Parallel Implementation of Fast Fourier Transform

Ibrahim Salat

(Dated: December 12, 2022)

INTRODUCTION

The introduction of supercomputers in the 1960's paved the way for high performance computing (HPC), that allows the processing of data and complex algorithms at high speeds. A modern supercomputer can perform over one quadrillion floating point operations (FLOP) in a second, which is a million times more than a standard 3GHz processor laptop. A supercomputer has multiple compute nodes in a network that work using parallel processing to compute a task. This is similar to multiple desktop computers working connected together working on the same task. Parallelisation or parallel processing is a process of dividing the computational workload over multiple threads or processes to reduce the time it takes to execute a code. This can be achieved using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) which takes advantage of distributed memory structure of the supercomputer and the multi-cpu processor to parallelise the code.

Fast Fourier Transform (FFT) algorithms are a class of efficient algorithms for computing the discrete Fourier transform (DFT) and its inverse. The Fourier transform is a mathematical tool used to decompose a signal into its constituent frequencies, allowing for the analysis and manipulation of the signal in the frequency domain. The FFT is important because it allows for the efficient calculation of the DFT, which would otherwise be computationally intensive. This makes the FFT a key tool in a wide range of applications, including signal processing, image, video processing, and data analysis.

This paper attempts to simulate the Fast Fourier Transform algoritm on the university of Bristol HPC, BlueCrystal4 by parallelising it using the distributed memory approach and multiprocessing method. This paper goes on to compare the data between the different approaches and come up with improvements

THEORY

Description of the FFT

The fast fourier transform is a computationally efficient way to compute the Discete Fourier transform. Just like the continuous fourier transform, the DFT converts a time-domain signal into a frequency domain signal, but for a discrete number of sample points of the signal, N. The DFT of a signal is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{kn} \tag{1}$$

where $k = 1, 2, ..., N-1$ and $W_N = e^{-j2\pi/N}$ Simulating equation 1 would be very computationally expensive, as the flop count of the DFT algorithm would scale up in the order of N^2. The FFT algorithm computes the same task in much more efficient manner, as the FLOP count of the FFT algorithm scales up with $(N/2)log_2 N$. At $N = 1024$ the FFT is already 204 times faster than the DFT.

The FFt algorithm implemented in this paper is the one formulated by Cooley and Turkey in the 1960s. It takes advantage of the symetry of the fourier transform to divide the DFT into smaller DFTs. Equation 1 can be written as sum of 2, $N/2$ - point DFTs:

$$X(k) = \underbrace{\sum_{n=0}^{(N/2)-1} x(2n) W_{N/2}^{kn}}_{N/2\text{-poin DFT}} + W_N^k \underbrace{\sum_{n=0}^{(N/2)-1} x(2n+1) W_{N/2}^{kn}}_{N/2\text{-point DFT}} \tag{2}$$

where the first sum is the DFT of the even indexes of the sample points and the second sum is the DFT of the odd indexes of the sample data points. Each DFT needs to be calculated only from k = 0 to N/2-1 due to the periodicity of $(W_(N/2))^k$ over N/2. This reduces the number of calculation needed to be made to perform the complete DFT. This process can be repeated on each of the 2 DFTs to give 4, N/4 point DFTs.This Divide and conquer strategy is applied until all the DFTs are just a 2 point DFTs, eventually giving the FFT algorithm. Figure 1 describes the algorithms using butterfly diagrams with N=8.

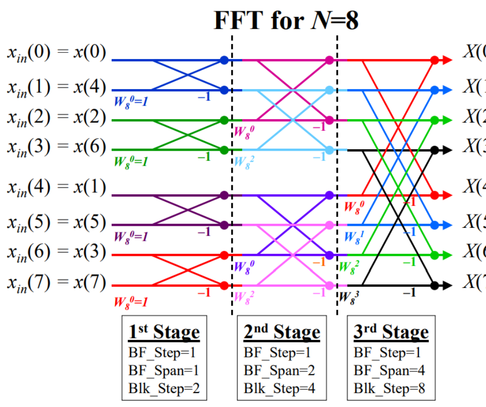

FIG. 1. Butterfly diagram for the FFT algorithm with 8 sample points. Taken from [1]

The diagram is divided into stages, with each stage having N/2 butterflies. The configuration of butterflies in each stage is different. The output of one stage is the input of the next stage. The order of the input in the first stage is a result of the repeated divide and conquer strategy which is calculated using a bit reversal algorithm that is performed before the butterfly calculation in the above diagram. Another important characteristic of the diagram is the number of blocks in every stage. The first stage can be thought of as having N/2 blocks with 1 butterfly per block, where as the second stage has 2 blocks with 2 butterfly per block. It is important to note that each stage has N/2 butterflies. A single butterfly operation can be understood using the first butterfly of stage one. To get the first output, x(4) is added to x(0), and to get the 2nd output, x(4) is multiplied by the twiddle factor, $(W_8)^0$ and subtracted from x(0). The same operations are performed for the rest of the butterflies. 2 more variable derived from the diagram in figure 1, which are used in the FFT algorithm in this paper are the block step and butterfly span. Block step is the index distance between consecutive blocks in a stage and the butterfly span is the index distance been the indexes in a butterfly. Note that all the properties of the diagram are a function of the stage which depends on the number of sample data points. Thus the relation between all the variables is given in the table below, which are used in coding the FFT algorithm. —

	k^{th} Stage (Origin-0)
Num Blocks	$N/2^{(k+1)}$
BFs/Block	2^k
BF Span	2^k
Block Step	$2^{(k+1)}$
BF Step	1

(3)

Shared Memory

Modern computer architecture consists of multi-core CPUs in which all cores share and have access to the same memory. APIs such as `OpenMP` take advantage of such an architecture to introduce multi-threaded parallelism into an algorithm. OpenMP uses a fork-join model of parallelism. The code starts at the `master-thread` and creates the instructed ammount of threads when it hits the parallel region of the code. The portion of the code instructed to be parallelised, runs in parallel until every thread finishes their work, after which the threads synchronize and join into one single master thread. The synchronization barriers can end up being a bottleneck if the loading of threads is imbalanced. OpenMP provides the dynamic Scheduling option to control the chunk size each thread gets to balance the load on each thread.

OpenMP's advantage lies in the fact that it requires almost no change to the serial code to implement threaded parallelisation. A call to a compiler directive, a run time library and environment varialble such as `num_of_threads` is all that is added to implement parallelisation. since all cores share the memory there is no need for communication between threads, giving it an advantage over distributed memory approach. Although there is no communication overhead when using OpenMP, there is a wait time over head, as only one thread can access the memory at once.

OpenMP is very easy to implement in a code but would be limited by the number of cores available on the system, therefore it would be expected to run the FFT algorithm faster for smaller problem size as MPI would have communication overheads acting as a bottle neck.

Distributed Memory approach

HPCs like BlueCrystal4 are made up of multiple compute nodes that each have multi core CPUs. Each node can be considered as an individual computer, and utilizing multiple computer (nodes) to perform the same task can give even greater speedups. MPI (Message Passing Interface) is an API that enables communications between multiple compute nodes/CPU. Each compute node is assigned a rank number used to identify the rank. Any code written in MPI will be executed on each node unless specified using the rank number. In theory, each rank can perform different types calculations on each rank, but programming such a algorithm will be difficult and often not an efficient one. Usually, a master rank distributes the data to other compute nodes that all perform the same type of calculation on different sets of data. The master then collects the results from all the compute nodes. To understand how to divide the input data to be given to all the compute nodes, a deep understanding of the algorithm is needed

There are multiple ways the FFT algorithm can be parallelised using MPI [2]. The 2 ways considered during this project were , Parallel over butterflies and Parallel over blocks. The Parallel over butterfly method requires sending each core equal number of butterflies. The master would have to identify the pair of data points for every butterfly operation and send the data specifically to the appropriate rank. The Parallel over blocks method sends an equal number of blocks to every rank. Since each block contains an integer number of butterflies, there is no communication needed between each block and therefore each worker rank. The downfall for this method is that the level of parallelisation decreases with each stage. For the aim of this project,a modified version of Parallel over blocks method was used, as the parallelisation would only start to decrease when the number of blocks in a stage is equal to p, the number of compute nodes. when $N \gg p$ this would only happen in the final stages of the algorithm. The overhead in this method would therefore be less than the communication overhead and the data arrangement overhead in each stage of the algorithm in the parallel over butterfly method.

The master divides the input array into equal chunks to be distributed to each worker and keeps the fist chunk with itself. This is done by using `MPI_Scatter` function. Each node performs the FFT algorithm on the data it has and then the `MPI_Gather` function is used to collect all the results into the master. This is performed in a loop from stage 0 to the stage where the number of block become equal to the number of processors, p. Then all the data is sent back to the master which performs the rest of the FFT algorithm in serial and gives the output in the correct order.

DATA ANLYSIS

3 different FFT algorithm implementations were written in C++, the serial, shared memory approach (OpenMP) and distributed memory approach (MPI). Each was run to measure time using the `chrono_high_resolution_clock` or the in built in wall clock timer in MPI and OpenMP, to find the effects of increasing the number of processes, p, or scaling the problem by increasing the number of data points, N. The runs were performed on BlueCrystal Phase 4 supercomputer [3], which consists of 512 compute nodes each consisting of 2, 14 core 2.4 GHz Intel E5-2680 v4 (Broadwell) CPUs, and 128 GiB of RAM. Running long jobs on BlueCrystal have an element of error in the measured time due to the randomness of the cores selected to run the job, which could happen to be physically further away from each other. Other running jobs, or ghost jobs could be sharing the same core, therefore sharing the same resources and affecting the time measured to complete the job. Due to such randomness, the error is calculated as the standard error in the measured mean of time over 3 runs although all measurements were done 4 times. . The first set of runs always had higher measured time then the rest of the subsequent runs.Using the `EXCLUIVE` flag in slurm seem to have deduced the error because of this this. It is also likely that the subsequent jobs can take advantage of the resources in the cached memory from the first run. Therefore only the last 3 runs are considered in the plots in this paper.

Comparing Serial, OpenMP and MPI

Figure 2 compare the performance of serial, shared memory approach and distributed memory approach, as the problem size scales. For relatively small problem size it is expected that the serial code be the fastest as the time saved in parallelising the code is not less than the time spent in communication overheads in MPI. The shared memory is also slower than serial in this range as serial code can take advantage of the individual core cache memory to store data which is faster than using a shared memory accross all the cores. All core trying to access the data at the same time also contributes to the increase in run time.

Figure 2 shows the time taken to run the algorithm in a logarithmic scale, measured from $N = 2^{15}$ to 2^{26}. The OpenMP is runs use 25 threads and the MPI has 32 workers. These were chosen as these were the number of threads/workers that performed the algorithm the quickest. The expected trend can be seen on the graph. At lower problem size the MPI is the slowest as communications in MPI take much longer than the overheads in OpenMP. The logarithmic scale is chosen as it clearly

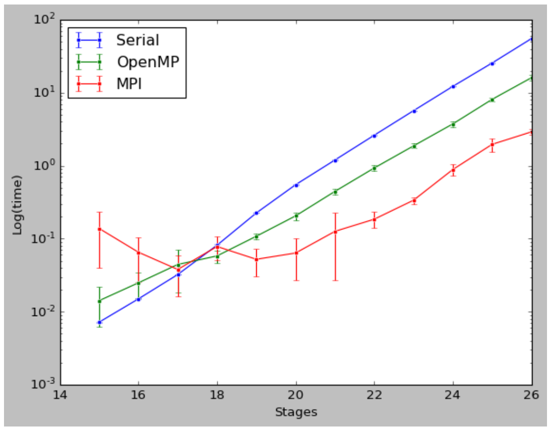

FIG. 2. Comparing the performance of each approach over a range of problem size. The OpenMP is performed with 25 threads and MPI with 32 processes. The scale used is a log scale.

shows at what size the parallelisation starts to payoff. This happens for both, MPI and OpenMP, between $N = 2^{17}$ and 2^{18}. Once the parallel compute time is greater than the communication time in MPI, the distributed approach remains the fastest as the problem scales. The OpenMP is slower at larger problem size is because more cores try to access the shared memory, which results in threads waiting to access data. this waiting time scales with the problem size and so OpenMP is slower. It is important to note that at lower problem size the MPI time has relatively large error bars as communication time is a large percentage of the measured time, which contributes to most of the error as the CPUs in a node are chosen at random based on availability.

Speed up and Efficiency

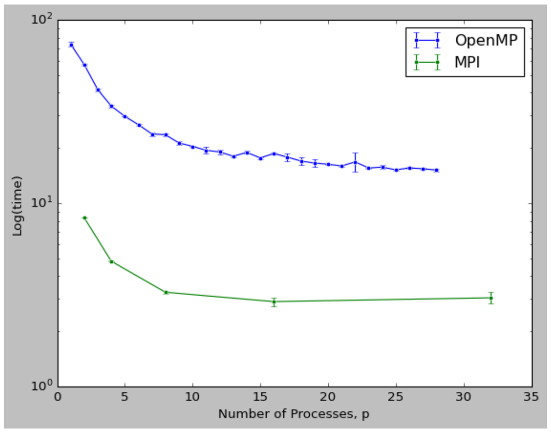

FIG. 3. Plot compares the performance increases with an increase in processes/threads using the 2 approach to parallelisation. The openMP uses static scheduling.

Figure 3 plots the time taken to run the simulation with increasing number of threads/processes. It is clear that as p increases, the time measured decreases. This is an expected behaviour as more parallelisation would result in lower time spent at high enough problem size. All the runs in figure 3 and figure 4 are done with N=2^{26}. Figure 4 plots the speed up and the efficiency over a range of processes. Speed up is the a measurement of improvement in time at process, p_n, compared to the speed at the first process. Then efficiency can be thought of, as the contribution of each core to the speedup at that process. Speed up, S and efficiency, E are given by:

$$S = T_1/T_n n \tag{4}$$

$$E = S_n/n \tag{5}$$

The MPI code is almost 10 times faster than OpenMP at every process, but after 8 processes it becomes less efficient and shows very little speed up. This is because the part of the code that is parallelised by the MPI code takes less time than the part of the code where arrays are initialized. Therefore increasing the processes will reduce the parallel computer part of the code but the over all time will not decrease as the communication time and the setup time of the code are unaffected by the number of processes. Therefore it can be concluded that there is a max number of process for each problem size that will show a speed up.

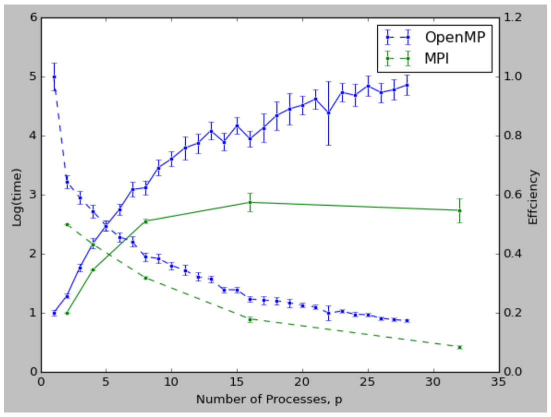

FIG. 4. The plot shows how the speed up and efficiency varies with an increse in processes. The solid line is the speedup and dashed line is the efficiency.

Scheduling

OpenMP provides scheduling which gives control over how much data each thread gets every iteration. This can be used to reduce the wait time of threads balance out the load over the threads. Figure 4 shows the effects of implementing dynamic scheduling into the OpenMP code. Dynamic scheduling does not wait for all the threads to finish their task but each thread asks for a chunk when done with its chunk. So the distribution of data is decided during run-time. Dynamic scheduling is usually a good choice for algorithms that have asymmetric computational load. From figure 4 it can be seen that as the chunk size increases the speed up gain decreases. For the FFT algorithm it turns out that static scheduling is the most optimum scheduling as most of the calculations are similar in nature.

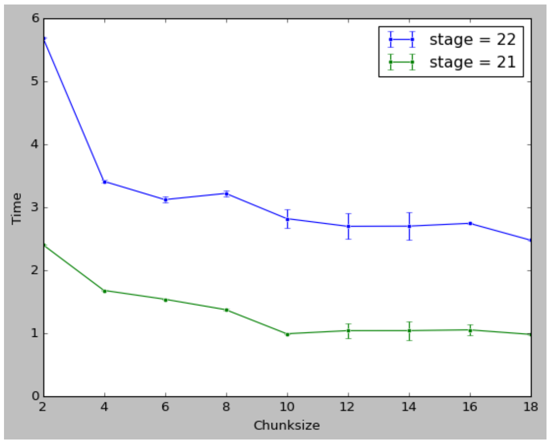

FIG. 5. Plots the decrease in run time for different chunk sizes using dynamic scheduling in OpenMP.

FUTURE WORK

In OpenMP, another type of scheduling provided is guided, which is simular to dynamic in the sense that it distributes the load at run time but also reduces the chunk size as the program progresses to reduce inefficiencies introduced by large chunk size.

The load balancing on the MPI code is not optimal, as the master code does most of the work. This is due to the fact that the algorithm was divided such that each core get an integer amount of blocks, but this is not possible when the number of blocks per stage is less than the number of processors. To only get a certain number of workers to perform a calculation, `MPI_COMM` can be used to create a subset of communicators, which would then allow the last stages of the algorithm to be parallelised.

The MPI code is also very memory inefficient. This reduces the flexibility of increasing the problem size greater that $N = 2^26$. This is largely because the way the code handles complex numbers in separate arrays of `Doubles`. Instead using `STD::COMPLEX` would have reduced the memory required to run the code. This would enable investigation into larger problem sizes. A combined implementation of MPI and OpenMP code was also written but since limited by the maximum problem size no effects of the the combined code was seen on the measured time. Making the code more memory efficient would also enable the possibility of combine shared and distributed memory approach.

It is also worth mentioning that there are other types of FFT implementation that are faster than the algorithm implemented here, which are worth looking into [4].

CONCLUSIONS

This paper has been successful in showing that for an FFT algorithm parallelisation techniques makes the algorithm faster. The MPI code is always almost 10 times faster than the OpenMP code. It can also be concluded that static scheduling in OpenMP works the best for this algorithm. Some techniques have been mentioned that would make the implementations more memory efficient and enable better investigations.

REFERENCES

1. [Online; accessed 12. Dec. 2022]. June 2015. `http://www.ws.binghamton.edu/Fowler/Fowler%20Personal%20Page/EE302_files/FFT%20Reading%20Material.pdf`.
2. Meiyappan, S. Implementation and performance evaluation of parallel FFT algorithms. *ResearchGate.* `https://www.researchgate.net/publication/228991757_Implementation_and_performance_evaluation_of_parallel_FFT_algorithms` (Dec. 2022).
3. *BlueCrystal Phase 4 - ACRC, University of Bristol* [Online; accessed 12. Dec. 2022]. Mar. 2018. `https://www.acrc.bris.ac.uk/acrc/phase4.htm`.
4. [Online; accessed 12. Dec. 2022]. June 2016. `https://www.tamps.cinvestav.mx/~wgomez/material/AID/fft_algorithms.pdf`.