PARALLEL COMPUTING PROJECT

SHAMEEL NKOSI

STD Num: 1814731

INTRODUCTION

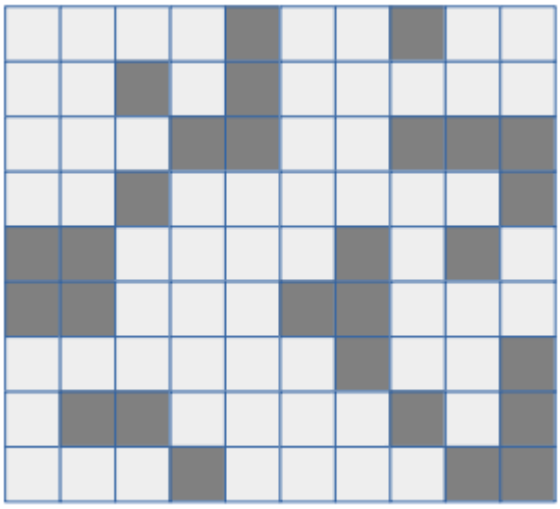
In this project we aim to solve two problems, each in parallel, Conway's Game of life and Bitonic sort. These are to be solved using MPI, MPI and OpenMP respectively. The reason we do this in parallel is to achieve performance by reducing the time taken to solve the problems at hand. MPI stand for Message Passing Interface, which is a distributed memory interface where processors can communicate with each other. On the other hand, OpenMP can use both shared and distributed memory approach. Below we discuss how these are used and the approach taken to implement the algorithms in parallel. The format of this report is that it will start with discussing the Game of life, all of it, then followed the Bitonic search as if it were two different reports.

CONWAY'S GAME OF LIFE

Description

Conway's game of life is a cellular automata problem on a 2 dimensional grid, where every point on the grid is a cell with 8 neighbors which are also cells, this includes the cells at the edges of the grid (all cells have 8 neighbors). It wraps around, as if these cells were living on a globe. Cells live to survive the current generation or die depending on a few rules regarding it's neighbors which are:

- If a dead cell has 3 neighbors who are alive, it comes to life in the next generation.
- If a cell is alive and has 2 or 3 neighbors alive, it survives the current generation.
- If a cell is alive and has less than 2 neighbors alive, it dies of under population.
- If a cell is alive and has more than 3 neighbors alive, it dies of over population.



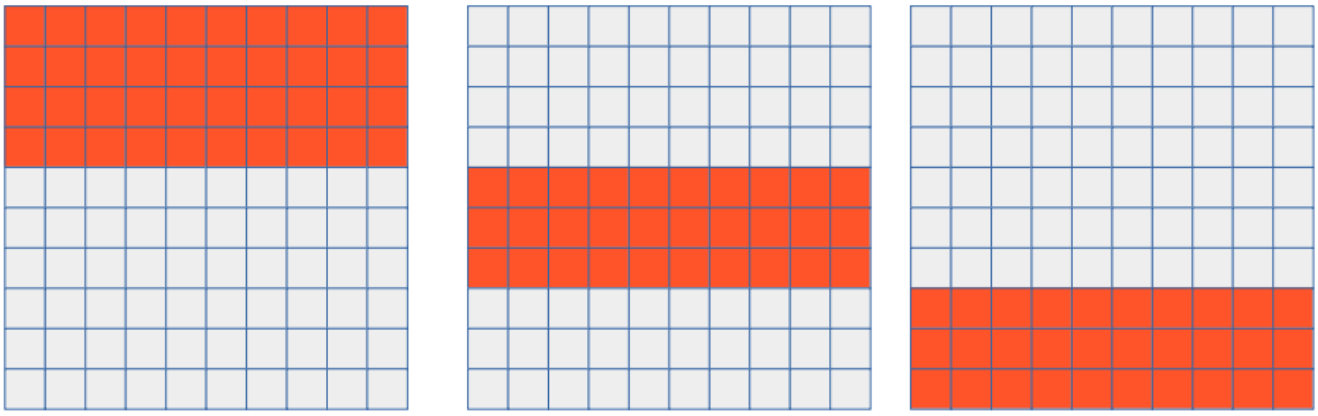
Codes

There are three C files namely, GOL_Sequential.c , GOL_Parallel.c and GOL_Simulation.c . GOL_Simulation.c simulates the game of life on a 55 by 55 grid for 500 generations. The other 2 files are the serial and parallel implementations of the Game Of Life. These are seperate codes

which means calculating speedup and efficiency will be done manually and not in the code. The reason these codes are seperate is because MPI is creates copies of the same program, and adding serial code together with parallel code duplicates the serial code and this isn't efficient. Please note these code requires you to input the number of generations only but doesn't require the size of the grid. The sizes of the the grids are initialized to 55x55, if one wishes to change these they will have to manipulate the code itself.

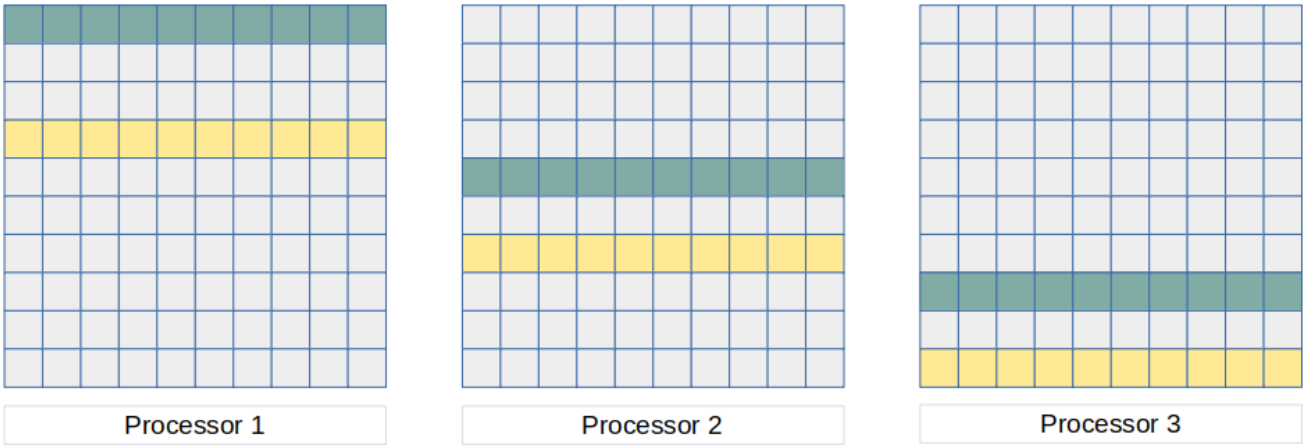
Parallel Implementation

Implementing the game of life in serial is trivial as mentioned in the handout. It is as trivial as implementing the rules above. The grid is represented by ones and zeros. Where 1 represents a living cell and zero represents a dead cell. In the above grid however, the dark blocks represents the living cells and the light blocks represents the dead cells.
A parallel implementation involves splitting work among different processors. Where we try to give each processor a fair share of work. We therefore split the the grid into sub-grids. Where each processor works on a sub-grid assigned to it. The sub-grid is divided by certain rows. Take for instance the above 10 by 10 grid running on 3 processors, the algorithm figures out which processor should solve which part of the grid. In the above case, processor zero will take the first four rows, the second processor will take the three rows after the first four and the last processor takes on the last three rows, shown in the diagram below.



The calculations and assigning of the processors' works as follows:

- Calculate the chunk size (the number of rows) for each processor and it's offset.
- The offset is the row at which the processor begins to calculate the next generation's states.
- The chunk size are calculated as number of rows divided by the number of processor and the remainder is added to the rows of the first processor.
- Before MPI is initialized, The grid is world is generated, and when the MPI is initialized, every processor has the original generation in it's memory and calculations start from there.



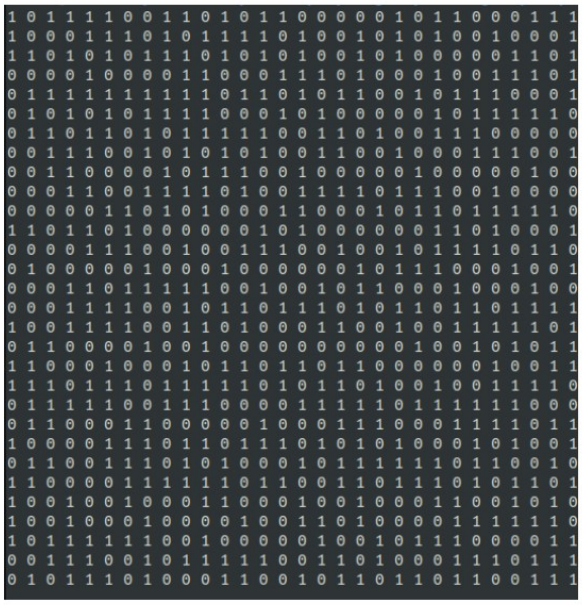
- Processors are set up in a ring topology, each green row is sent to its previous processor i.e. the green on processor one is sent to processor 3 since the yellow row in processor 3 needs processor one's green row to calculate the next generation.
- Each yellow row is sent to the to the next processor, since it's needed by the next processors green row.
- Each processor then prints it's part, this happens sequentially before the next generation is calculated.

In the Parallel implementation, communication comes at the expense of performance, therefore this is the only communication that takes place to just so that communication is reduced, and therefore maximizing the performance. One would then ask, How is the next generation printed? Every processor prints it's part to the screen, one processor at a time, this then becomes a sequential process as processor 2 will wait for processor 1 to print it's part before it prints it's part. This gives the illusion that the next generation's grid is one grid when in fact the cells are scattered in different processors' grids. This doesn't affect the generations, but rather reduces communication and sending of large data from one processor to the other. The code snippet below shows the above description.

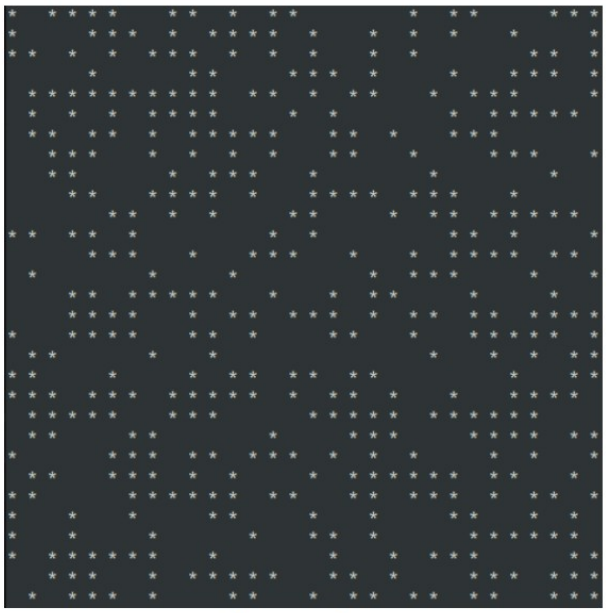
```
int main(int argc, char * argv[]) {
    setGrid(grid, followingGen);

    int numtasks, rank, source, dest, chunksize, offset, tag1, tag2;
    int top[SIZE], bottom[SIZE] ;


    MPI_Status status1 ,status2;
    MPI_Init( & argc, & argv);
    MPI_Comm_size(MPI_COMM_WORLD, & numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, & rank);

    tag1 = 1; // tag for chunk size
    tag2 = 2; // tag for offset
    source = MASTER;
    chunksize = (SIZE) / numtasks;
    offset = (SIZE) % numtasks;
    if(rank == MASTER){

        chunksize = (SIZE % numtasks) + chunksize;
        offset = 0;
    }
    for (int i = 0; i < numtasks - 1; i++) {
            if(rank== i + 1){
                offset += chunksize*rank;
            };
    }


    for(int i = 0 ; i < numtasks ; i++){
         MPI_Barrier(MPI_COMM_WORLD);
        if(rank == i){

            for(int j = 0 ; j < SIZE ; j++){

                top[j] = grid[offset][j];
                bottom[j]=grid[offset+chunksize-1][j];

            }
            MPI_Send(&top , SIZE, MPI_INT , (numtasks+(i-1))%numtasks , tag1 , MPI_COMM_WORLD);
            MPI_Send(&bottom , SIZE, MPI_INT , (numtasks+(i+1))%numtasks , tag2 , MPI_COMM_WORLD);
        }
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Recv(&bottom , SIZE , MPI_INT , (rank+1)%numtasks ,tag1, MPI_COMM_WORLD,&status1);
    MPI_Recv(&top, SIZE , MPI_INT , (numtasks+(rank -1))%numtasks , tag2 , MPI_COMM_WORLD , &status2);

    MPIcopyArr(top , bottom , offset , chunksize);
    MPIupdatefollowingGen(chunksize,offset,followingGen);
    MPIupdateGrid(chunksize , offset , followingGen);

     for(int i = 0 ; i < numtasks ; i++){
        MPI_Barrier(MPI_COMM_WORLD);
        if(rank == i){
            printMypart(grid , offset , chunksize);
        }
    }


    MPI_Finalize();

    return 0;
}
```

Experimental Setup

Below we discuss the results of the sequential vs parallel on the same set of inputs. We further show that both parallel and sequential implementations are scalable and show results thereof.



Grid as ones and zeros



Grid as stars and blanks

The original grid is one's and zero's how ever, the out put is showed as stars for cells that are alive and blank spaces for dead cells. This makes it easy to see the simulation of the game of life. Below is the same initial generation, of a 30 by 30 size grid i.e. 900 cells.
For the purposes of clear visuals, any grid snippet included will be made in the form of the stars and black so that it is easy to compare. The results below shows a progression from generation 0 to generation 1 only, only because the the algorithm doesn't change as time goes by. Before we discuss performs, the following visuals are comparisons between the sequential and parallel implementations of the above 30 by 30 grid. Pay close attention to the commands used to run the script.

Sequential after gen 0

Parallel after gen 0

The image above shows a comparison of results of running the program in sequential vs in parallel run using 5 processors. The results are the exact same which is proof of correctness.

Performance calculations, Evaluation of results and Discussions

I observed below, that with a small problem size, the sequential does better or completes faster than the parallel implementation, as the problem increases, the parallel implementation starts to do better than the sequential implementation and eventually it the parallel implementation completes half the time as compared to the sequential. Below are is the table of results.

Size (n)	Serial	Parallel – (num processors)
10	0.000097s	0.000125s – (6)
20	0.000189s	0.000340s – (6)
30	0.000223s	0.000144s – (6)
40	0.000394s	0.000227s – (6)
50	0.000503	0.000231s – (4)
70	0.001024s	0.000656s – (4)
85	0.001535s	0.000541s – (4)
100	0.001603s	0.000623s – (4)

Table – comparison of serial vs parallel implementation

Using the data gathered above, we now wish to measure two performance metrics, Speedup and Efficiency. This are only calculated for the parallel implementation as it wouldn't make sense to calculate for the serial implementation. Speed up is a measure of how much does additional Processing Elements help. Which is calculated as:

$$S(p) = Ttotal(1) / Ttotal(p)$$

which is the total time taken to solve the problem in serial divided by the time taken to solve the problem in parallel.

On the other Hand, Efficiency, which is in the range from 0 to 1, the closer to one, the better and it is measured as follows:

$$E = S(p) / P$$

which is the Speedup divided by the number of processors. In the table below, we append the to the table above 2 more columns, one for the Speedup and the other for efficiency.

Size (n)	Serial	Parallel – (num processors)	Speedup	Efficiency
10	0.000097s	0.000125s – (6)	0.776	0.13
20	0.000189s	0.000340s – (6)	0.556	0.09
30	0.000223s	0.000144s – (6)	1.549	0.26
40	0.000394s	0.000227s – (6)	1.736	0.29
50	0.000503	0.000231s – (4)	2.177	0.54
70	0.001024s	0.000656s – (4)	1.561	0.39
85	0.001535s	0.000541s – (4)	2.837	0.71
100	0.001603s	0.000623s – (4)	2.573	0.64

Table – serial time , parallel time, speedup and efficiency

If you multiply the efficiency by 100, it gives you a percentage of how much more efficient is using x amount of processing elements. In the above, we two tables, as we scale up with the size of the grid, we also scale down on the number of processors working on the grid. This shows that the parallel implementation is scalable and efficient.

References and citations

https://computing.llnl.gov/tutorials/mpi/
https://en.wikipedia.org/wiki/Conway's_Game_of_Life
https://www.youtube.com/watch?v=FWSR_7kZuYg&ab_channel=TheCodingTrain

Bitonic Sort

Description

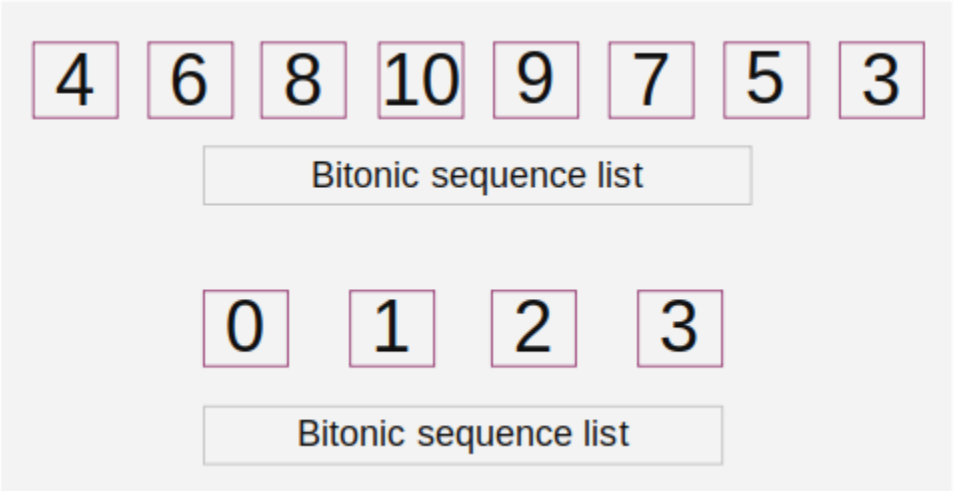
Bitonic sort is an intresting type of sort. It makes possible it to sort an a list of items in parallel. Bitonic sort sorts a bitonic sequence. A bitonic sequence is a list of numbers which are at first monotonically increasing and then monotonically decreasing or vice versa. Given an arbitrary list of numbers, this list can be converted to a bitonic sequence in parallel and then sorted in parallel. In this project however, we only sort the bitonic sequence and not convert a list of numbers to one. Moreover, we are going to sort and interesting case of the bitonic sequence, one of length 2^n with the first half being monotocally increasing and the second half being monotonically decreasing or vice versa.

2^3 (8) sized bitonic sequnce

I would like to believe I have taken a different approach compared to most of peers and below I discuss the implementation approach as well as the parallel implementation of the bitonic sort algorithm.

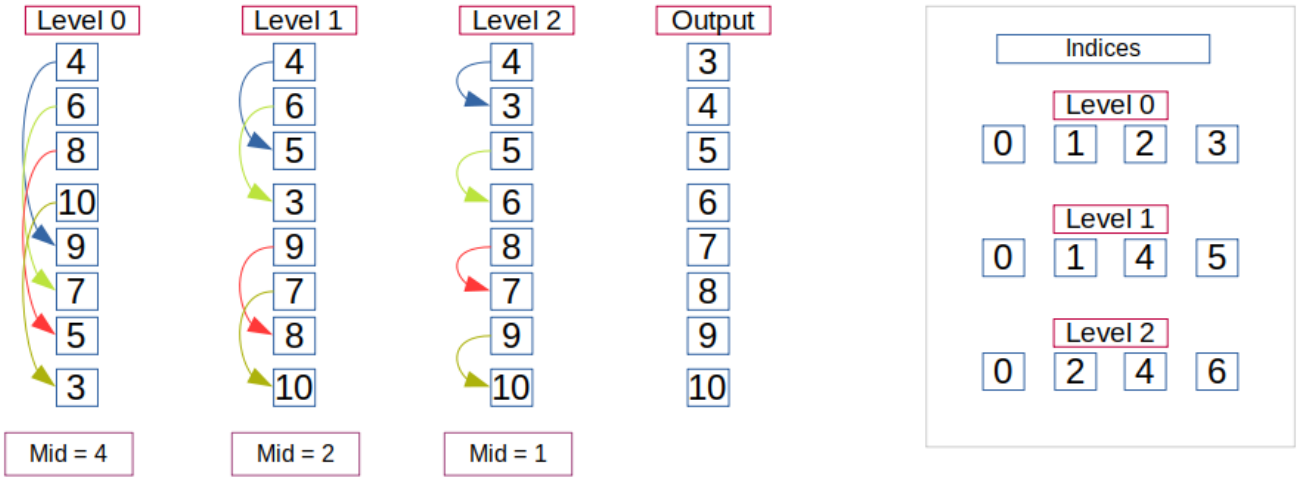
General approach

The main part, which enables easy parallelization is that I keep track of two lists, a list of the bitonic sequence and a list of indices which is half the size of the bitonic sequence list. The indices list stores a numbers which are reference indices of the bitonic sequence.



In addition to keeping track of these two lists, we keep keep of an integer “mid” , which basically tells us how to compare and switch positions of items in the list, this will be clearer as we progress with the report. Just to digress a little bit, recall we mentioned that the bitonic list is of length 2^n, there are always n levels of the lists manipulation, and at each level, the both the lists

are updated, and the integer “mid” changes as well. Below is a digram which explains the approach.



We have 3 levels, each indices is mathematically calculated at the beginning of each level. Mid divides by two from one level to another. This is how the sorting works:

- at level zero, initialize the level zero index list.
- Initially, set mid = 4 which is the size of the indices list.
- Move from level the current level as follow:
    1. compare bitonic_list[0] with bitonic_list[0+mid],
    2. if swap if bitonic_list[0] is larger than the two
    3. do this for every index in the indices list
    4. after all swaps, you should have the bitonic list in level one
    5. update your indices list, as well as “mid”.
    6. Then go back to 1. and repeat until the list is sorted.
- The arrows in the picture above basically shows how mid works together with the indices list to magically sort the list.

Most implementations keep things simple and use one list and sophisticated maths and loop dependacies to sort a list. With the implementation above, it is relatively easy to parallelize the task, but keeping too much data puts us at a high risk of messing with performance.
Every program code requires input, the size of the array and the array itself.

Parralel implementation

This part should be relatively short as bulk of the explanation has been done above. Starting with openMP:

- Store every variable as a global variable, this includes the lists we are keeping track of.
- In the above example, we’d have a maximum of 4 processors working on the problem, 4 being the size of the indices array.
- We use the for construct applied on the indices list, this means at every level, each processor will tackle an operation.

- If there are two processors, then the for construct, will divide the indices list into two and every processor will tackle two elements of the indices array each and this will be in parallel.
- If for any reason we'd use more processors than the size of the indices array, these will be idle and might be bad for performance.

The openMP implementation is straight forward because we are using a shared memory approach, things get a little bit more interesting when it comes to MPI since it uses a distributed memory approach, and it works as follows:

- we initialize both lists… again.
- We then initialize MPI, which creates an instance of the program for every processor which inherits every initialized variable and lists and keeps it's own copies.
- The main thread or processor is the only one that is responsible for updating the indices list, and before the next level, it broadcasts it to all other processors.
- For a small size bitonic list, the above might be computationally expensive but the as there list size gets larger, we might have good performance, but we are yet to find out.
- Every processor calculates it's own chunk and calculates which chunk it should work on, this is achieved through the knowledge of other initialized variables as well as the knowledge of how many processors are there in total.
- Now that every processor has it's own list, it works on it's own chunk and send the entire partially updated list to the main thread.
- When the main thread receives all the partially updated lists, it uses these to create on fully updated list in for the next level, it then broadcasts this list to all other processors for the next level.
- And the above points describes how processors communicate, this might not be the most effiecient way, we could have used scatter for instance, but for the ease of programming, this approach was of best interest.

Now that we have discusssed the implementation approach, we can now go on to discuss and compare performance of all these three implementations and see if we can conclude on which approach is better.

Performance calculations, Evaluation of results and Discussions

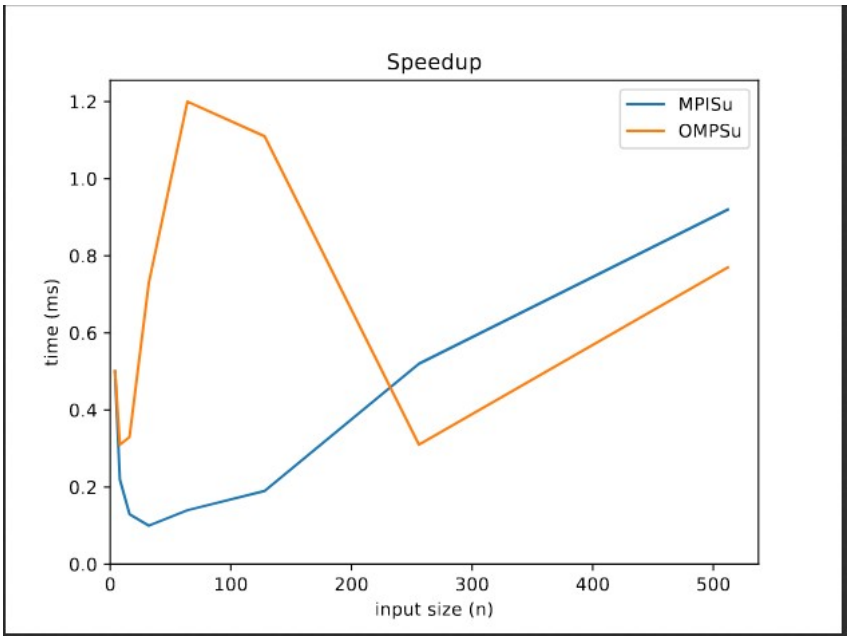
We've writen three algorithms for the bitonic sorter, 2 different parallel implementations and one serial implementation. We wish to receive input from the user, where the input is the size of the list and the the list itself. For the list to be sorted, it is required that the list is a bitonic sequence. There could be a misunderstanding in terms of whether the algorithm was supposed to receive a list of any form and then turn it into a bitonic sequence or it should just receive a bitonic sequence, I only came to a realisation after the algorithm was already implemented. For The purposes of scalability, scalability in terms of the length of the list. I have included a python script that receive a number *n* and returns a list of size 2^n in a bitonic sequence. This input can then be passed into the bitonic sorter as input on the terminal.

Below are tables of data we gathered.

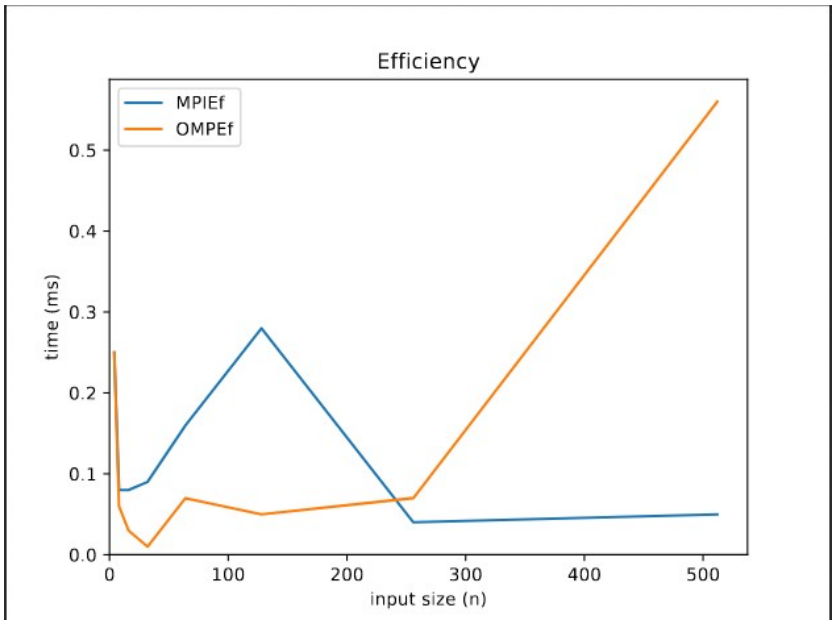
Size (n)	Serial	OpenMP – (num Proc)	Speedup	Efficiency
4	0.000002s	0.0000035s – (2)	0.5	0.25
8	0.000002s	0.0000065s – (4)	0.31	0.08
16	0.000002s	0.0000061s – (4)	0.33	0.08
32	0.000003s	0.0000041s – (8)	0.73	0.09
64	0.000004s	0.0000033s – (2)	1.2	0.6
128	0.000006s	0.0000054s – (4)	1.11	0.28
256	0.000012s	0.000039s – (8)	0.31	0.04
512	0.000023s	0.000030s – (16)	0.77	0.05

Table – serial time, OpenMP parallel time , speedup and efficiency

The implementation uproach is what brings about the poor perfomance of the system. This is due to storing more data like an extra array. With that most of the operations needed to wait for other processors to complete before the program can proceed. Since the implementation is pretty basic, memory access in the shared memory approach i.e. OpenMP, accessing memory requires one processor to wait for another if it needs to access memory. This is therefore computationally expensive. With MPI on the other hand, we do a lot of data broadcasting. We update 2 arrays at every iteration and broadcasting them, and then sending results back to the controlling processor. Bare in mind that the data being moved around isn't just integers but arrays and communication is therefore no minimized. The more the processors, the more communications, the more the computation cost as much time is spent in communication and not solving the problem itself. As you can see from the data above, the serial code is way better as compared to both parallel codes. We could change the entire implementation so that we are achieve better perfomance, yes we could, but due to time constraints am afraid we cannot. Below we compare between MPI and OpenMP.



Above is the graph comparing speedup and below is the graph comparing efficiency.



From the above results, we can safely conclude that using the general implementation, which shows to be not the best, the serial implementation is way better than both parallel implementations.

References and citations

Wikipedia Bitonic sort
https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Mullapudi-Spring-2014-CSE633.pdf
Reserch Gate website on Bitonic sort

(note: if accessing from ubuntu, ctrl + right click … opens the link)