The fast and the Fourious: A study of parallel Fast Fourier Transform Algorithms

Khalil Pierre
Department of Physics, University of Bristol
(Dated: July 2021)

In this project multiple 1D FFT algorithms were built and tested using both serial and parallel paradigms. For the serial FFTs a recursive, vectorised and iterative FFT algorithms were built. The cythonized iterative FFT algorithm performed the best, calculating the DFT of a $N = 2^{22}$ signal array in 1.30 ± 0.02 seconds. The recursive and vectorised FFT had timings of 21.96 ± 0.04 and 2.89 ± 0.01 seconds respectively for the same size signal array. Both shared and distributed memory parallel FFT algorithms were built using the OpenMP and MPI packages. The parallel algorithms were tested and timed on BlueCrystal phase 3. The shared memory FFT algorithm achieve a maximum speed up of 2 using 8 threads for a $N = 2^{25}$ size array. However, for lower size arrays the speed up was less significant. With the shared memory FFT performing worse then the serial iterative FFT for arrays sizes less then $N = 2^{12}$. Two distributed memory FFT algorithms were built using cyclic block mapping and static block mapping. Both were slower then the serial iterative FFT algorithm for array sizes up to N^{25}, with the cyclic block mapping algorithm performing worse then the static block mapping.

INTRODUCTION

There are few mathematical tools as powerful or as widely used as the Fourier transform. Fourier transforms convert a function from its original domain, into it's representation in the corresponding frequency domain. In the realm of mathematics, Fourier transforms are taken over an infinite range and sample every infinitesimal point. Computers however are limited physically and cannot perform such an operation. The numerical counterpart of the Fourier transform is the discrete Fourier transform (DFT), which samples a function at evenly spaced intervals and returns an equally sized discrete Fourier transform. The DFT has many applications from signal analysis to file compression. Most importantly to physicist, the DFT is a useful tool for approximating solutions to partial differential equations [1]. In this report an example of the DFT being used to calculate the diffraction pattern of monochromatic light passing through a single slit apparatus will be shown.

The naive implementation of the DFT results in a complexity of $\mathcal{O}(N^2)$ [3, 4, 6] where N is the size of the input signal. This scaling can be problematic for large N. For example audio files are commonly sampled at 44kHz, for a 10 second audio recording performing the naive DFT requires approximately 10^{10} operations. The fast Fourier transform (FFT) algorithm is able to reducing the complexity of calculating the DFT to $\mathcal{O}(N \log N)$ [3, 4, 6]. For the same audio file the FFT would require 10^6 operations, an impressive reduction.

The modern FFT algorithm was popularized by James Cooley and John Tukey in 1965 however a version of the FFT was know to Carl Friedrich Gauss in 1805 [5, 9]. Today the FFT has many different forms and is ubiquitous in computing, being named one of the top 10 algorithms of the 20th century by the IEEE magazine [7]. For a comprehensive list of the different FFT algorithms that exist today see [4, 6].

This project investigated different versions of the FFT and how computing the FFT on distributed and shared memory systems effects the speed of the FFT.

BACKGROUND THEORY

For an equally sampled signal the DFT is given by

$$F_r = \sum_{l=0}^{N-1} x_l \omega_N^{rl}, \tag{1}$$

where F_r is the rth element of the frequency spectrum, x_l is the lth element of the signal array, N is the size of the signal array, and ω_N is the Nth primitive root of unity defined as

$$\omega_N = e^{\frac{2\pi i}{N}}. \tag{2}$$

It is immediately evident that equation 1 can be written in matrix form as shown bellow for the N = 4 case

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}. \tag{3}$$

The naive approach to calculating the DFT is to simply perform the matrix calculation above. This results in the $\mathcal{O}(n^2)$ complexity. The FFT exploits the symmetries of the ω_N^{rl} values (commonly refereed to as twiddle factors) to reduce the complexity to $\mathcal{O}(n \log n)$.

One important symmetry of the twiddle factors is the positive negative paired symmetry, $\omega_N^{j+N/2} = -\omega_N^j$ of the twiddle factors. This is most easily seen by plotting the twiddle factors in complex space as shown in figure 1. This symmetry allows the problem to be reduced into two half size sub problems. The DFT can then be performed on each sub-problem, requiring $(N/2)^2$ operations, the results can then be combined to attain the full length DFT. Each half size sub problem can be divided by another factor of 2. This process can be done recursively to reduce the number of operations. For an array of size 2^n repeating this process an operational cost of

$\mathcal{O}(n \log n)$ can be achieved. This is a general outline of the divide and conquer paradigm that the FFT employs [3, 4, 6]. The FFT can still be performed on arrays that are not size 2^n, by padding the array with zeros until it is a N = 2^n array without any information loss.

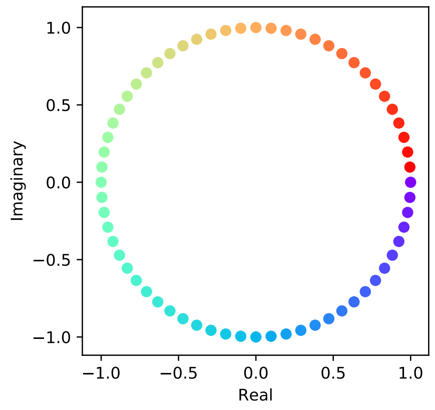

FIG. 1: Argand diagram of twiddle factors for N=64. A clear symmetry appears when the roots of unity are plotted.

Whilst many different versions of the FFT exists they generally all follow the same divide and conquer paradigm. The problem is divided into two or more sub problems of smaller size. These sub-problems are then solved recursively by applying the same algorithm. Boundary conditions are used to terminate the recursion once the sub-problems reach a certain size. And finally the sub problems are combined to give the solution.

In this project the radix 2 decimation in frequency (DIF) FFT algorithm was the central focus. A DIF FFT algorithms divides the N data point into the first half of N/2 data points and the second half of N/2 data points[6]. To derive the radix 2 DIF FFT Equation 1 can be rewritten as

$$\begin{aligned} F_r &= \sum_{l=0}^{N/2-1} x_l \omega_N^{rl} + \sum_{l=N/2}^{N-1} x_l \omega_N^{rl}, \\ &= \sum_{l=0}^{N/2-1} x_l \omega_N^{rl} + \sum_{l=0}^{N/2-1} x_{l+N/2} \omega_N^{r(l+N/2)}, \\ &= \sum_{l=0}^{N/2-1} \left(x_l + x_{l+N/2} \omega_N^{rN/2} \right) \omega_N^{rl}. \end{aligned} \tag{4}$$

Upon close inspection it becomes apparent that $\omega_N^{rN/2}$ is equal to one for even r and -1 for odd r. This means there will be an even and odd solution as shown bellow

$$F_{2k} = \sum_{l=0}^{N/2-1} \left(x_l + x_{l+N/2} \omega_N^{kN} \right) \omega_N^{2kl}, \tag{5}$$

$$F_{2k+1} = \sum_{l=0}^{N/2-1} \left(x_l + x_{l+N/2} \omega_N^{(2k+1)N/2} \right) \omega_N^{(2k+1)l}. \tag{6}$$

Using the following identities

$$\omega_N^{N/2} = -1; \quad \omega_{N/2} = \omega_N^2; \quad \omega_N^N = 1, \tag{7}$$

equations 5 and 6 can be rewritten as

$$F_{2k} = \sum_{l=0}^{N/2-1} \left(x_l + x_{l+N/2} \right) \omega_{N/2}^{kl}, \tag{8}$$

$$F_{2k+1} = \sum_{l=0}^{N/2-1} \left(\left(x_l - x_{l+N/2} \right) \omega_N^{lN} \right) \omega_{N/2}^{kl}, \tag{9}$$

respectively. Defining $Y_k = F_{2k}$ and $y_l = x_l + x_{l+N/2}$ for the even solutions and $Z_k = F_{2k+1}$ and $z_l = \left(x_l - x_{l+N/2} \right) \omega_N^l$, equations 8 and 9 become

$$Y_k = \sum_{l=0}^{N/2-1} z_l \omega_{N/2}^{kl}, \tag{10}$$

$$Z_k = \sum_{l=0}^{N/2-1} z_l \omega_{N/2}^{kl}. \tag{11}$$

The DFT has been reduced to two half size DFTs, which can of course be further subdivided. The data flow for any sub problem can be represented graphically by a Gentleman-Sande butterfly as seen in figure 2. Finally a subtle but important point is the order of the terms in the output array. For a DIF algorithm a bit reversal permutation is required for the output array elements to match the input array elements [3–6]. The DIT FFT algorithm can be derived in a similar way by splitting the input array into even and odd index terms in the first step.

METHOD

Sequential FFT algorithms

In this project three different sequential FFT algorithms were built and tested; a recurrent, a vectorised and an iterative FFT. The fastest of the three the cythonized iterative FFT algorithm was then selected to be paralised. The recursive and vectorised FFT algorithms are DIT FFTs whereas the iterative algorithm is a DIF FFT.

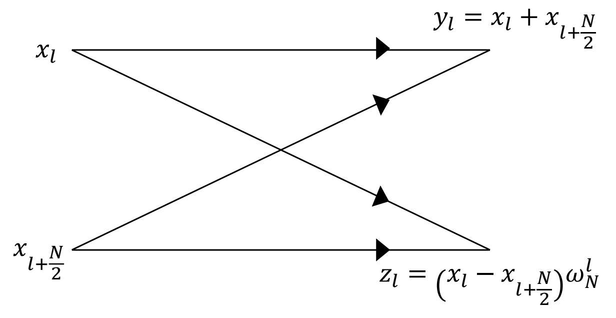
x_l
$x_{l+\frac{N}{2}}$
$y_l = x_l + x_{l+\frac{N}{2}}$
$z_l = \left(x_l - x_{l+\frac{N}{2}}\right)\omega_N^l$

FIG. 2: The Gentleman-Sande butterfly.

The simplest FFT algorithm was the recurrent FFT algorithm. The input array is split into even and odd index elements. The recursive FFT function is then called on the two sub problems. This recursive loop continues until the boundary condition, the sub problem size is meet. When the boundary condition is meet a DFT is performed and the result is returned. On the next level up the DFT of the even and odd arrays are multiplied by the twiddle factors and combined according to the DIT FFT equation. The resulting array is then returned. This process is repeated for every layer as the recursive function is exited. A flow chart of this process can be seen in figure 3.

The vectorised FFT algorithm works by taking advantage of the vector properties of numpy arrays. The input array is reshaped from a (N,1) array to a (2, N/2) array where the l^{th} element is now aligned with the (l + N/2) element. A N=2 DFT can then be performed on each column by using the numpy dot product function and the 2x2 twiddle factor matrix. The resulting array is then reshaped again forming a (4, N/4) array. The array is split into even and odd terms and combined according to the DIT FFT equation. Because of the vector properties of numpy arrays this stage is as simple as adding two array together with the odd array being multiplied by a twiddle factor array. This process is repeated until a (N,1) array is reformed. This array is then returned.

Finally the iterative FFT algorithm. In the previous section the DIF FFT was described in mathematical detail. The graphical representation of the DIF FFT can be built upon and extended. The data flow of the iterative FFT algorithm for an input array of size N = 8 is shown in figure 4. The problem is broken into multiple stages where in each stage the sub problem size is half that of the previous stage. By using the cython for loop the FFT can be performed efficiently using 3 nested loops. The first loop iterates over the different stages. For each stage the sub problem size and the number of problems are calculated. The second loop iterates over each sub problem and calculates the array indices associated with each sub problem. The final loop iterates over the indices for a given sub problem and performs a Gentleman-Sande butterfly operation on each array element within the index range. If each iteration calculates y_l and z_l at the same time the final loop size can be halved as well as each iteration becoming independent of all others. Whats more a careful observation of figure 4 will reveal that the twiddle factors are reused in the different stages. Instead of calculating the twiddle factors for each operation a look up table is used instead. Finally a bit reversal operation is performed so that the output array is in the right order.

Shared memory

To paralyse the FFT on a shared memory system the OpenMP package was used. Open MP is a shared memory interface. The different cores in a shared memory system all have access to the same memory cache. This means that unlike distributed memory systems there is no communication overhead. There is however overhead due to data access; two cores cannot access the same information at the same time. OpenMP uses a single master thread which spawns multiple worker threads. The worker threads perform the tasks and then recombine once all the calculations are complete. This means that the operating time is set by the slowest thread. To achieve optimal results then, the load should be balanced between all threads [2].

To paralyse the FFT the cython prange function was used. Each iteration within a loop is handed off to a different thread, allowing a loop to be paralysed easily. This of course requires the iterations to be independent of one another, which is the case for second and third loop within the iterative FFT. The first attempt at paralysing the FFT was to use the prange function on the third loop (method 1). There were two reasons for this, both to do with balancing the load. Firstly the number of operations in each iteration is identical so each iteration should take approximately the same time. Secondly the sub problem size for a radix 2 FFT is always an even number and so divide the iterations between an even number of threads should divided the load evenly. However when this method was tested it proved to be slower then the serial iterative FFT.

The poor performance was initially believed to be caused by bottle necking as the sub problem size approached the number of threads. It was believed this may cause thread allocation issues as well as potential load balancing issues. This theory is partially supported when the timing per stage is plotted as shown in figure 5. The timing increases exponentially as the sub problem size decreases. When more threads are used the exponential growth start earlier as the bottle necking is achieved sooner. One problem with this theory though is the sub problem size approaches the number of threads around stage 17, however the slowdown starts much earlier. Upon reflection the reasoning may be simpler. As the sub problem size decreases, the speed up of performing a sub problem using multiple threads decreases, at the same time the number of problems is increasing becoming the rate determining step. Whats more the next sub problem cannot be attempted until the previous one is complete. However this alone cannot explain why increasing the number of threads causes slowdown

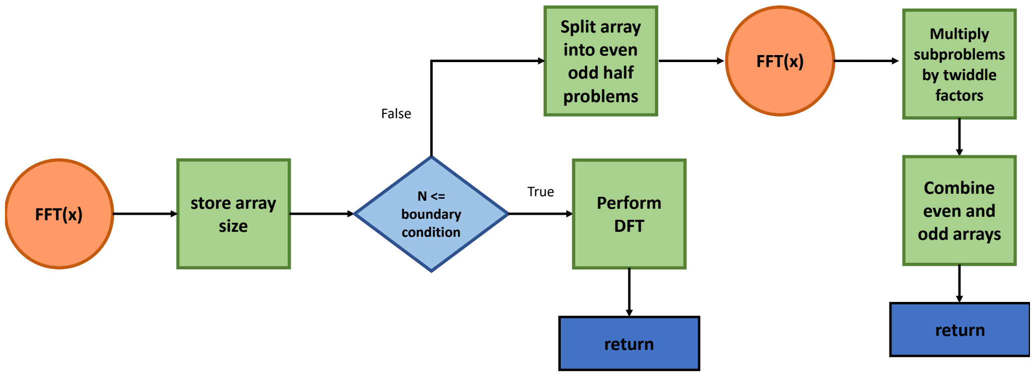
FFT(x)
store array size
N <= boundary condition
False
True
Split array into even odd half problems
FFT(x)
Multiply subproblems by twiddle factors
Combine even and odd arrays
return
Perform DFT
return

FIG. 3: Flow chart of recurrent FFT algorithm.

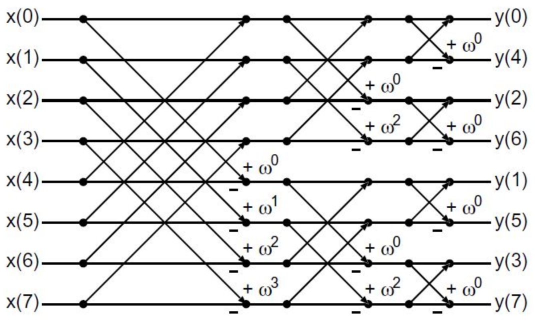

FIG. 4: Data flow of iterative radix 2 DIF FFT. Source [6].

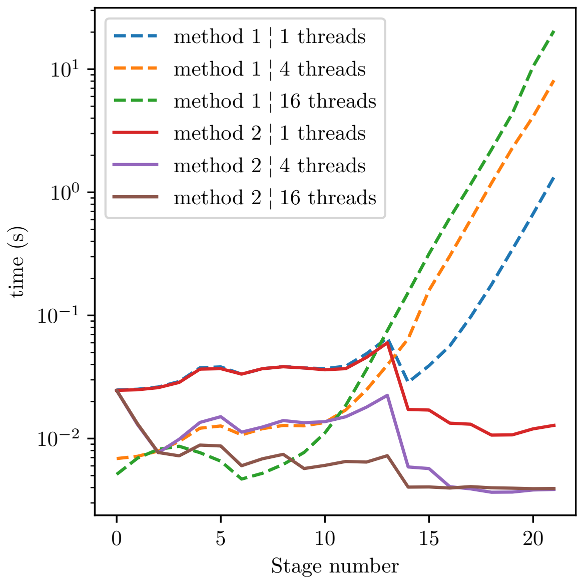

FIG. 5: Timing of different stages for a $N = 2^{22}$ array, when loop 3 is paralysed (method 1) and when loop 2 is paralysed (method 2).

to start earlier. The poor performance is probably a combination of the outlined factors.

It is for these reasons that loop 2 was chosen to be the paralysed loop (method 2). Good load balancing should be achieved automatically as loop 2 iterates over each sub problem and sub problems are all the same size. However when loop two was initially paralised it ran into a similar problem as method 1. For the first couple of stages the number of threads was greater then the number of problems. To solve this the thread size was set to be equal to the number of problems, until the number of problems exceeded the number of available threads, at which point the thread size was capped. The behaviour of method 2 can be seen in figure 5. Method 2 does not experience the slowdown that method 1 experiences, though the initial stages are slower then method 1. Whats more around stage 14 there is a significant drop in the timing per stage. The exact reasoning for this is not known. It would be interesting to see if this or similar behaviours occur for other array sizes. The results in figure 5 do suggest that a more optimal shared memory FFT could be achieved. If the initial performance of method one was combined with the final performance of method 2. Some threshold criteria where the methods were swapped could be easily implemented. Though finding the threshold criteria that holds for different array sizes would be more difficult.

Distributed memory

To paralyse the FFT for a Distributed memory system the MPI (message passing interface) package was used. MPI is designed to run code over multiple cores that do not have access to the same memory cache. This makes MPI perfect for running code on large super clusters. However, MPI has a serious communication overhead which can seriously effect performance. The number of communications between cores should be limited as much as possible. MPI runs the same code on every core with the core rank being used to differentiate the tasks each core is expected to perform. A master core is usually assigned to collect, update and redistribute data as necessary.

Two different MPI FFTs were built. Before going into the differences the similarities between the two functions should be highlighted. Both functions have the cores communicating between stages. With the master core receiving the partial calculation of each worker, updating the master array and then redistributing said array in chunks using MPI. Bcast to the workers for the next stage. The chunk size as well as the number of communications required means the communication overhead is high. Both function are also more memory expensive then the previous FFT algorithms described, as the independence between array pairs cannot be achieved, as pairs may be split between different cores. This means each core requires two arrays, one to hold the initial array values and the second to calculate the new values. The memory cost can be somewhat reduced by having the second array be equal to the cores chunk size.

The difference between the two is based on the mapping of cores to array elements. The first and simplest to understand is the static block mapping (SBM). Essentially each core is assigned an index range and performs calculations on this array range for every stage. However problems arise due to the fact that the array range of sub problems changes each stage. The Gentleman-Sande butterfly operation is dependent on knowing the sub problem index. This means for each stage each core has to work out the start and end array index of each sub problem. The core then loops between the sub problem start and end array index and checks if any of the indices within the range match the cores assigned indices. If there is a match then the core can perform one half of the Gentleman-Sande butterfly. In each stage therefor the core has to iterate over the whole array once. This was foreseen as potentially being a time bottle neck and so the second MPI FFT function was built.

The second MPI FFT uses cyclic block mapping (CBM), figure 13 shows how the processors are distributed for a N = 32 signal array. Essentially each sub problem is divided between the number of cores. The block each core works on is calculated from the sub problem start index and the rank of the core. This means the sub problem index is known to the core and so iterating over the whole array is not required. However there is a problem with CBM. For a signal array of size N = 32 five stages are required, however only 4 stages are shown in figure 13. This is because once the block size becomes less then one the CBM mapping algorithm breaks. To overcome this problem if the block size is less then one SBM is used instead. The CBM FFT requires the number of cores to be an integer power of two whereas the SBM FFT requires the cores to be an even number.

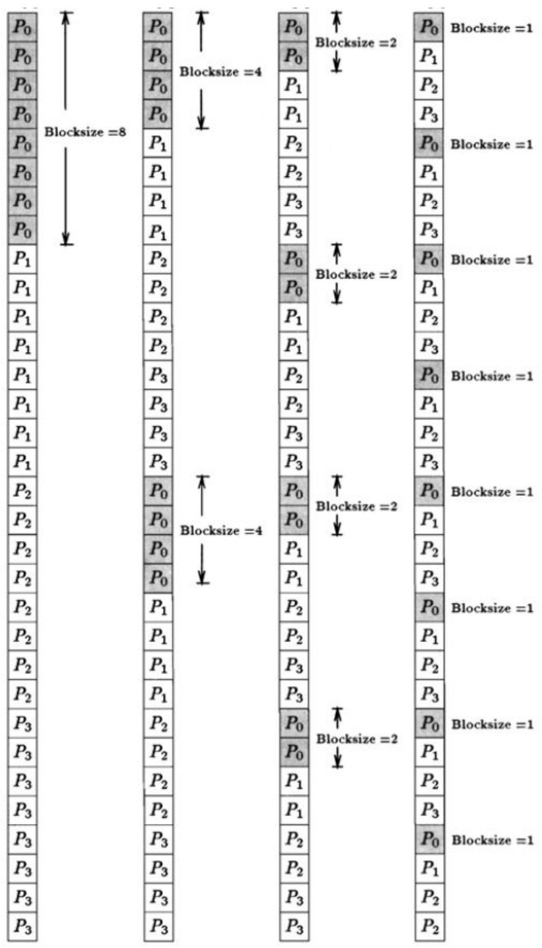

FIG. 6: Processor mapping using CBM for a N=32 array. Source [4].

Fraunhofer diffraction

To test the accuracy of the FFT algorithms that were built, as well as showing one of the many physics applications of the FFT, the diffraction pattern of monochromatic light passing through a square aperture was calculated. The diffraction pattern of light through an aperture is predicted by the Fresnel diffraction equation. Which in the far field (Fraunhofer) limit simplifies to [8]

$$E(x_0) \propto \int \exp\left(-i\frac{kx_0}{z}x_1\right) Aperture(x_1)dx_1, \qquad (12)$$

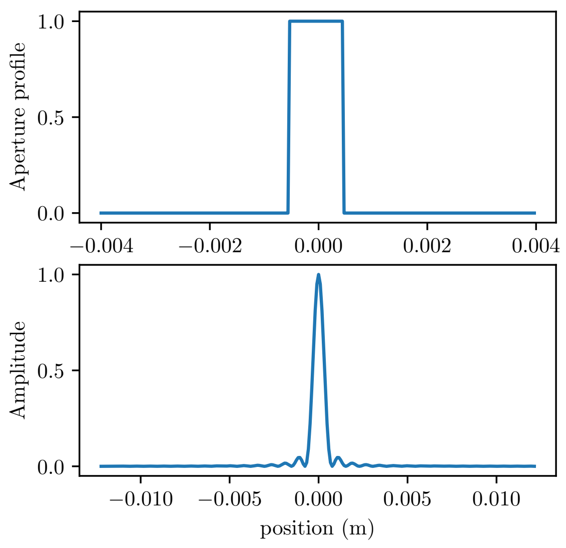

FIG. 7: Diffraction pattern of light passing through a square aperture. The top image is the aperture profile and the bottom image is the light profile.

where x_0 is the coordinate system of the screen the light is projected on, x_1 is the coordinate system of the aperture, k is the wave number and z is the distance between the aperture and the screen. Equation 12 is simply the Fourier transform applied over the aperture profile. The conditions for the far field limit are $z >> D$ where D is the aperture size and $z >> \pi D^2/\lambda$ where λ is the wavelength of light [8].

RESULTS

Fraunhofer diffraction

To meet the far field criteria $0.6\,\mu\mathrm{m}$ wavelength light was simulated passing through a $1\,\mathrm{mm}$ wide square aperture. The distance between the screen and the aperture was set at $1\,\mathrm{m}$. The results are shown in figure 7. The results show the characteristic sinc function of light passing through a square aperture. The results bellow agreed with the numpy FFT algorithm to within a tolerance of 10^{-10}. Whilst the simulation bellow is a nice demonstration of the FFTs practical uses all the FFT algorithms were tested more robustly using the numpy allclose function to check the results against the numpy FFT function.

Sequential algorithms

Once the different sequential FFT algorithms were built and tested, the timings of the different functions, for different array sizes was measured. For the sequential algorithms the timings were taken on a Hp Pavilion Laptop with an Intel i5

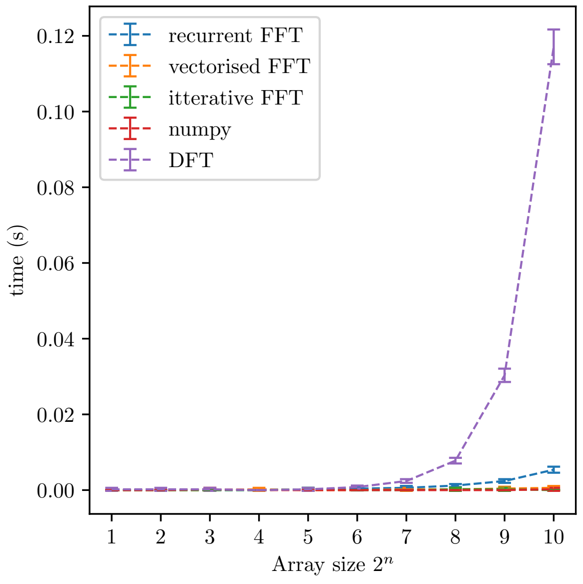

FIG. 8: Timing of sequential FFT algorithms and naive DFT.

2.5GHz processor. Errors displayed on the plots are the standard deviation taken over 5 runs. The deviations in run times are due to background processes running on the same cores. The results for the sequential algorithms are shown in figures 8 and 9. It is immediately apparent from figure 8 the poor performance of the naive DFT when compared to the sequential FFT algorithms. The naive DFT calculates the DFT of a $N = 2^{10}$ array in a comparable time as the iterative FFT calculates the DFT of a $N = 2^{19}$ array. The DFT has a $\mathcal{O}(N^2)$ scaling as expected. The FFT algorithms have a $\mathcal{O}(\log N)$ scaling as expected, this is most clearly seen in the figure 9 subplot (note the x scale is essentially a log 2 scale). The recurrent FFT algorithm performs the worst with the iterative FFT algorithm performing the best coming closest to matching the numpy FFT algorithms impressive speeds.

Parallel algorithms

For the parallel FFT functions the timings of the different functions were performed on BlueCrystal phase 3. This allowed the functions to be run on up to 16 2.6 GHz SandyBridge cores. To evaluate the performance of the parallel FFT functions the efficiency was calculated using

$$\eta = T_1/T_p, \tag{13}$$

where T_1 is the timing of the serial iterative FFT function and T_p is the timing of the paralised FFT functions for p cores. Similar to before the timing of each function was averaged over 10 runs. The error bars on the efficiency however were not calculated using the average but instead through er-

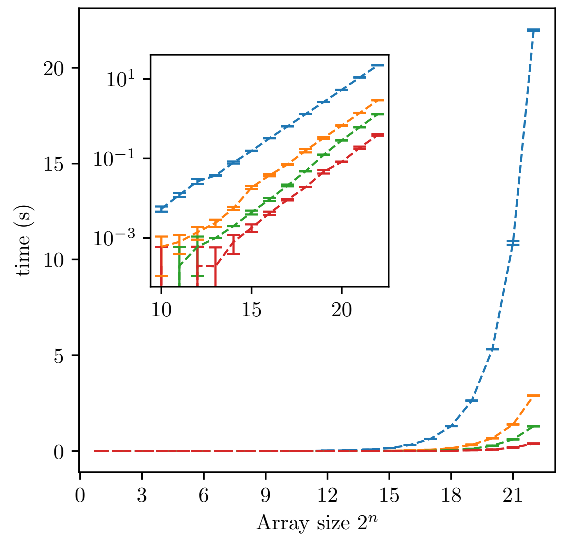

FIG. 9: Timing of sequential FFT algorithms up to an array size of $N = 2^{22}$. The key is the same as figure 8. The subplot has the same axis but the y axis uses a log scale.

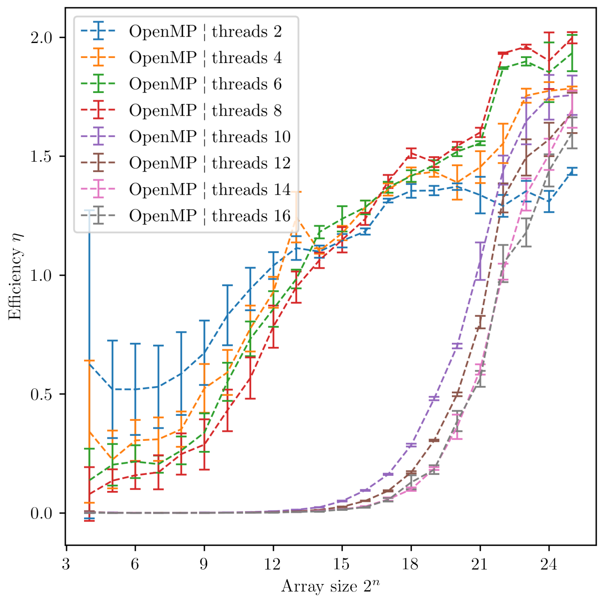

FIG. 10: The efficiency of the shared memory FFT algorithm compared to the iterative FFT algorithm.

ror propagation. The error on the timing T_1 and T_P was taken as the standard deviation over ten runs.

Number of threads	Efficiency	Error
2	1.44	0.01
4	1.79	0.01
6	1.93	0.08
8	2.00	0.02
10	1.76	0.08
12	1.68	0.09
14	1.70	0.08
16	1.60	0.06

TABLE I: Timing of the OpenMP FFT algorithm for an array size of 2^{25}.

OpenMP

The results for the shared memory FFT can be seen in figures 10 and 11. What is immediately clear is the divergence between lower thread runs and higher threads runs. All runs above 10 threads seem to demonstrate similar behaviour. With an initial decrease in efficiency as the array size increases until around 2^7, the efficiency at the minimum is lower for the larger thread sizes. After the minimum, the speed of the 10 + thread runs increases but doesn't become comparable to the speed of the serial function until around array sizes of 2^{21}. After this there is a speed up for the 10+ thread runs when compared to the iterative FFT. The 10+ thread runs overtake the performance of the 2 thread runs for array sizes greater then 2^{21}, however they do not out perform the other sub 10 thread runs. This can be more clearly seen in table I. The sub 10 thread runs performance increases as the array size increases outperforming the serial FFT around array sizes of 2^{14}. Eight threads seems to be the optimal number for large arrays, however the 10+ threads may over take for even larger array sizes.

The slower performance of the multi-thread FFT algorithm for smaller array sizes is not unexpected. When code is run on a single core, data is stored on that cores memory cache. This has a lower latency then the shared memory cache. This does not however explain the relationship between thread number and performance. With a larger number of threads performing worst for smaller N. One common reason for poor performance for a greater number of threads is multiple cores trying to access the same information at the same time. However, for the FFT algorithm each array pair is independent of all other array elements and so only one thread should access each array element per stage.

Instead low granularity is likely the culprit. If granularity is too fine then the performance can suffer due to communication overhead from synchronization, becoming the rate determining step. This may also partially explain the distinct behaviour of the 10+ thread runs. Because the number of threads is capped by the number of problems, for an $N = 2^4$ array only the final stage will use all available threads, if the thread size is greater then 10. For 10 threads 6 threads will have to perform two Gentleman-Sande butterfly operations with the rest performing one. The load imbalance plus the fine granularity of this final stage may cause the distinct divergence from

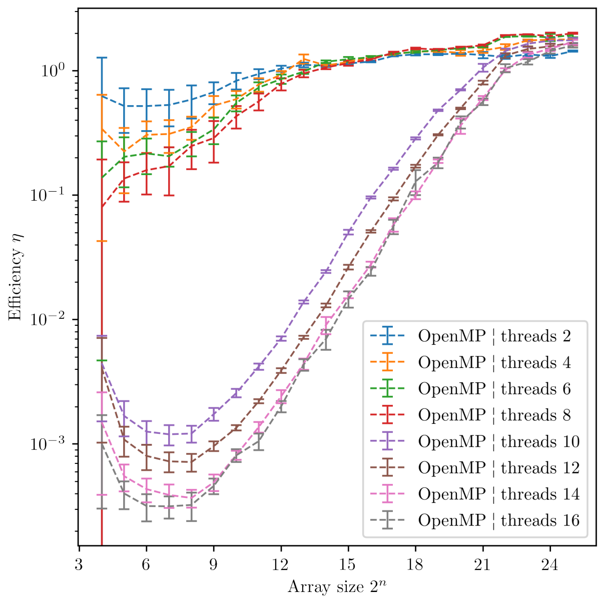

FIG. 11: The efficiency of the shared memory FFT algorithm compared to the iterative FFT algorithm plotted on a log scale.

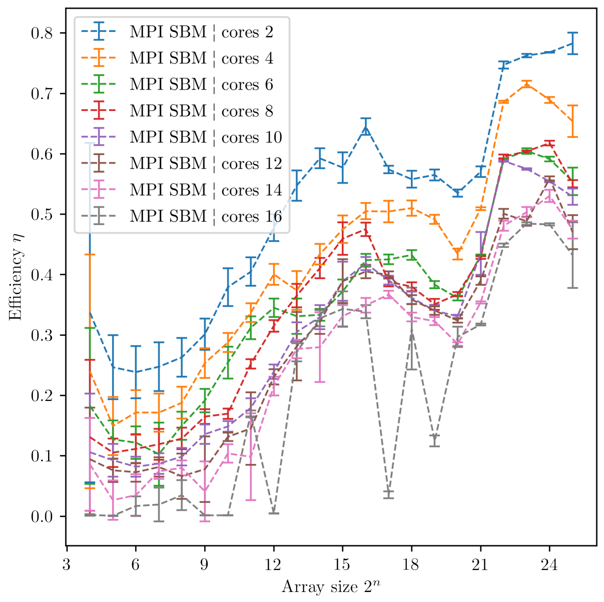

FIG. 12: The efficiency of the distributed memory SBM FFT algorithm compared to the iterative FFT algorithm.

the sub 10 thread runs. This highlights two important points. Firstly, that equal load balancing will only be achieved for thread numbers that are an integer power of 2, possibly explaining the optimal performance of 8 thread. Secondly, it is likely that the divergence that starts from the 10 thread run, is related to 10 threads being the first thread number over an integer power of 2 (8). However is is unclear why this issue persists for so long.

MPI

The results for the distributed memory FFT algorithms are shown in figures 12 and 13. Neither distributed memory algorithm was able to out perform the serial iterative FFT algorithm. This is not unexpected as the communication overhead for the distributed memory systems was always expected to be the limiting factor. The specific algorithms that were used, used block communication at each stage which exacerbated the problem. Somasundaram Meiyappan [10] proposes two different algorithms that can reduce the communication overhead. Both use a similar routine. The initial stage is calculated on a single core, the master core then sends half the array in a non blocking communication to a second core. Each core the works on its half of the problem for the second stage. Both cores then send half the array problem to a third and fourth core, each core works on its problems subset. This process is repeated until the maximum number of cores is reached. As well as a smaller number of communications being required blocking communication is only required in the final stage to combine the results. Implementing and testing a similar algorithms would be a natural extension to this project.

What is unexpected is the poor performance of the CBM FFT compared to the SBM FFT. It was expected that the CBM would perform better then the SBM. One explanation for why this is not the case, is that the SBM FFT can access contiguous data, whereas the CBM jumps over the data array. To understand the behaviour of the SBM and CBM FFT more it would be interesting to produce a timing graph per stage as shown in figure 5. It could be seen how the timing changes when the CBM FFT switch's over to the SBM algorithm.

Whilst the distributed memory FFT did not perform well for the 1D FFT, it is likely that the distributed memory system would work very well for N+1D signal array. To perform a 2D FFT for example, a 1D FFT is performed on every row of a (NxN) array, the array is then transposed and again a 1D FFT is performed on every row. Finally, the array is then transposed back and the DFT has been calculated. This has a $\mathcal{O}(N^2 \log N)$ scaling. As the calculations of each row are entirely independent of one another; rows could be handed of to different cores. The speed up of this paralisation would probably be significant approaching embarrassingly parallel. Limited of course by the number or rows.

CONCLUSION

In this project multiple 1D FFT algorithms were built and tested using both serial and parallel paradigms. For the serial FFTs a recursive, vectorised and iterative FFT algorithms were built. The cythonized iterative FFT algorithm performed the best, calculating the DFT of a $N = 2^{22}$ signal array in 1.30 ± 0.02 seconds. The recursive and vectorised FFT had

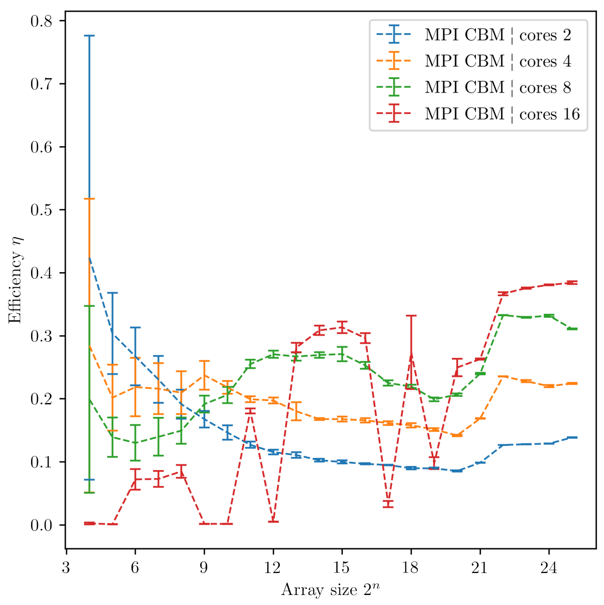

FIG. 13: The efficiency of the distributed memory CBM FFT algorithm compared to the iterative FFT algorithm.

timings of 21.96 ± 0.04 and 2.89 ± 0.01 seconds respectively for the same size signal array. The FFT algorithms were able to produce the Fraunhofer diffraction pattern for light passing through a square aperture. Both shared and distributed memory parallel FFT algorithms were built using the OpenMP and MPI packages. The parallel algorithms were tested and timed on BlueCrystal phase 3. The shared memory FFT algorithm used the cython prange function to achieve a maximum speed up of 2 using 8 threads for a $N = 2^{25}$ size array. However, for lower size arrays the speed up was less significant. With no number of threads achieving any speed up for arrays less then $N = 2^{12}$. The shared memory FFT experienced slow down for smaller arrays. Two distributed memory FFT algorithms were built using cyclic block mapping and static block mapping. Both were slower then the serial iterative FFT algorithm for array sizes up to N^{25}, with the CBM algorithm performing the worst.

[1] Ronald N Bracewell. The fourier transform and its applications. 1965.

[2] Robit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[3] Eleanor Chu and Alan George. Fft algorithms and their adaptation to parallel processing. *Linear Algebra and its Applications*, 284(1):95–124, 1998. International Linear Algebra Society (ILAS) Symposium on Fast Algorithms for Control, Signals and Image Processing.

[4] Eleanor Chu and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms (1st ed.)*. Eleanor Chu and Alan George, 1999.

[5] J. Cooley and J. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19:297–301, 1965.

[6] Takahashi D. *Fast Fourier transform algorithms for parallel computers*. Springer, 2019.

[7] J. Dongarra and F. Sullivan. Guest editors introduction to the top 10 algorithms. *Computing in Science Engineering*, 2(1):22–23, 2000.

[8] Joseph W Goodman. Introduction to fourier optics. *Introduction to Fourier optics, 3rd ed., by JW Goodman. Englewood, CO: Roberts & Co. Publishers, 2005*, 1, 2005.

[9] Michael Heideman and et al. Gauss and the history of the fast fourier transform.

[10] Somasundaram Meiyappan. Implementation and performance evaluation of parallel fft algorithms.