Parallel Computing Assignment

Yaseen Haffejee

1827555

University of the Witwatersrand

Abstract

In this assignment we were required to parallelize two algorithms, Conway's Game of Life and Dijkstra's Algorithm respectively. For Conway's Game of Life we were required to use a distributed memory system in order to achieve a performance enhancement. For Dijkstra's algorithm we had to implement two approaches, using a shared memory system as well as a distributed memory system in order to achieve an increased performance.

Parallel Computing Assignment

Conway's Game of Life

Method of parallelization

In order to parallelize Conway's Game of Life using MPI we needed a way to break the board up into tiny sub-boards to ensure each process has an adequate piece of the board to compute. The approach I undertook in order to split the board up was simply to allocate a certain number of rows to each process. The number of rows that a process will work on are consecutive rows of the board and can be determined by dividing the number of rows of the board by the number of processes that are going to be utilised. This assumes that the number of processes being utilised is indeed a factor of the number of rows.
In order for each process to access the rows assigned to them , each process finds the index of their first row by multiplying their process id by the number of rows each process should have. The index of the final row of their computation is calculated by adding the number of rows each process should have to the starting index of the respective process. Each processes range of rows is not inclusive of the final row index that is calculated.
The next step was storing the actual board. Since communication is usually the factor that slows down MPI programs, in order to mitigate this , each process stores their own two dimensional board in order to ensure less communication amongst the processes. Since we run the program over a stipulated amount of iterations , after each iteration before any calculations can occur, each process updates their representation of the board to the new generation that was computed previously. One important note is the fact that the initial board is randomly generated and passed in as a parameter to the serial implementation and is generated prior to any parallel computation for the parallel version. Thereafter it is updated from generation to generation.
We now enter the computation heavy loop. In here, each process loops through their range of rows and determines whether or not the neighbours of a certain index should remain

alive, die or be reborn in the next generation. Once each process completes this computation, each process is storing certain rows of the next generation. Note that the results of the computation are stored in a vector to ensure the results are in contiguous memory blocks. We then communicate these results to each process using MPI_ALlgather. Note that each process will now have a one dimensional representation of the board. Each process will then convert the one dimensional representation into a two-dimensional representation and update the board since we utilise the board to compute the next generation. This alludes to the reasoning behind the Allgather function. It is imperative that it is utilised since each process needs to have the entire board for the next iteration. However, if we were to only compute one generation, we could use MPI_Gather instead. We continue this until the stipulated number of generations are computed.

Method of validation and testing

In order to validate and test that the results of the parallel implementation, we compare it to the results generated by the serial implementation. In the parallel code, before any parallel computations occur, we call the serial implementation. The serial implementation returns a vector containing one dimensional representations of each generation computed. For example, if we had five generations , the serial version returns a vector containing five vectors and each of those five vectors represent a new generation. The board at index 0 is the parent of the new generation of the board at index 1 and this continues for all. The serial implementation is timed from the beginning of the call and timing ends immediately after. We then store the serial time.

We place a barrier after the serial implementation so that all the other processes wait for process zero to compute the serial result. Once all the processes reach here, we begin timing the parallel implementation and begin with the parallel computation.

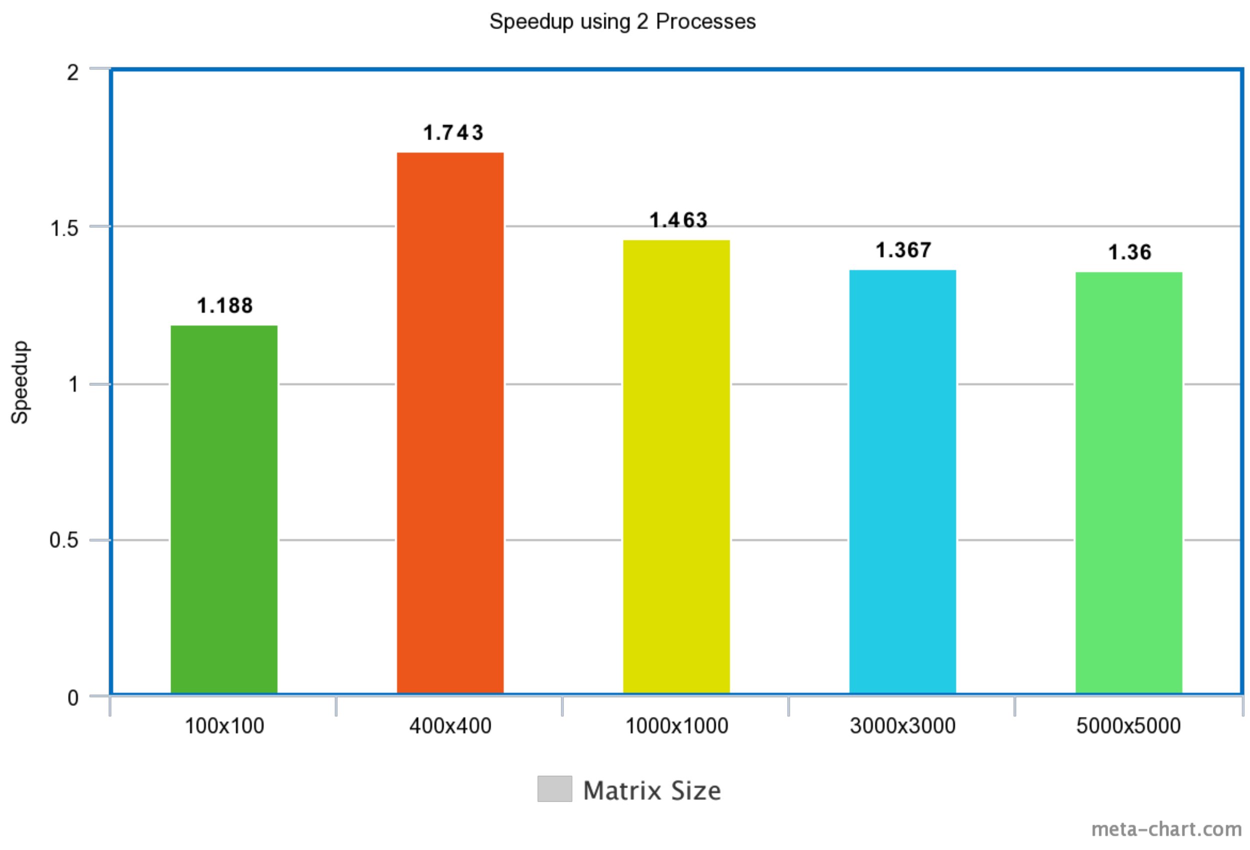
In order to compare results and not add unnecessary time that is not spent on computation to the parallel version , once each process has the one dimensional representation of the board, we let process zero add the result to a vector which will store one-dimensional representations of the board for the parallel version. This is done for each generation. Once all the generations are computed, we stop timing the parallel version and using MPI_Reduce to accumulate all the parallel times for each process and take the average. Thereafter, process 0 will loop and check that the parallel results and serial results are the same for each generation. This is simply comparing two one-dimensional vectors. If they are equal , it is stipulated and if the results are different , that too will be stipulated. Once all of this is complete, we simply output the size of the board, number of iterations which represents how many generations we computed, the number of processes used, the serial time , parallel time as well as the speed up.

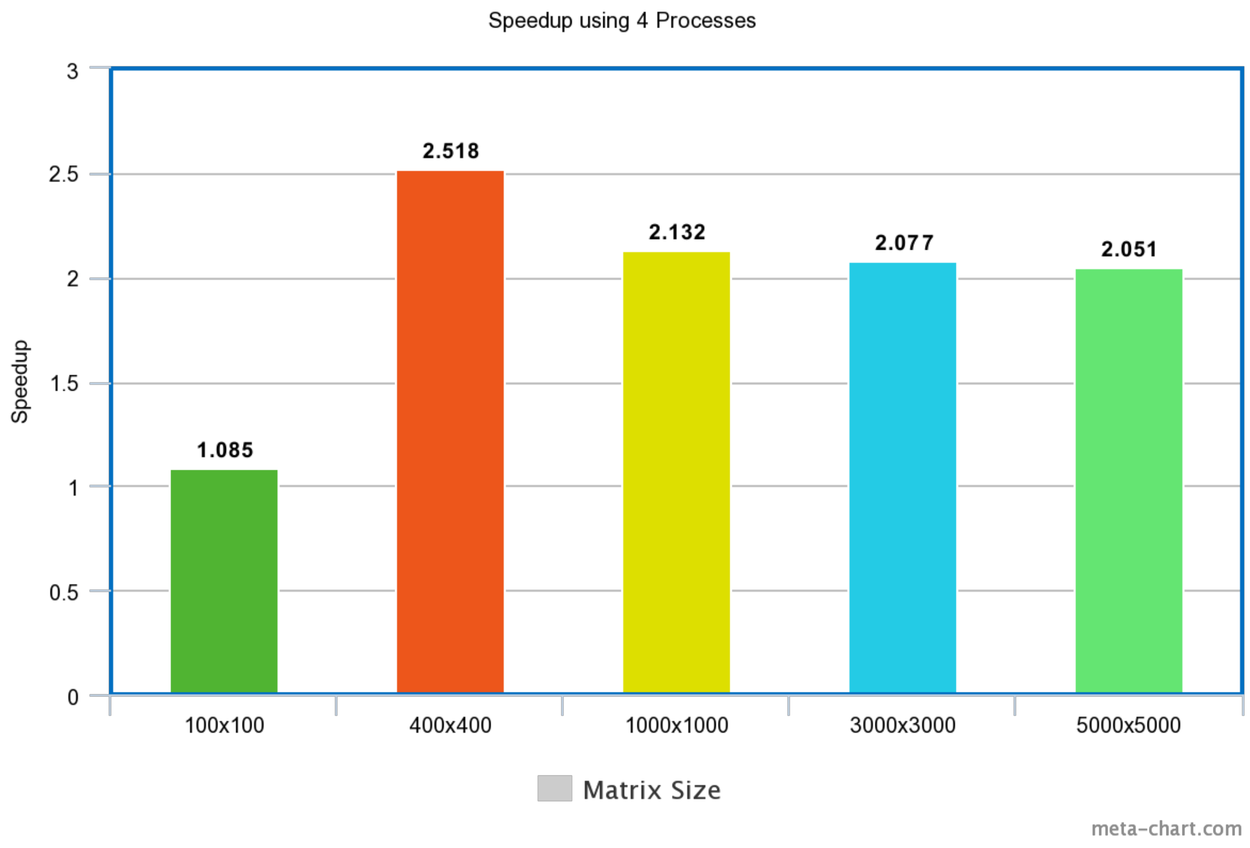
Results

Matrix Size	Number of Processes	Serial Time	MPI Time	Speedup
100x 100	2	0.00072	0.00061	1.188
100x 100	4	0.00086	0.00079	1.085
400x 400	2	0.0146	0.00838	1.743
400x 400	4	0.0155	0.00616	2.518
1000 x 1000	2	0.0759	0.0519	1.463
1000 x 1000	4	0.0759	0.0356	2.132
3000 x 3000	2	0.6600	0.4826	1.367
3000 x 3000	4	0.7145	0.3440	2.077
5000 x 5000	2	1.7689	1.3002	1.360
5000 x 5000	4	1.9709	0.9608	2.051

Table 1

Table showing the results of parallelzing various board sizes with various number of processes.


Speedup using 2 Processes
Speedup
2
1.5
1
0.5
0
1.188
1.743
1.463
1.367
1.36
100x100
400x400
1000x1000
3000x3000
5000x5000
Matrix Size
meta-chart.com



Discussion of Results

Considering the table above , we can clearly see that using 4 processes instead of 2 processes, always lead to a greater speedup for all matrix sizes excluding the 100x100 matrix. This is simply due to the fact that more of the heavy computation is distributed amongst more processes. However , if we increase the number of processes being utilised even further , we will not achieve a greater speedup. This is simply because , at some point the overhead of communication between these processes will simply outweigh the benefits of computation and consequently we will not achieve a speedup. This is simply a manifestation of Amdahl's Law. Looking at the 100x100 case , this is clearly a representation of Amdahl's Law since we did not increase the workload and simply increased the number of processes, hence we did not acquire a speedup. As mentioned earlier, this could be a result of the cost of the additional communication between processes.

However, if we were to increase the matrix sizes even further ,larger than the 5000x5000 matrix ,we could then increase the number of processes as well. In doing so , we would see that as the workload increased, being able to share the workload over more processes will result in a faster computation and ultimately a speedup.
From this we can conclude that even though parallelism will lead to a speedup , simply increasing the number of processes being utilised is not the solution to achieve a greater speedup. We need to experiment and find the optimal number of processes to utilise, this will enable us to achieve an optimal speedup without any wastage of resources.

Dijkstra's Algorithm

Method of parallelization OpenMP

In order to parallelize Dijkstra's algorithm using OpenMP, we split up the number of vertices in the graph amongst each thread. The number of vertices each thread will have is determined by dividing the total number of vertices by the number of threads the program will use. This assumes that the number of threads is a factor of the number of vertices. Each thread is then assigned $\frac{Number of vertices}{Number of threads}$ consecutive vertices that they will compute the distances for.

In order to ensure we avoid race conditions we need to declare certain variables as shared and others as private. The shared variables are the adjacency matrix of the graph, the visited array, the distances array, the global minimum distance and minimum vertex, the number of vertices and the offset which represents how many vertices each thread will have. Each thread computes their start vertex and end vertex , however the end vertex is not inclusive in the range of vertices each thread will work on. Initially the distance at the source vertex is set to 0

We begin by having an outer loop that will loop through each vertex. This ensures we find the minimum distance from every vertex to the source vertex. Within this loop we begin by having a single construct which is used to initialise the global variables which store the global minimum distance and global minimum vertex. Thereafter, each thread initialises their local minimum distance and local minimum vertex variables. Each thread then loops through their range of vertices and updates their local minimum distance and local minimum vertex variables if the vertex has not been visited and if the distance is less than what the current local minimum distance is. Once this loop is completed , we encounter a critical section. This ensures we avoid a race condition since within this section, each thread updates the global minimum distance and global minimum vertex variables if their local variables are smaller. Once the critical section is complete, the global variables now store the global minimum distance and which vertex it belongs to. Note that after the critical section we place a barrier since we required the global variables to updated completely prior to any thread continuing. Thereafter we allow thread 0 to mark the global minimum vertex as visited in the visited array. Since only process 0 does this , we place a barrier after it since each thread utilises the visited array and it needs to be updated before they use it or else it will yield incorrect results. Finally , each thread loops through each vertex in their range of vertices and will update the distances array if required. Once again , there is a barrier outside this loop since we need all threads to complete updating their respective parts of the array before continuing onto the next vertex since we utilise these distances. This process continues until all the vertices have been looped through.

Method of parallelization MPI

In order to parallelize Dijkstra's algorithm using MPI, we split up the number of vertices in the graph amongst each process. The number of vertices each thread will have is determined by dividing the total number of vertices by the number of processes the program will use. This assumes that the number of process is a factor of the number of vertices. Each thread is then assigned $\frac{Numberofvertices}{Numberofprocesses}$ consecutive vertices that they will compute the distances for.
Each process computes their start vertex and end vertex , however the end vertex is not inclusive in the range of vertices each thread will work on. Initially the distance at the source vertex is set to 0.
Each process will have their own local distances array as well as visited array. We then begin the computation intensive loop. We loop through each vertex. Within this loop, each process computes the vertex with the current minimum distance. Each process then marks their visited array at this vertex as true. Each process then loops through their range of vertices and updates their local distances array. Once this is complete , we have an MPI_Allreduce communication whereby we we reduce each processes local distances array into one using the MPI_MIN operation. This ensures that the result of MPI_Allreduce

contains the minimum value at each index.For example, or index 0 of the result, it will take all the incoming distances arrays, look at the first element and take the smallest from them all. This is done for all the elements. Once this is complete , we move onto the next vertex and continue this until we have looped through each vertex.

Method of validation and testing

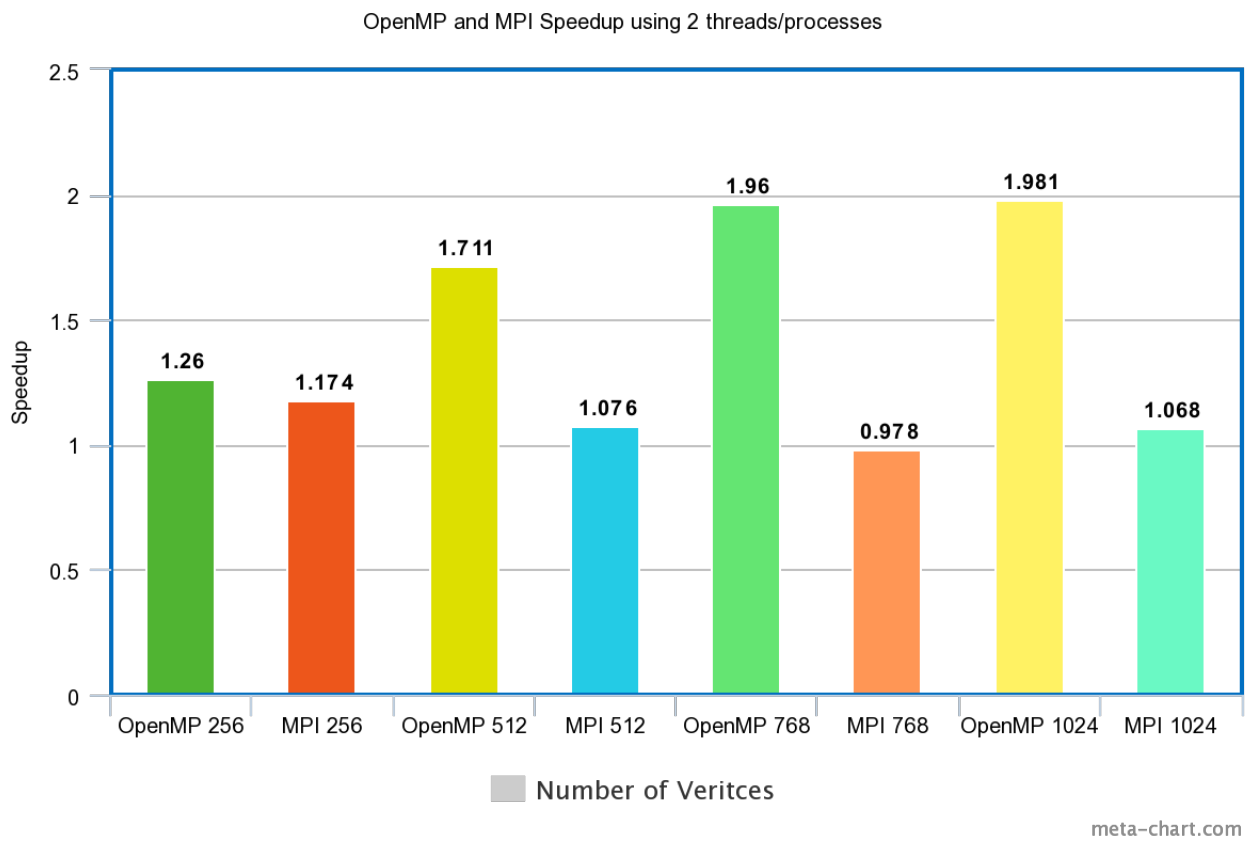
In order to validate and test that the results of the parallel implementation are correct , we simply run the serial program and store the distance vector that is returned. We also time this program from before we call the serial implementation to immediately after. We then call the parallel implementation of OpenMP which will also return a distance vector. We compare the two vectors and we compare the result of the MPI implementation and if they are equal then we are certain that the parallel implementations are correct. We also time the parallel implementation from the before we call it and stop timing it immediately when it returns.However, we take the parallel time as the average time of each process for the MPI program.If the results are equal , we output the time taken for the serial and parallel implementations and the speed up. We also output the number of vertices in the graph, the number of threads used and source vertex.

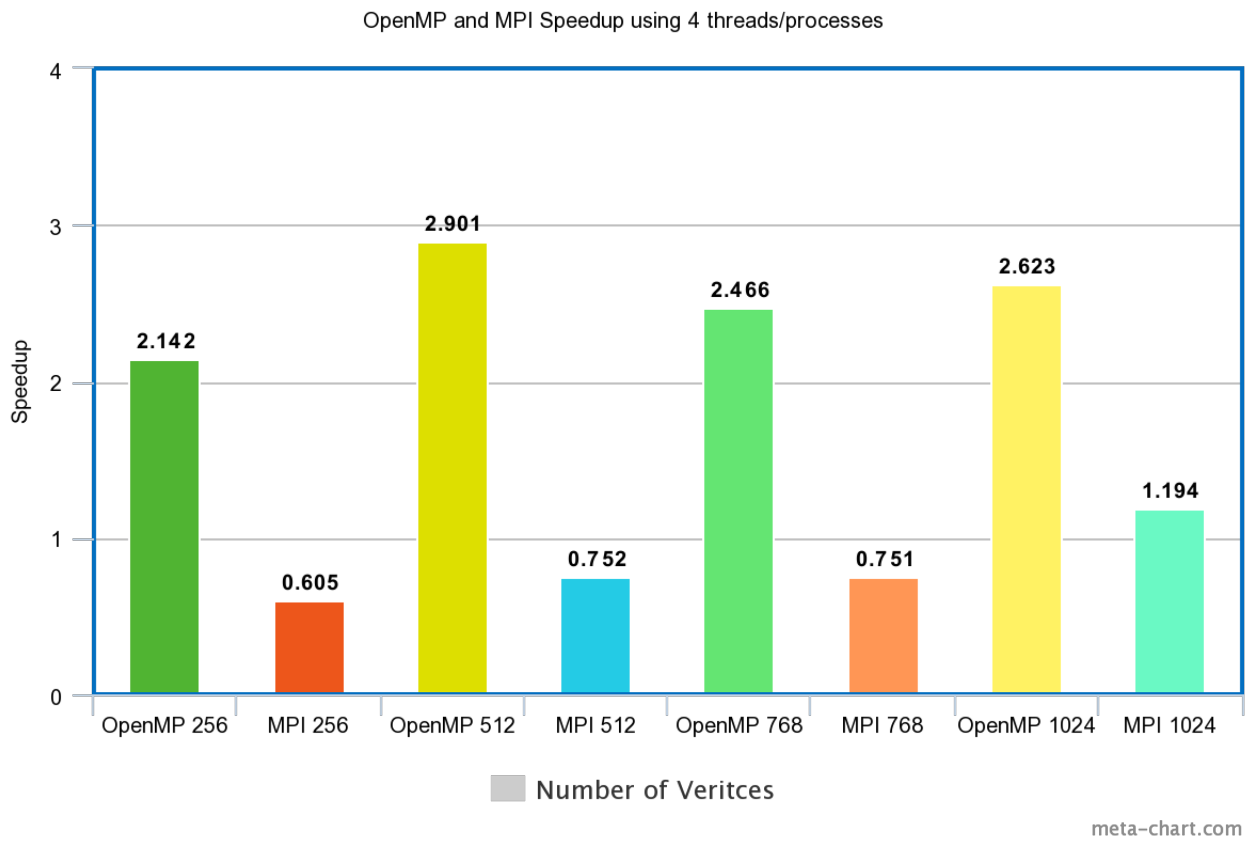
Results

Number of Vertices	Number of Processes	Serial Time	OpenMP Time	MPI Time	OpenMP Speedup	MPI Speedup
256	2	0.0256	0.00203	0.00218	1.260	1.174
256	4	0.0037	0.00174	0.00615	2.142	0.605
512	2	0.0124	0.00724	0.01152	1.711	1.076
512	4	0.0152	0.00526	0.02031	2.901	0.752
768	2	0.0250	0.01276	0.02556	1.960	0.978
768	4	0.0244	0.00989	0.03248	2.466	0.751
1024	2	0.0421	0.02126	0.03945	1.981	1.068
1024	4	0.0436	0.01661	0.03649	2.623	1.194

Table 2

Table showing the results of parallelzing various graph sizes with various number of processes.


OpenMP and MPI Speedup using 2 threads/processes
Speedup
2.5
2
1.5
1
0.5
0
1.26
1.174
1.711
1.076
1.96
0.978
1.981
1.068
OpenMP 256
MPI 256
OpenMP 512
MPI 512
OpenMP 768
MPI 768
OpenMP 1024
MPI 1024
Number of Veritces
meta-chart.com



Discussion of Results

Considering the table above, we can see that the OpenMP implementation achieves a greater speedup than the MPI implementation. One major reason for this is the communication overhead that is coupled with the MPI implementation. Distributed memory systems require consistent communication for shared data, whereas shared memory systems can access shred data utilising shared variables. Taking this into account, updating a shared variable using a single thread is much faster than updating a shared variable using a single process and thereafter communicating the results to the other processes.

For the OpenMP implementation we can see that we achieve a greater speedup using 4 threads instead of 2. This is simply because we are distributing a larger workload over more processes. Consequently we are parallelising a greater portion of the workload which ultimately leads to a larger speedup. However, if we were to increase the number of threads further without increasing the workload, we would not achieve a greater speedup. This is

due to the fact that eventually the overhead of all the barrier constructs will begin to dominate the time, as well as the overhead of creating these threads. Since each thread will have workloads that are too small, consequently the speedup will decrease drastically.
If we isolate the MPI speedup, we can clearly see that using 2 processes enabled us to achieve a greater speedup than using 4 processes. This alludes to the cost of communication in the implementation. This blatantly tells us that the communication time is greater than the computation time. Consequently, we achieve a lesser speedup. Thus we can conclude that in order to achieve a larger speedup , we need to minimise the amount of communication that is required across the processes.
We can deduce that parallelism does give us an advantage. However , we need to be sure to optimise our parallel implementations , if we want to achieve a speedup. Taking into account my two implementations, i would choose the OpenMP implementation. However , if we were able to reduce the amount of communication in the MPI implementation, the speedup achieved could be similar or even greater than that of the OpenMP implementation.Thus, we need to always optimise our implementations and ensure that we are using an appropriate amount of processes such that we are achieving an optimal speedup as well as being optimal in our resource consumption.