there would be a lot of wasted time waiting on input/output (IO) which could be used on computation.

I chose to use asynchronous/non-blocking messaging (`MPI_Isend(), MPI_Irecv()`) because it allows each process to send/receive the rows that are needed to/from neighbouring processors without waiting for the message to complete. This is especially useful in my method as I calculate the averages for the upper and lower rows first, and then can immediately continue computation on other rows.

With this method there is a hope that by the time that computation has finished, the messages will have been sent/received successfully, allowing each process to move on to the next computation straight away. However, as IO is never guaranteed to happen within a certain time frame due to networking issues, I ensure that messages are complete by using the `MPI_Wait()` function after computation has finished.

It is worth noting that using non-blocking messaging does not reduce the messaging overhead, it simply allows other code to be executed whilst waiting for messaging to complete in another thread.

Settling

It is a simple task to check whether the elements in each process have settled down to below the given precision, but the task becomes more complicated when all processors need to know the settled status of all the other processors.

The first solution I found was to use `MPI_Allreduce()` with the `MPI_LAND` function, which is analogous with a logical AND. What this does is collects the settled values from all processors, and will return `TRUE` if all the values are `TRUE`, or return `FALSE` if any of the values are `FALSE`. This means that the function will only return `TRUE` if all the processors have settled, indicating that the entire array has settled.

I could have used the `MPI_Reduce()` function, which is similar to the one described above, but it only sends the result to the base process. This would mean that a one-to-many message would need to be sent from the base process, such as a broadcast, which is bad because it means a higher messaging overhead, i.e. more time waiting instead of computing.

However, I opted for a non-blocking solution which included the use of `Isend()` and `Irecv()` with all processors. This is a good way of overlapping communication and computation, thus allowing more time for computation, which makes it a better solution than the ones described above.

Gathering

After all processors have settled, we need to collect the computed averages from all of the processors. This can be done by using `MPI_Gather()` which collects an equal count of data from each process, however some processors may have more rows than others, so it is more appropriate to use `MPI_Gatherv()`, which collects a variable count of data from each process.

It is necessary to create two arrays for the use of this function; one containing the counts of data to be sent from each process, and another containing the displacements in the array to output the data.

I chose to only gather the array in the base process than in all processors because it may take longer to send values from all processors to all processors, than just from all processors to one process, thus reducing the messaging overhead.

Correctness Testing

As described in the previous coursework assignment, I created a large testing harness for correctness testing. The tests can be run using:

```
mpirun -np $SLURM_NTASKS a.out
```

The testing harness contains the following sets of tests:

1. Testing all implementations against a small hand-calculated array
2. Testing the shared implementation against the sequential implementation
3. Testing the distributed implementation against the sequential implementation

We can assume that the sequential implementation is correct if it compares as equal against a number of hand calculated arrays. Then it follows that the other implementations are correct if their result is the same as the sequential algorithm for the same starting array and precision.

With this testing harness I was able to test against any size of array, to any precision, which is a much more powerful method than just testing small hand-calculated arrays against each implementation.

To further my testing, I performed iterative tests of my sequential algorithm against my other algorithms to ensure that changing an input value does not affect the result by means of a race condition or otherwise:

- Increasing thread count from 1 to 16
- Increasing dimension from 3 to 20
- Increasing precision from 0.1 to 0.0000000001

As all of the above tests passed, I am confident that my implementation works as required.

Scalability Investigation

Time

The table and graph below show an experiment that aims to find out the relationship between time taken to perform relaxation and the number of processors used. Results are taken from running my distributed algorithm with increasing numbers of processors, using the same 10,000x10,000 matrix. The matrix was generated with random integers and saved into a file named 10000x10000.txt.

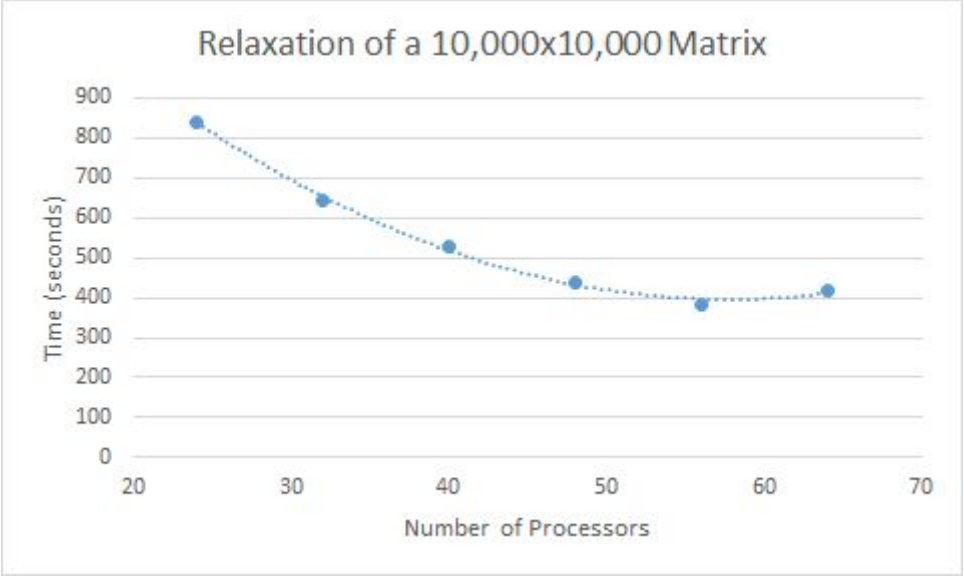
The command I used:

```
mpirun -np $SLURM_TASKS a.out dist n 1 1 10000
data/10000x10000.txt
```

Table:

Nodes	Tasks per Node	Total Processors	Dimension	Precision	Time (s)
4	2	8	10,000	0.1	DNF
4	4	16	10,000	0.1	DNF
4	6	24	10,000	0.1	841
4	8	32	10,000	0.1	644
4	10	40	10,000	0.1	528
4	12	48	10,000	0.1	438
4	14	56	10,000	0.1	385
4	16	64	10,000	0.1	416

Scatter graph with trendline:



It is evident from the graph that as the number of processors increases, the time it takes to perform relaxation decreases exponentially. That is the case for all processor counts, except for the 64 processors where the time has a slight increase from that of the 48 processors time. This is probably due to using an extra node, which means that there will be a larger communication overhead as messages take longer between nodes than between processors on the same node.

Some times as shown as DNF, which stands for Did Not Finish. Here, the array was too large to be relaxed by the number of processors used in the maximum amount of time allowed for tasks on Balena, 15 minutes. However, using the equation of the trendline, it can be estimated that the time for 16 processors would have been about 1,070 seconds, and the time for 8 processors would have been about 1,350 seconds.

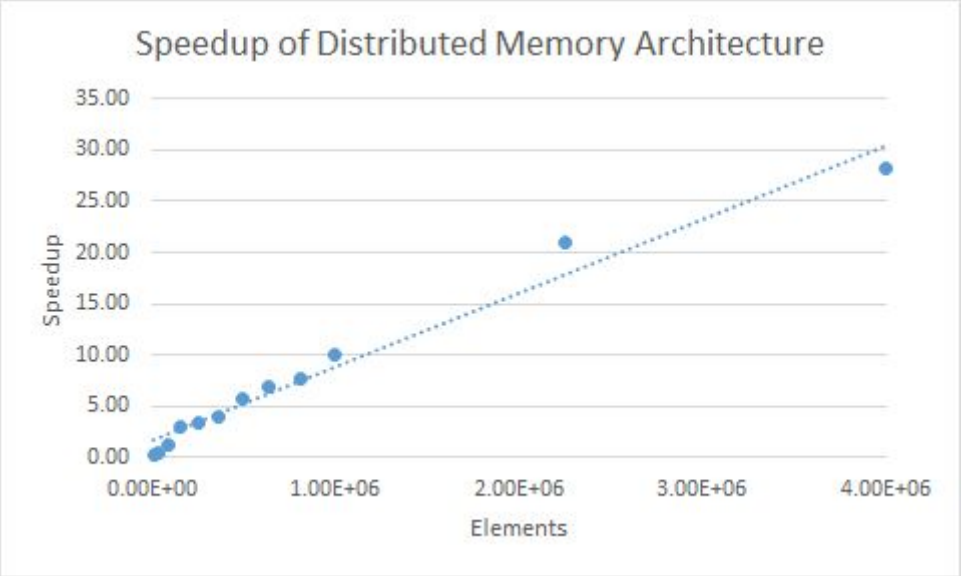
Speedup

Speedup is defined as:

$$Sp = \frac{time\ on\ a\ sequential\ processor}{time\ on\ p\ parallel\ processors}$$

The table and graph below show the relationship between speedup and number of elements in the matrix using 64 processors:

Elements	Time on Sequential (s)	Time on Distributed (s)	Speedup	Efficiency
10,000	1	4	0.25	0%
40,000	2	4	0.5	1%
90,000	5	4	1.25	2%
160,000	12	4	3	5%
250,000	14	4	3.5	5%
360,000	16	4	4	6%
490,000	35	6	5.83	9%
640,000	41	6	6.83	11%
810,000	62	8	7.75	12%
1,000,000	80	8	10	16%
2,250,000	189	9	21	33%
4,000,000	282	10	28.2	44%



The graph shows a linear increase of speedup for an increasing number of elements, which means that an increase in elements used brings about a constant increase in the speedup, for the same number of processors.

My results agree with Gustafson's law as we should be able to see near perfect speedup when the problem size becomes large enough. This is because as the problem size increases, the computation time will become large enough to ensure that all communications will be complete before the computations are complete, which means that there is less serialisation.

Efficiency

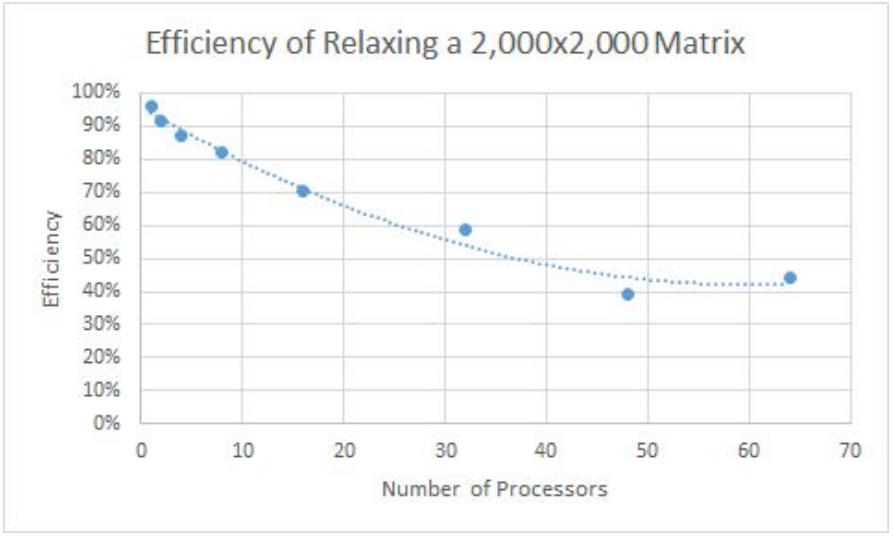
Efficiency is defined as:

$$Ep = \frac{Sp}{p} = \frac{time\ on\ a\ sequential\ processor}{p\ x\ time\ on\ p\ parallel\ processors}$$

The graphs below show the relationships between increasing processor counts and increasing dimension against the time taken to relax:

Dimension	Sequential Time (s)	Processors	Distributed Time (s)	Speedup	Efficiency
500	14	1	14	1.00	100%
500	14	2	7	2.00	100%
500	14	4	5	2.80	70%
500	14	8	4	3.50	44%
500	14	16	2	7.00	44%
500	14	32	5	2.80	9%
500	14	48	6	2.33	5%

500	14	64	4	3.50	5%
1000	80	1	82	0.98	98%
1000	80	2	43	1.86	93%
1000	80	4	25	3.20	80%
1000	80	8	14	5.71	71%
1000	80	16	9	8.89	56%
1000	80	32	7	11.43	36%
1000	80	48	7	11.43	24%
1000	80	64	7	11.43	18%
1500	189	1	188	1.01	101%
1500	189	2	110	1.72	86%
1500	189	4	57	3.32	83%
1500	189	8	32	5.91	74%
1500	189	16	19	9.95	62%
1500	189	32	12	15.75	49%
1500	189	48	10	18.90	39%
1500	189	64	9	21.00	33%
2000	282	1	293	0.96	96%
2000	282	2	154	1.83	92%
2000	282	4	81	3.48	87%
2000	282	8	43	6.56	82%
2000	282	16	25	11.28	71%
2000	282	32	15	18.80	59%
2000	282	48	15	18.80	39%
2000	282	64	10	28.20	44%



It is interesting to see that there was superlinear speedup in one of the tests above; highlighted in orange. The test shows that for a processor count of 1, the distributed algorithm was faster than the sequential algorithm. This is a strange result to see because the distributed algorithm has larger amount of setup than the sequential algorithm, so I would have expected to see it take longer, rather than shorter.

A simple explanation for this superlinear speedup is randomness, and as the speedup is only a small percentage over perfect speedup, this could be the case. However, after further looking into the sequential and distributed algorithms I found that the way I calculate the settled flag is different. The sequential settled flag is calculated by iterating through every element, and only comparing elements if the array has not been found to be settled yet. In contrast, the distributed settled flag is calculated by iterating through the array until an unsettled element is found, then exiting the array. This means that in the sequential algorithm, the settled check will always loop through the entire array (best case = worst case = number of elements), whereas in the distributed algorithm the settled check will exit when the settled flag has been determined (best case = 1, worst case = number of elements). Thus the sequential algorithm may take longer than the distributed algorithm for large arrays.

My results above mostly agree with Amdahl's law, which says that every program has a natural limit on the maximum speedup it can attain, regardless of the number of processors used. The only rows that disagree with the law are highlighted in the table. They disagree because they show a speedup of greater than or equal to the number of processors used. However, this could be due to the low resolution used to measure time with (1 second). For example, if both the sequential time and the distributed time are both equal to 7 seconds in the table, they may have had actual values of 6.8s and 7.3s respectively, thus it is possible that a perfect speedup was not attained.

Additional Time

If I had more time to invest in this coursework, I would look into sharing unfinished computations between processors. This could be done through a manager/worker architecture where a manager provides a list of jobs to be completed, and each worker takes work as it needs it. Using this architecture would be good because it has automatic load balancing. However, this may prove to be impracticable because IO operations are slow in comparison to arithmetic operations. However, I think that given a large enough array of uncomputed data, it may be worth sharing between processors.

There is also the possibility of assigning processors work based on their topology, or physical location, to reduce communication overhead. The processors should be organised in such a way that they are close to the two processors that they communicate with in the code.

I would also like to look into probing for messages. This could be useful when checking for other processors settled flags. Using probing I would be able to abstain from sending the settled flag to other processors before the processor itself has settled, which would save a lot of communication and would possibly reduce the overall time to settle.

Excerpts from Testing

```
1/1 tests passed in test_expected_1
1/1 tests passed in test_expected_2
1/1 tests passed in test_expected_3

- 10/10 tests passed in test_dimension_shared_3
- 10/10 tests passed in test_dimension_shared_4
- 10/10 tests passed in test_dimension_shared_5
- 10/10 tests passed in test_dimension_shared_6
- 10/10 tests passed in test_dimension_shared_7
- 10/10 tests passed in test_dimension_shared_8
- 10/10 tests passed in test_dimension_shared_9
- 10/10 tests passed in test_dimension_shared_10
- 10/10 tests passed in test_dimension_shared_11
- 10/10 tests passed in test_dimension_shared_12
- 10/10 tests passed in test_dimension_shared_13
- 10/10 tests passed in test_dimension_shared_14
- 10/10 tests passed in test_dimension_shared_15
- 10/10 tests passed in test_dimension_shared_16
- 10/10 tests passed in test_dimension_shared_17
- 10/10 tests passed in test_dimension_shared_18
- 10/10 tests passed in test_dimension_shared_19
- 10/10 tests passed in test_dimension_shared_20
18/18 tests passed in test_dimension_shared

- 10/10 tests passed in test_precision_shared_0
- 10/10 tests passed in test_precision_shared_1
- 10/10 tests passed in test_precision_shared_2
- 10/10 tests passed in test_precision_shared_3
- 10/10 tests passed in test_precision_shared_4
- 10/10 tests passed in test_precision_shared_5
- 10/10 tests passed in test_precision_shared_6
- 10/10 tests passed in test_precision_shared_7
- 10/10 tests passed in test_precision_shared_8
- 10/10 tests passed in test_precision_shared_9
- 10/10 tests passed in test_precision_shared_10
11/11 tests passed in test_precision_shared

- 10/10 tests passed in test_threads_shared_1
- 10/10 tests passed in test_threads_shared_2
- 10/10 tests passed in test_threads_shared_3
- 10/10 tests passed in test_threads_shared_4
- 10/10 tests passed in test_threads_shared_5
- 10/10 tests passed in test_threads_shared_6
```

```
- 10/10 tests passed in test_threads_shared_7
- 10/10 tests passed in test_threads_shared_8
- 10/10 tests passed in test_threads_shared_9
- 10/10 tests passed in test_threads_shared_10
- 10/10 tests passed in test_threads_shared_11
- 10/10 tests passed in test_threads_shared_12
- 10/10 tests passed in test_threads_shared_13
- 10/10 tests passed in test_threads_shared_14
- 10/10 tests passed in test_threads_shared_15
- 10/10 tests passed in test_threads_shared_16
16/16 tests passed in test_threads_shared

- 10/10 tests passed in test_dimension_dist_3
- 10/10 tests passed in test_dimension_dist_4
- 10/10 tests passed in test_dimension_dist_5
- 10/10 tests passed in test_dimension_dist_6
- 10/10 tests passed in test_dimension_dist_7
- 10/10 tests passed in test_dimension_dist_8
- 10/10 tests passed in test_dimension_dist_9
- 10/10 tests passed in test_dimension_dist_10
- 10/10 tests passed in test_dimension_dist_11
- 10/10 tests passed in test_dimension_dist_12
- 10/10 tests passed in test_dimension_dist_13
- 10/10 tests passed in test_dimension_dist_14
- 10/10 tests passed in test_dimension_dist_15
- 10/10 tests passed in test_dimension_dist_16
- 10/10 tests passed in test_dimension_dist_17
- 10/10 tests passed in test_dimension_dist_18
- 10/10 tests passed in test_dimension_dist_19
- 10/10 tests passed in test_dimension_dist_20
18/18 tests passed in test_dimension_dist

- 10/10 tests passed in test_precision_dist_0
- 10/10 tests passed in test_precision_dist_1
- 10/10 tests passed in test_precision_dist_2
- 10/10 tests passed in test_precision_dist_3
- 10/10 tests passed in test_precision_dist_4
- 10/10 tests passed in test_precision_dist_5
- 10/10 tests passed in test_precision_dist_6
- 10/10 tests passed in test_precision_dist_7
- 10/10 tests passed in test_precision_dist_8
- 10/10 tests passed in test_precision_dist_9
- 10/10 tests passed in test_precision_dist_10
11/11 tests passed in test_precision_dist

- 8/8 tests passed in all_10
```