Parallel Approaches to the Fast Fourier Transform

Francis Lewis
MSci. Laboratory, Department of Physics, University of Bristol.
(Dated: December 15, 2022)

One and two dimensional fast Fourier transforms are implemented in C using both MPI and OpenMP for parallelisation. The effect of problem size, and parallelisation technique are examined in terms of execution time speedup and efficiency over a range of processors.

I. INTRODUCTION

A. The Fourier Transform

The Fourier transform is a widely used algorithm with applications in signal processing, spectroscopy and quantum mechanics [1] [2]. A Fourier transform is a way of decomposing signals into different frequency components along with their respective amplitudes and phases. This is useful in media compression [3], machine learning [4] and even radar devices[5]. Due to such widespread use in research and industry, computationally fast methods and low resource implementations are of great interest.

As digital systems are discrete by nature, Discrete Fourier Transforms (DFT) are used in these applications. A DFT of size N takes $O(N^2)$ time when computed directly, but there are algorithms known as Fast Fourier Transforms (FFT) which can compute this in just $O(Nlog(N))$ time [6]. The most common of these is the algorithm first discovered by Gauss in 1805, but not popularised until it's re-discovery by James Cooley and John Turkey in 1965 [8]. Many algorithms are now referred to under the umbrella of "Cooley-Turkey FFTs", and many computational implementations aim to improve their performance. The specific implementation used here is discussed in section II, although the general approach is that of parallel processing.

B. Parallel Computing Methods

Parallelisation methods are a group of closely related techniques used to speedup the execution time of a computer program. In the last 20 years, most processor manufacturers have focused on increasing chip performance through the use of parallel, multi-processor designs [9]. Many computational problems can be divided into smaller problems, which can each run simultaneously on different processors. This dividing up of problems, known as domain decomposition, usually governs the decrease in execution time when using parallel methods. For quantification between parallel methods, the speedup S_P, will be compared.

$$S_p = \frac{T_1}{T_p} \tag{1}$$

The speedup S_p, where p is the number of processors, is the ratio of execution time T_1 using a single process to the execution time T_p when utilising p processes. We can also define the efficiency, E_p, as the speedup divided by the number of processes utilised:

$$E_p = \frac{S_p}{p} \tag{2}$$

In most real world applications, programs are not 'embarrassingly parallel', that is they contain serial sections that must be computed in sequential order. Even in highly parallelisable situations, problem set up and result gathering is usually required to happen in serial. Amdahl's law characterises the available speedup in these situations [10].

$$T_p = T_1(F_s + \frac{F_p}{p}) \tag{3}$$

T_p, T_1 and p have the same meaning as in Eqn. 1, while F_s and F_p are the fractions of code (in terms of execution time) that can be run in serial and parallel respectively.

1. *OpenMP*

Parallel methods can be grouped into shared memory, and distributed memory approaches. The most common shared memory approach is that of OpenMP [11]. OpenMP is an application programming interface (API) providing a simple interface to shared memory multithreading. In this approach, a main thread runs serial code and then 'forks off' into multiple other threads for parallel sections (achieved through parallel 'pragmas' in OpenMP), before running serial code on the main thread again if necessary.

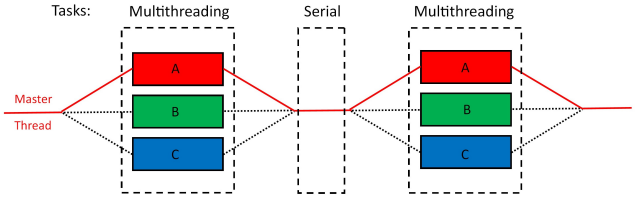

FIG. 1. A diagram showing an example of multithreading code which 'forks' into parallel tasks between serial sections.

Memory, along with variables are shared between by threads by default. This enables OpenMP to be used in existing serial code with very little modification.

2. The Message Passing Interface

A different approach is that of distributed memory parallelisation. Here, each processor has it's own individual memory and communication between processors is done explicitly over a network. The most common communication protocol is the Message Passing Interface (MPI) [12]. MPI libraries provide many different communication functions to allow parallel tasks to be carried out on a range of processors. Usually processors will carry out the same instructions, each on a different sub section of data, before sending the computed data to a master thread which collects this data. This can be thought of as SIMD processing within Flynn's taxonomy [13].

II. METHODS

A. The Cooley-Turkey Algorithm

While the original Cooley-Turkey FFT involves function recursion, this does not lead to easy parallelisation. Thankfully, there are many ways to re-arrange the FFT algorithm to avoid this. Starting with the equation for the DFT:

$$X_p = \sum_{q=0}^{N-1} \left(x_q \, e^{-\frac{2\pi i}{N} pq} \right) \tag{4}$$

Where X_P is the output at index p (ranging from 0 to $N-1$), and x_q is the input value at index q for a total size N.

This can be re-arranged into [14] [15]:

$$\begin{aligned} X_p &= E_p + e^{\frac{-2\pi i}{N} p} \, O_p \\ X_{p+\frac{N}{2}} &= E_p - e^{\frac{-2\pi i}{N} p} \, O_p \end{aligned} \tag{5}$$

Where E_p and O_p are:

$$\begin{aligned} E_p &= \sum_{n=0}^{N/2-1} x_{2n} \, e^{\frac{2\pi i}{N/2} 2np} \\ O_p &= \sum_{n=0}^{N/2-1} x_{2n+1} \, e^{\frac{2\pi i}{N/2} 2np} \end{aligned} \tag{6}$$

With E and O standing for even and odd respectively. E_p sums over the even input values; O_p over the odd values.

The ability to sum over only half of N, is due to the periodicity of the complex exponent, allowing values at step $p + \frac{N}{2}$ to be calculated at step p.

The serial implementation loops over $\frac{N}{2}$ twice, once to calculate every value of X_p (and simultaneously $p + \frac{N}{2}$). To calculate each of these values, a loop over $\frac{N}{2}$ is required to calculate the sums seen in Eqn. 6.

As the aim of this paper is to investigate the speedup of each FFT implementation, each FFT is computed on randomly generated input, produced at run time using the `rand()` function from the C standard library.

So far only the one dimensional case has been discussed. For two dimensional data of size $\sqrt{N} \times \sqrt{N}$ (e.g. an image), a 1D FFT is first computed across every row, and then additional FFTs are computed across every column of the results. As C stores arrays in row-major order, accessing elements by column is much slower than by row [16]. This is due to the processor caching blocks of data along each row [17]. Operating on data within the cache or processor registers is significantly faster than accessing data within the random access memory (RAM). For this reason, in this 2D implementation, after computing FFTs along each row, the $N \times N$ matrix of values is transposed in-place and FFTs are computed along the rows once again before transposing back to the initial orientation.

While there is some computational overhead to the transpose that cannot be parallelised, by using the GNU Scientific Library [18] which contains well optimised routines for matrix transposes, this is negligible.

B. OpenMP

OpenMP was used to parallelise the inner loop (as described in the previous subsection) of each 1D FFT. That is, Eqn. 6 is computed across many different processor threads to calculate values for E_p and O_p in parallel. The 2D approach just repeats this process along each axis, as described in the previous subsection, using parallel 1D FFTs.

C. MPI

The MPI approach parallelises the FFT at a different level to that of the OpenMP implementation. The 1D input array is split into equal size sub-arrays which are distributed to each MPI process via the `MPI_Scatter()` function. These arrays are also given the original array index of each value in the sub-array. Each process then computes the FFT of this sub-array in the same method as the serial code before sending the results to the master thread, which collects and joins the results using `MPI_Gather()`. Every rank apart from the master also shifts the index of the data by their rank number in their respective sub-array. This ensures that once the master rank gathers the sub-arrays, the resulting final array is in the correct order, regardless of the execution order of each process. The 2D case is much the same, with `MPI_Barrier()` used before each transpose to ensure each process has returned it's transformed sub-array. This is to eliminate race conditions where processes may update the same variable in an undesired order.

D. Combined OpenMP and MPI

OpenMP and MPI can be combined, resulting in a distributed cluster of different shared memory parallel processes. The implementation here is largely based upon the existing MPI code. Each MPI sub-process still calculates a smaller

size FFT before the master thread gathers them all. Now, each of these tasks takes advantage of shared memory parallelisation via OpenMP by parallelising the inner loop of the FFT algorithm as described in section II B. Only a 1D implementation of this approach is included due to time constraints.

E. Accuracy

The accuracy of these FFT programs were compared against FFTW [19], a popular and robust C FFT library. For each approach, programs to calculate the root mean square error (RMSE) compared to FFTW were also created, and have been included in the supplementary materials. The RMSE was calculated using Eqn. 7.

$$RMSE = \sqrt{\frac{\sum_{i=0}^{N} (P_i - O_i)^2}{N}} \tag{7}$$

Where O_i is the observed value from a given FFT implementation, P_i is the predicted or 'known' value from the FFTW library results and N is the total problem size.

F. Compiler Flags

As C is a compiled language, there are many choices of both compiler and compiler settings which can have an impact of execution speed. BlueCrytal Phase 4 (BC4) is one the University of Bristol's high performance computing systems, consisting of compute nodes with two Intel E5-2680-v4 2.4 GHz 14 core processors. For code compiled on BC4, the Intel C/C++ compiler (icc) was used, and optimisation flags were investigated following Intel guidelines [20].

III. RESULTS

A. Accuracy

The accuracy of each approach was investigated as described in the preceding section. For both the serial and OpenMP implementations, there was no discrepancies found when compared to FFTW. In the MPI approach, there were significantly more errors. The valid input size (that which returned an error of zero when compared to the FFTW results), was found to increase by a differing step size for changing number of MPI processes when analysed with Python.

Number of MPI processes, p	1	2	3	4
Step increase of valid input size	2	4	6	8

It was found that the condition for valid input size, N, in the MPI implementation using p processes is:

$$N mod(2 \times p) = 0 \tag{8}$$

The MPI implementation was subsequently changed to round up each input value to the nearest valid value before generating data to be transformed. In applications on real world data, this could easily be changed to zero-padding, so is of negligible importance.

B. Compiler Flag Optimisation

The execution times for 2 thread, OpenMP FFTs were recorded. Figure 2 shows these execution times, as measured by the OpenMP function `omp_get_wtime()`.

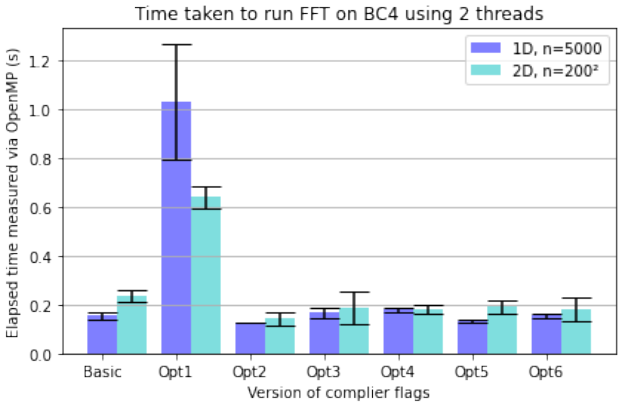

FIG. 2. Execution times for different compiler flag combinations when compiling and running 1D and 2D FFTs using OpenMP over 2 threads on BC4, over 3 repeats.

Most compiler flag combinations result in binaries with similar execution time, with the exception of the `-O1` flag in 'Opt1'. Figure 3 shows these results without the combination labelled 'Opt1' for an easier comparison.

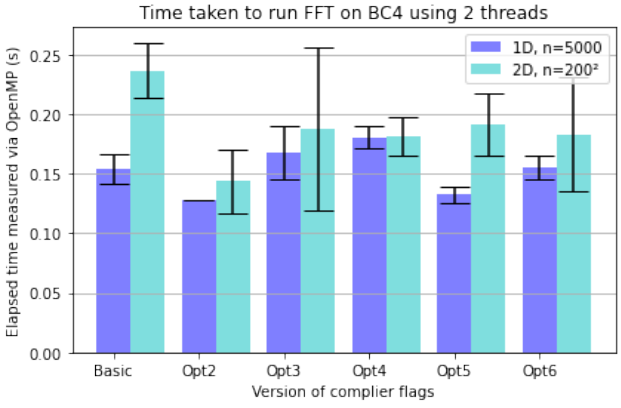

FIG. 3. Execution times for different compiler flag combinations when compiling and running 1D and 2D FFTs using OpenMP over 2 threads on BC4, over 3 repeats.

The fastest flag combination was found to be 'Opt2', where the flags of interest were `-xhost -O2`. Full compiler flags can be found in the appendix and within the supplementary material.

C. Problem Size

For each method, the execution time was measured for varying problem sizes, N. For all of the execution time measurements in this paper, the following functions were used:

- Serial programs: `time() from <time.h>`
- OpenMP programs: `omp_get_wtime()`
- MPI programs: `MPI_Wtime()`

Times were measured for varying problem sizes in both 1D and 2D on an Intel i7-7700HQ 2.80GHz 4 Core processor (i7), running Ubuntu via Windows Subsystem for Linux [21] on Windows 10.

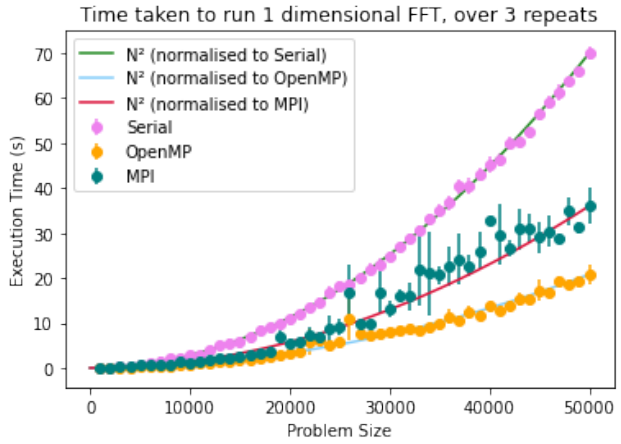

FIG. 4. Execution times for varying problem size for serial, OpenMP and MPI 1D FFTs on BC4. Each is over 3 repeats.

In the 1D case, execution times increased roughly as a function of N^2. The OpenMP implementation was the fastest, followed by the MPI version and then the serial version, with each of these differences also increasing with problem size.

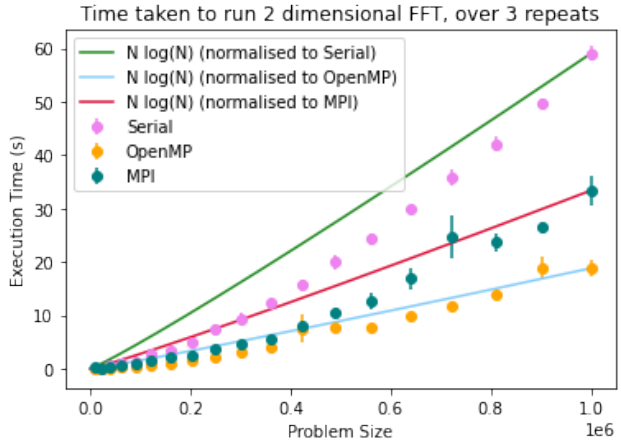

FIG. 5. Execution times for varying problem size for serial, OpenMP and MPI 2D FFTs on BC4. Each is over 3 repeats.

In 2D, OpenMP remains the fastest, followed by MPI and then serial. Here the maximum problem size is larger than in the 1D version, as the absolute times were smaller for the 2D size $\sqrt{N} \times \sqrt{N}$ FFT than the 1D size N FFT across all methods. The differences between each also grow with increasing N, as in 1D . Here, $Nlog(N)$ is plotted instead of N^2. Although the serial code looks to still roughly follow a N^2 relationship, the parallel methods approach that of $Nlog(N)$.

D. OpenMP and MPI

OpenMP and MPI implementations were run on BC4 across varying numbers of threads (up to 28) for both 1D and 2D.

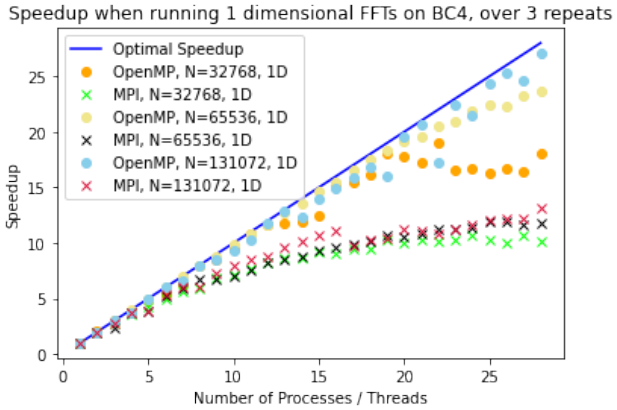

FIG. 6. Speedup of different size OpenMP and MPI 1D FFTs with varying numbers of threads / processes. Using BC4, over 3 repeats.

The OpenMP implementation showed near linear speedup as the number of threads was increased. As the problem size increased, this speedup became very close to the optimal value. The MPI version showed close to linear speedup for a small (< 8) number of processes, but then starts to quickly diverge from the linear relationship. The effect of problem size was less pronounced in the case of MPI, although this may be due to the smaller absolute values of speedup.

Similar results were recorded in 2D. Once again, increasing problem size resulted in the OpenMP method approaching optimal speedup, although to a lesser degree. The largest size tested in 2D, 4095^2, is significantly greater (by a factor of 2^7) than the largest size tested in 1D (131072). Yet in the second case, the speedup is further from optimal. The MPI implementation again showed a very small region of linearity, before plateauing for larger number of processes (> 5). The speedup of the MPI approach was again much less affected by problem size.

The efficiency, as defined in Eqn. 2, again shows the difference between the OpenMP and MPI approaches. In 1D, the OpenMP method retains relatively great efficiency as the number of threads increases. For smaller problem sizes, this does diverge from the optimal level with larger a number of threads. For larger problem sizes, the efficiency remains very

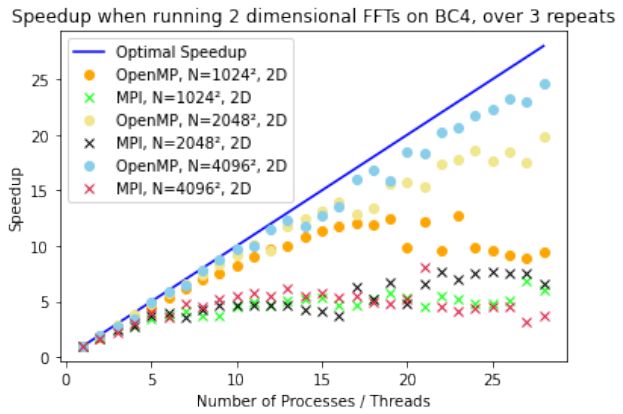

FIG. 7. Speedup of different size OpenMP and MPI 2D FFTs with varying numbers of threads / processes. Using BC4, over 3 repeats.

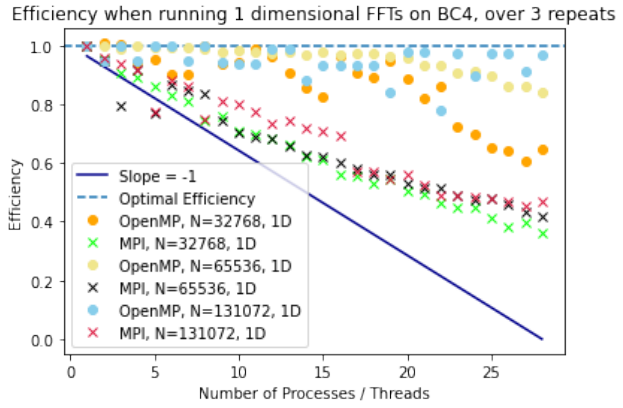

FIG. 8. Efficiency of different size OpenMP and MPI 1D FFTs with varying numbers of threads / processes. Using BC4, over 3 repeats.

close to the optimal level throughout. The MPI method very quickly decreases in efficiency as the number of threads increases, and the problem size makes less of a difference.

In 2D, the general trends were the same for both methods. In the case of OpenMP, the overall efficiency is less than that of 1D for every problem size, although the qualitative relations remain unchanged. The efficiency of the MPI implementation is higher in 2D than in 1D, although still lower than OpenMP. It is also still relatively undisturbed by problem size.

E. Combined OpenMP and MPI

Only certain combinations of the number of nodes, number of MPI processes and number of OpenMP threads are valid choices on BC4. This causes some complications for direct comparison to individual MPI and OpenMP approaches. Due to this, alongside time constraints, only one combination of MPI processes and OpenMP threads was analysed. As shown in Fig. 10, for 4 MPI processes and 7 OpenMP threads the combined approach is faster than the MPI approach using 4 processes for every problem size tested. Compared to MPI, speed increases of 1.61x, 1.98x and 2.01x (for $N = 32768, 66536$ and 132072 respectively) were recorded. The OpenMP approach with 7 threads was significantly faster than the combined approach for all problem sizes.

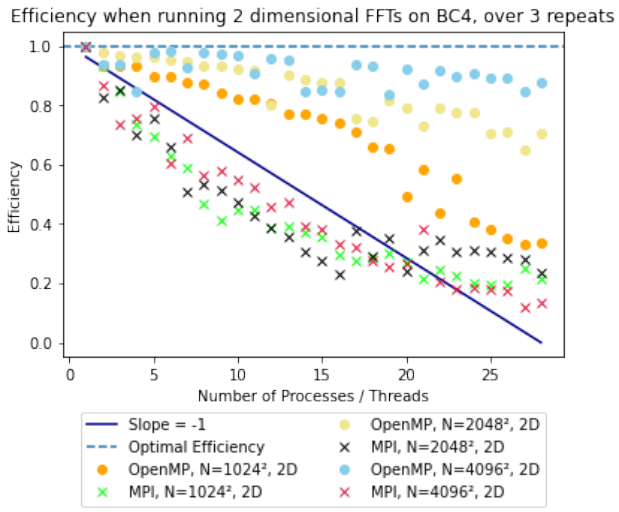

FIG. 9. Efficiency of different size OpenMP and MPI 2D FFTs with varying numbers of threads / processes. Using BC4, over 3 repeats.

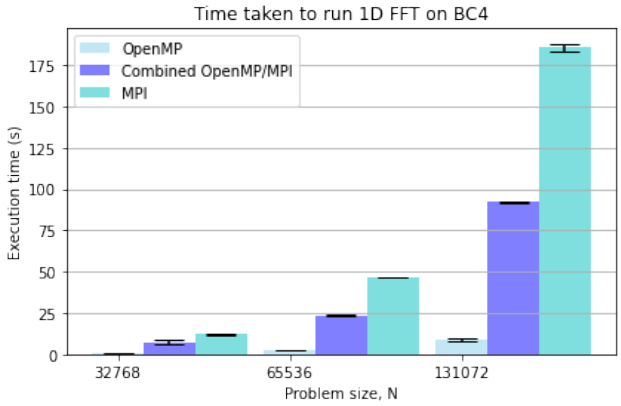

FIG. 10. Execution time of 1D FFT on BC4 for OpenMP, MPI and combined approaches, over 3 different problem sizes. For OpenMP, 7 threads were used. For MPI 4 processes were used. For the combined approach, the number of MPI processes was 2 and the number of OpenMP threads was 7, across 2 nodes of BC4.

IV. DISCUSSION

A. Accuracy

The constraints on input size for the MPI approach (Eqn. 8) can be explained by program design. Due to segmentation fault errors when generating large amounts of random input data, half of the input data is zero padding. For this to work correctly the input size must be even, which explains term 2 in Eqn. 8. When the master thread scatters the input data

to sub-arrays, the size of the array must be a multiple of the number of processes, to ensure the entire array is distributed. This explains the p term in Eqn. 8. Future improvements could divide any length array into sub-arrays, and compute the smaller leftover data on a separate process.

B. Compiler Flag Optimisation

Compiler flag optimisation provided only small improvements in execution time over the default settings. The large difference in execution time (x5) between the Opt1 and Opt2 settings show that naive compiler flag application can lead to a significant *decrease* in speed, even if total speed *increase* is limited.

C. Problem Size

In the ideal Cooley-Turkey FFT algorithm, time should scale as $Nlog(N)$. In the 1D implementation, this relationship is less time efficient and much more closely follows N^2. As the total problem size increases in the 2D case, this relationship does begin to tend towards $O(Nlog(N))$ time. In the OpenMP case, while the most computationally intensive loop is parallelised, there is still serial sections of the program. Setting up the arrays, allocating memory and generating random input values were all carried out in serial. Both OpenMP and MPI do have a computational cost to setting up communication, which is relatively fixed. This overhead will increase the execution time from the perfectly parallelised case. The contribution is expected to get smaller in proportion as the problem size increases (and the computationally intensive parallelised part dominates), which does occur.

As expected, both parallel methods were faster than the serial implementation. The MPI version is significantly slower than OpenMP for problem sizes above $N = 20000$. The MPI program not only scatters the input data between processes, but also scatters an array of index values used to order the data regardless of the computation order. The way in which this is implemented means that the data scattered is a mixture of matrix rows and columns. This loss of cache efficiency likely contributes to the longer execution times. In further work, redesigning this data handling to ensure row-major computations will likely lead to a significant speedup. The errors on the MPI results were significantly larger than the other methods. One explanation of this is due to taking these measurements on an i7 running Windows 10. There will likely be many more background tasks running than in the minimal UNIX environment on BC4 causing increased competition for processor cycles. If only a single process is delayed, this will impact the overall run time significantly as every process must return data to the master process before exiting.

D. OpenMP, MPI and Combined Approaches

In 1D, the OpenMP program showed near optimal speedup and efficiency across varying numbers of threads. As the problem size increases, the speedup more closely follows this relationship. As described in section IV C, this is likely due to an overhead in setting up OpenMP communication. Interestingly, for even larger problem sizes in 2D the relationship is not as close to the ideal case. The increasing problem size should theoretically increase the speedup. However the addition of two matrix transposes in serial, along with data changes between C99 standard `complex` and GSL `gsl_complex` types, adds overhead. This can be thought of as increasing F_s in comparison to F_p in Eqn. 3.

OpenMP shows weak scaling as both the problem size and the number of threads has to increase for linear speedup. The $1D, N = 131072$ case looks to have linear speedup without increasing N, e.g. strong scaling. What is more likely is that under the maximum number of 28 threads, N is large enough for linear speedup. This is the maximum number of shared memory processors on a single node of BC4. Given a larger number of threads, it is expected that this speedup would plateau as in the smaller N cases.

MPI is not limited to a single node of BC4. Due to the inefficient and slow behaviour of the MPI method below 28 processes, using more than one node would not have brought a large benefit - although 2 nodes were trialed for the combined implementation. The MPI program was slower than the OpenMP version in all scenarios. This, combined with the fact that both the speedup and efficiency were unaffected by changing N, points towards serial bottlenecks and program design being dominating factors in execution time when compared to the parallelised tasks. While preliminary measurements were taken to profile the run time of both the parallel and serial sections, a comprehensive analysis of this would be of great benefit.

The combined approach was found to be notably faster than the equivalent MPI program, with execution time halved for 3 and 4 OpenMP threads per distributed task. This is not the case in all problems [22]. The OpenMP-only approach was still significantly faster than this combined method though. If improvements are made to the MPI code, it is expected that this combined approach will continue to be faster than the MPI-only program.

V. CONCLUSIONS

Accurate, parallel FFT algorithms for 1D and 2D were introduced. The speedup and efficiency of OpenMP, MPI and combined parallel approaches to the FFT are described. The execution time of all parallel methods was less than that of the serial version. Using OpenMP gained the largest speedup, with close to strong scaling observed in the regime investigated. Shortcomings of the MPI approach are analysed and it is shown that using a combined approach can alleviate some of the performance issues from these.

VI. ACKNOWLEDGMENTS

This work was carried out using the computational facilities of the Advanced Computing Research Centre, University of Bristol - http://www.bris.ac.uk/acrc/.

VII. APPENDIX

1. Compiler Flags

- Basic: icc -o output.o input.c -lgsl -lgslcblas -fopenmp -lm

All the following combinations contain the basic flags and options. Additional flags for each combination are listed below:

- Opt1: -xhost -O1
- Opt 2: -xhost -O2
- Opt3: -xhost -O3
- Opt4: -O2
- Opt5: -xhost
- Opt6: -xhost -ipo -O2

VIII. REFERENCES

[1] Kita, D.M., Miranda, B., Favela, D. et al. High-performance and scalable on-chip digital Fourier transform spectroscopy. *Nature Communications*, *9*, 4405 (2018). doi:10.1038/s41467-018-06773-2

[2] Bracewell, R. N., & Bracewell, R. N. (1986). *The Fourier transform and its applications* (Vol. 31999, pp. 267-272). New York: McGraw-Hill.

[3] Rasheed, M. H., Salih, O. M., Siddeq, M. M., & Rodrigues, M. A. (2020). Image compression based on 2D Discrete Fourier Transform and matrix minimization algorithm. *Array*, *6*, 100024. doi:10.1016/j.array.2020.100024

[4] Yuan, Y., Xun, G., Jia, K., & Zhang, A. (2017). A Multi-View Deep Learning Method for Epileptic Seizure Detection Using Short-Time Fourier Transform. *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology,and Health Informatics*, 213–222. doi:10.1145/3107411.3107419

[5] J. Xu, J. Yu, Y. -N. Peng and X. -G. Xia. (2011). Radon-Fourier Transform for Radar Target Detection, I: Generalized Doppler Filter Bank. *IEEE Transactions on Aerospace and Electronic Systems*. *47*(2), pp. 1186-1202. doi:10.1109/TAES.2011.5751251.

[6] Bekele, A. J. A. A. (2016). Cooley-turkey fft algorithms. *Advanced algorithms*.

[7] Heideman, M., Johnson, D., & Burrus, C. (1984). Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine, 1*(4), 14–21. doi:10.1109/MASSP.1984.1162257

[8] Cooley, J. W., & Tukey, J. W. (1965). An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation, 19*(90), 297–301. doi:10.2307/2003354

[9] Pacheco, P., & Malensek, M. (2021). *An introduction to parallel programming an introduction to parallel programming* (2nd ed.). doi:10.1016/c2015-0-01650-1

[10] Amdahl, G. M. (2013). Computer Architecture and Amdahl's Law. *Computer, 46*(12), 38–46. doi:10.1109/MC.2013.418

[11] Dagum, L., & Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering, 5*(1), 46–55. doi:10.1109/99.660313

[12] Walker, D. W., & Dongarra, J. J. (1996). MPI: a standard message passing interface. *Supercomputer, 12*, 56-68.

[13] Flynn, M. J. (1966). Very high-speed computing systems. *Proceedings of the IEEE, 54*(12), 1901–1909. doi:10.1109/PROC.1966.5273

[14] Takahashi, D. (2019). *Fast Fourier transform algorithms for parallel computers* (1st ed.). doi:10.1007/978-981-13-9965-7

[15] Akl, S. G. (1989). *The design and analysis of parallel algorithms*. United states: Prentice-Hall

[16] Almurayh, A. (2022). Improved Matrix Multiplication by Changing Loop Order. *Mobile Information Systems*, vol. 2022. doi:10.1155/2022/9650652

[17] Rountree, B. (2012). *Cache Performance Analysis and Optimization* (No. LLNL-TR-604112). Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States).

[18] GNU. (2022, Feb). *GSL - GNU Scientific Library*. https://www.gnu.org/software/gsl/

[19] Frigo, M., & Johnson, S. G. (2005). The Design and Implementation of FFTW3. *Proceedings of the IEEE, 93*(2), 216–231. doi:10.1109/JPROC.2004.840301

[20] Intel. (2018). *Quick Reference Guide to Optimization with Intel® C++ and Fortran Compilers v19* (Report No. 671224). https://www.intel.com/content/www/us/en/content-details/671224/quick-reference-guide-to-optimization-with-intel-c-and-fortran-compilers.html

[21] Microsoft. (2022, Dec). *What is the Windows Subsystem for Linux?*. https://learn.microsoft.com/en-us/windows/wsl/about

[22] Cappello, F., & Etiemble, D. (2000). MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. SC '00: *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 12–12. doi:10.1109/SC.2000.10001