Parallelization of Fast-Fourier Transform (FFT) Algorithms

Kate L. Bolam,[1]★
[1]HH Wills Physics Laboratory, University of Bristol, Tyndall Avenue, Bristol BS8 1TL

27 November 2024

ABSTRACT

Efficient parallelization of the radix-2 Cooley-Tukey Fast Fourier Transform (FFT) algorithm is crucial for high-performance computing (HPC) applications, being a cornerstone in signal processing, scientific simulations, and data compression. This report evaluates parallel FFT implementations using OpenMP, MPI, multiprocessing, and JIT parallelization, exploring performance across shared and distributed memory systems upon different hardware. Challenges such as communication overhead, synchronisation delays, and matrix transpositions are identified as key bottlenecks to parallel performance. The codebase, along with SLURM scripts for execution on HPC systems, is publicly available on GitHub.

1 INTRODUCTION

The Fast Fourier Transform (FFT) is a cornerstone of digital signal processing, and was central to the advance of the digital age. Through a 'divide-and-conquer' approach, the FFT implements a markedly efficient computation of the Discrete Fourier Transform (DFT), by decomposing an N-point time-domain signal into N frequency signals. In signal processing, FFT is instrumental in real-time processing in telecommunications, audio engineering and image processing through its efficient signal analysis.

Beyond this, the FFT has become significant in scientific computing, with particular input in the simulation and modelling of complex systems. FFT-based methods are at the core of all Direct Numerical Simulation (DNS) codes that are used in studies of fluid dynamics and turbulence (e.g. Orszag 1970). Such computationally-intensive simulations push the limits of high-performance computers (HPC); it is therefore crucial to implement only the most efficient algorithms for both serial and parallelized FFTs, which are often the bottlenecks in high-performance simulations.

The advent of multi-core processors and HPC clusters has brought parallel computing to the forefront, enabling the distribution of calculations across multiple threads. It is important to understand the multi-level computer architecture in order to appreciate how the multi-level parallelism that follows from this operates, as will be discussed from here. A node refers to a single computing unit in a larger system, often a cluster or HPC environment, and consists of one or more processors (CPUs), each containing multiple cores. A core executes a process autonomously, using a small cache memory that is only available to that one core. However, all cores share a common level of cache memory, enabling inter-core communication. Within each hierarchical level, there are therefore opportunities for tasks to be distributed; parallelization leverages the available cores to effectively 'spread the work'.

No matter how elegantly a parallel system is designed, there are inherent limits to the speedup, along with unavoidable overheads that come as a trade-off with parallelism. Tasks executed in parallel often depend on data computed across multiple cores (data dependencies). These dependencies must be communicated between cores before the task can proceed, which can introduce significant overhead. Amdahl's law sets theoretical speedup limits when using multiple processors through the concept of diminishing returns. Such that, the overall performance speedup gained by optimising a single part of a system is limited by the *fraction of time that the improved part is actually used* (Amdahl 1967). This principle highlights the importance of balancing parallel performance with the inevitable overhead that comes with increasing parallelism - a trade-off that will be explored in this report.

This report examines various parallelization methods for both shared and distributed memory architectures. It presents an evaluation of their performance, assessing speedup and scalability. A discussion of the inherent caveats and trade-offs associated with parallel computing then follows.

2 FAST FOURIER TRANSFORM ALGORITHM

The Cooley-Tukey FFT algorithm belongs to the family of 'Radix-Based' algorithms - specifically radix-2 (Cooley & Tukey 1965). Radix-2 algorithms decompose input data based on factors of the problem size N, where the problem size N is a power of 2. The radix-2 Cooley-Tukey FFT algorithm is the most common FFT algorithm and uses a recursive method to efficiently compute the DFT, for $N = 2^m$, where m is an integer.

In order to understand how the radix-2 Cooley-Tukey algorithm operates, we must first consider the Continuous Fourier Transform (CFT). The CFT is a technique that decomposes a waveform (or any periodic function) into its constituent frequencies. Equation 1 demonstrates the CFT of a function $x(t)$, where $X(f)$ represents the frequency-domain representation of $x(t)$.

$$X(f) = \int_{-\infty}^{\infty} x(t) e^{-i2\pi ft}\, dt \tag{1}$$

In practice, signals are often sampled at discrete intervals, therefore $x(n)$ is now the discrete signal (where n is an integer), replacing continuous-time signal $x(t)$. This gives rise to the DFT, expressed as Equation 2 (for a 1D vector), where the frequency spectrum is discretised into N equally spaced points. X_n is the complex number that represents the phase of the k-th frequency constituent, x_n is the input finite signal at sample n, and N is the number of equally-spaced points, i.e. the finite signal length.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i\frac{2\pi}{N}kn} \tag{2}$$

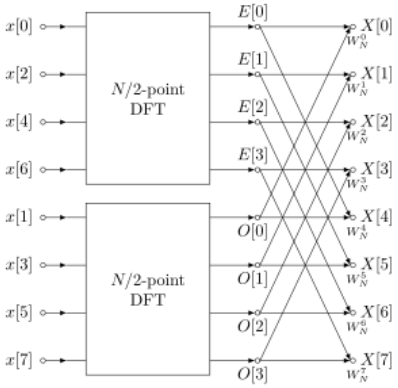

Figure 1. Data flow diagram for a radix-2 FFT with $N = 8$, illustrating the recursive decomposition into smaller DFTs and the butterfly operations used to combine results.

The radix-2 Cooley-Tukey FFT algorithm optimises the computation of the DFT through exploiting its periodicity and symmetry. By assuming $N = 2^m$, the DFT computation is divided into two parts: even-indexed inputs and odd-indexed inputs, as depicted by representations 3 and 4.

$$x_{2m} \quad \text{for } m = 0, 1, \ldots, \frac{N}{2} - 1 \quad \text{(even-indexed elements)}, \tag{3}$$

$$x_{2m+1} \quad \text{for } m = 0, 1, \ldots, \frac{N}{2} - 1 \quad \text{(odd-indexed elements)}. \tag{4}$$

The FFT first computes the DFTs of the even-indexed inputs ($x_{2m} = x_0, x_2, ..., x_{N-2}$), and of the odd-indexed inputs ($x_{2m+1} = x_1, x_3, ..., x_{N-1}$) and then combines the two results to produce the DFT of the whole finite sequence, as show by Equation 5.

$$\begin{aligned} X_k &= \sum_{m=0}^{N/2-1} x_{2m} e^{-i\frac{2\pi(2m)k}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-i\frac{2\pi(2m+1)k}{N}} \\ &= \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-i\frac{2\pi mk}{N/2}}}_{E_k} + e^{-i\frac{2\pi k}{N}} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-i\frac{2\pi mk}{N/2}}}_{O_k} \end{aligned} \tag{5}$$

By factoring out the common exponential term, the sum reduces down to the DFTs of the even and odd indices:

$$X_k = E_k + e^{-i\frac{2\pi k}{N}} O_k, \quad k = 0, 1, \ldots, \frac{N}{2} - 1. \tag{6}$$

The results of the recursive steps are then combined using a structure known as a "butterfly," which consists of a pair of additions and multiplications by the twiddle factor $W_N^k = e^{-i\frac{2\pi k}{N}}$, and each stage of the FFT consisting of $N/2$ butterfly operations. This recursive technique reduces the computational complexity of the DFT from O(N^2) to O($N\log N$).

3 PARALLELIZATION

3.1 Shared memory

3.1.1 Thread-based: OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that facilitates parallelization for multi-core, shared-memory systems. OpenMP leverages 'multi-threading' to split a larger task into smaller tasks, through a 'fork-join' model (illustrated in Figure 2. The larger task is initiated as a single process, the 'master thread', which executes serially until a parallel region is encountered (i.e., a fork). Each thread then rejoins the master thread when the computations on each are complete.

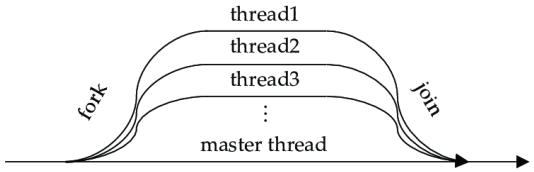

Figure 2. A schematic of the fork-join model utilised by OpenMP (Zhang et al. 2021)

The FFT is parallelized using OpenMP through splitting the computation of smaller DFTs (E_k and O_k) across threads. Each thread handles the calculation of even- or odd-indexed DFTs autonomously. Synchronisation barriers temporarily halt the master thread, ensuring that all threads complete their calculations before advancing butterfly operations, where results are combined using twiddle factors.

Synchronisation ensures that no thread evolves prematurely, forcing all threads to progress to sequential stages of the code synchronously. Major overhead may be incurred at synchronisation if load distribution is not optimised.

The scheduling of tasks may be tuned in OpenMP to optimise parallel performance, depending on workload characteristics. Evenly distributed tasks with minimum runtime overhead may benefit from 'static' scheduling, where tasks are divided into equally-sized chunks and assigned to each thread. 'Dynamic' scheduling may be utilized for irregular workload distribution, by allocating tasks to threads when they become available. While static scheduling minimises runtime overhead for uniform workloads, dynamic scheduling is better suited for irregular tasks but may introduce additional overhead from task assignment.

The advantage of shared memory architectures (such as OpenMP) is the lack of communication overhead, as all threads share the same memory space, allowing for efficient data access and task execution. However, this shared memory model is also its downfall, as only one core can access a specific piece of information at a time, often leading to bottlenecked performance.

3.1.2 Process-based: Multiprocessing

Process-based parallelization in Python is facilitated by the `multiprocessing` module, which enables tasks to run in parallel by creating multiple independent processes (Python Software 2024). Unlike thread-based parallelization (implemented by OpenMP), process-based parallelization bypasses Python's Global Interpreter Lock (GIL), allowing true parallelism across multiple CPU cores (Aziz et al. 2021). Python's GIL is a *mutex* (mutual exclusion lock), essentially allowing only one thread to execute Python bytecode at a time, severely limiting the performance potential of multithreading for CPU-bound tasks, such as FFTs.

Python's `multiprocessing` module employs a 'Master-Worker' model, where a master process manages a pool of worker processes. The master process maintains a pool of tasks, divides them into subproblems, and dynamically distributes these to worker processes as they become available. Each worker operates independently, bypass-

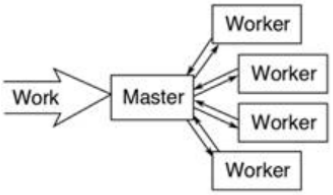

Figure 3. Illustration of the *Master-Worker* model employed by Python's `Multiprocessing` module (McClanahan & Sterling 2020)

ing the GIL to execute tasks concurrently. Upon completing a task, workers send their results back to the master process.

While multiprocessing is effective for CPU-bound tasks, its default approach of using independent memory spaces for each process results in higher memory usage compared to OpenMP. Although shared memory can be used explicitly in multiprocessing, it requires additional effort to manage synchronisation and data consistency between processes. Additionally, multiprocessing involves some inter-process communication (IPC), when transferring data between master and worker processes, which introduces some level of latency that multithreading (via OpenMP) is not prone to.

3.2 Distributed memory (Message Passing Interface; MPI)

Message Passing Interface (MPI) is a standard messaging interface that facilitates communication between cores in distributed memory architectures (Gropp et al. 1999). MPI parallel performance scales efficiently with a large number of cores, as unlike OpenMP, they are each not limited to a shared space of memory.

MPI assigns one core as the 'master' which handles the Input/Output (I/O) tasks, and the all cores (workers and master) carry out the computation in an distributed manner. The distribution of tasks across cores must be specified using MPI_Scatter, whose results are then gathered to the master core using MPI_Gather().

Inter-core communication generates overhead, which can dominate execution time in tasks requiring frequent exchange of data dependencies- in this instance, the matrix transpositions in the FFT. For computation of the 1D FFT here, the input 1D matrix, N, is divided equally among the available processes. Each process independently computes the FFT on its assigned chunk using a local FFT algorithm, which is embarrassingly parallel and does not require inter-process communication, thereby minimising communication overhead. Once the local FFT computations are completed, the partial results from all processes are gathered at the master core, where the final FFT is assembled. The communication overhead associated with the 1D FFT computation is relatively minimal, since most of the computation is carried out locally on the worker cores.

The 2D FFT introduces an additional level of communication due to the matrix positions between FFT computations along more than one dimension. The input NxN matrix is divided row-wise, each process receiving $\frac{N}{p}$ rows, where each process computes a 1D FFT locally on its row, without inter-process communication. Following this, a matrix transposition occurs in order to align data for column-wise FFTs. This is achieved through all-to-all communication, where each process sends and receives data to and from all other processes to transpose the data (Dalcin et al. 2019). The column-wise FFTs are computed and returned to the master core to be assembled.

While FFT butterfly operations in 1D and 2D FFTs are embarrassingly parallel, the matrix transposition occurring the the 2D computation incurs a great degree of communication overhead. This communication overhead grows with both problem size and number of processes, being contingent on data movement as opposed to computation. This is the dominant bottleneck on the scalability of MPI FFT computation.

3.3 Runtime compilation (Just-In-Time; JIT)

Runtime compilation, or 'just-in-time' (JIT) compilation is the process of compiling a program during execution. JIT parallelization is perhaps the most obvious form of multiprocessor JIT compilation, achieving modest speedup with minimal effort from the programmer (Saltz et al. 1990). The use of JIT decorator, `njit(parallel=True)` from Numba effectuates the code to be analysed for opportunities to parallelize loops (Numba Development 2024). If parallel loops are identified, Numba employs multi-threading to execute different chunks of the loop concurrently. For explicit parallelization in Numba, loops can be defined using the function '`prange`', which allows each iteration to be distributed on a separate thread. This mode of parallelization is simple and sometimes effective for embarrassingly parallel programs that contain iterative and separable loops. Nevertheless, JIT parallelization fails for loops with inter-iteration dependencies that require sequential execution and so cannot be parallelized, given Numba's lack of IPC.

4 RESULTS

4.1 Serial Program

A serial FFT implementation performs the recursive DFT decomposition entirely on a single core, executing butterfly operations sequentially to combine the results of smaller DFT computations. The serial program uses the `numpy.fft` module, which employs the radix-2 Cooley-Tukey 'divide-and-conquer' strategy discussed in Section 2 (Harris & Millman 2024). By splitting the input matrix into smaller sub-matrices, the serial FFT ensures that each sub-process fits within the CPU's L1 cache - the level of cache memory that is the fastest and closest to the core, therefore with the lowest latency (Farber 2011). Precomputed twiddle factors are stored contiguously in memory and reused across butterfly operations, to minimising redundant memory access and maximising throughput by making efficient use of cache locality.

The serial FFT program was executed on two hardware systems: the HPC BlueCrystal Phase 4 (BC4), with 16 x 2.6 GHz SandyBridge cores, and a 10^{th} Gen Intel Core i5-1135G7 processor (i5), with 4 cores and a base frequency of 2.4 GHz. Execution times for both 1D and 2D FFTs were measured across a range of problem sizes N, or NxN, as shown in Figure 4. As expected, the HPC BC4 outperforms the i5 processor in terms of scalability and overall clock speed.

4.2 1D FFT

Figure 5 presents the results of the parallelized 1D FFT computation using OpenMP, MPI, multiprocessing, and JIT parallelization, executed on a single BC4 node, with JIT results obtained on the i5 processor. In shared-memory methods (OpenMP and multiprocessing), the serial implementation outperforms at smaller problem sizes ($N \lesssim 2^8$), as it avoids overhead associated with synchronisation barriers.

Both shared-memory methods scale most effectively for larger problem sizes, where larger thread count parallel systems begin to

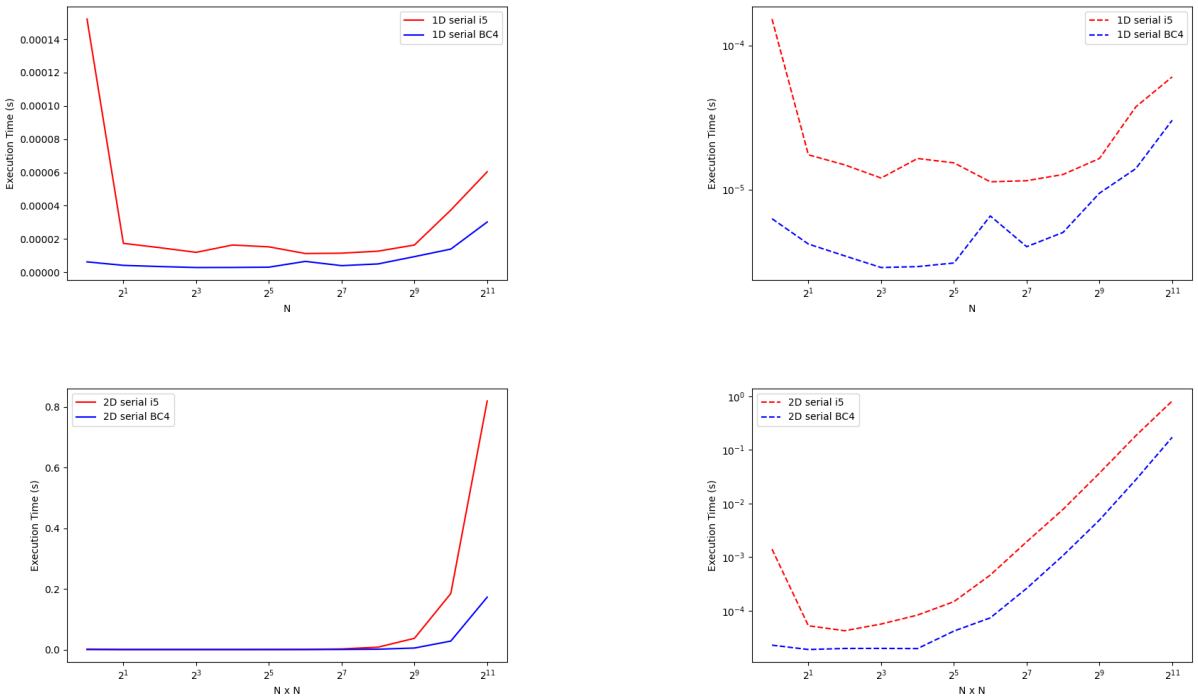

Figure 4. Plots of execution time versus problem size N, NxN, for both 1D and 2D serial FFT computation. Top left: 1D FFT (linear time), top right: 1D FFT (log time), bottom left: 2D FFT (linear time), and bottom right: 2D FFT (log time).

outperform the serial implementation at $N \gtrsim 2^8$. It is important to note the difference in stability between the OpenMP and multiprocessing parallelization methods across the given problem sizes; execution times are less variable for OpenMP performance than that of multiprocessing, being owed to OpenMP's use of lightweight threads and efficient thread-level synchronisation, compared to multiprocessing's method of separate processes that require IPC.

For MPI, the serial implementation consistently outperforms at all problem sizes. This could perhaps be due to the strengths of MPI being underutilised, for a relatively computationally inexpensive task like a single 1D FFT computation. In essence, each process is not being given 'enough to do', causing computational results to potentially be sitting idle in their cores much of the time, meaning that the workload per process is too small to maximise throughput by MPI. Moreover, performance is worsened by inevitable communication associated with MPI. The workload per core may be small, but MPI still incurs significant overhead from distributing N/p elements to each process, and then gathering the results.

For JIT parallelization, at $N \lesssim 2^4$, the computational workload is minimal, and Numba's lack of IPC in JIT parallelization eliminates communication overhead, enabling parallel threads to compute more tasks efficiently. At $N \gtrsim 2^4$, JIT parallelization underperforms as for increased problem sizes, data dependencies between threads grow, which require more memory access that JIT parallelization cannot facilitate.

Overall, in this analysis, shared-memory methods emerge as the most efficient parallelization mode for 1D FFT, with distributed-memory methods (MPI) being underresourced and JIT unsuited for this type of computation.

4.3 2D FFT

Figure 6 presents the performance results for the parallelization of the 2D FFT computation using OpenMP, MPI, multiprocessing, and JIT parallelization. Similar to the case of a single 1D FFT computation, the serial implementation outperforms parallelized methods for smaller problem sizes, as the overhead of parallelization dominates the relatively low computational intensity.

OpenMP demonstrates the best scaling performance, overtaking the serial implementation at lower N than the other methods, ($N \approx 2^3$, compared to multiprocessing's overtake at $N \approx 2^{10}$, and MPI's overtake at $N \approx 2^9$) and providing better scaling for larger problem sizes.

It is observed that at larger N, MPI has disappointing parallel performance, achieving only marginal improvement over the serial case. This is likely due to the matrix transpose operation that is carried out one element at a time, scaling with $O(N^2)$, thus creating a bottleneck in performance.

The performance of 2D FFT using JIT and multiprocessing parallelization mirrors the performance behaviour observed in the 1D case, caused by similar underlying factors. For JIT, the employment of Numba's loop-parallelization strategy once again yields results with poor scalability, with parallel performance diminishing as data dependencies grow and memory access demands intensify during 2D FFT computation.

4.4 Speedup & Efficiency

The respective speedup and efficiencies for each parallelization method were calculated using equations 7 and 8, where T is the execution time, and p is the number of processors or threads.

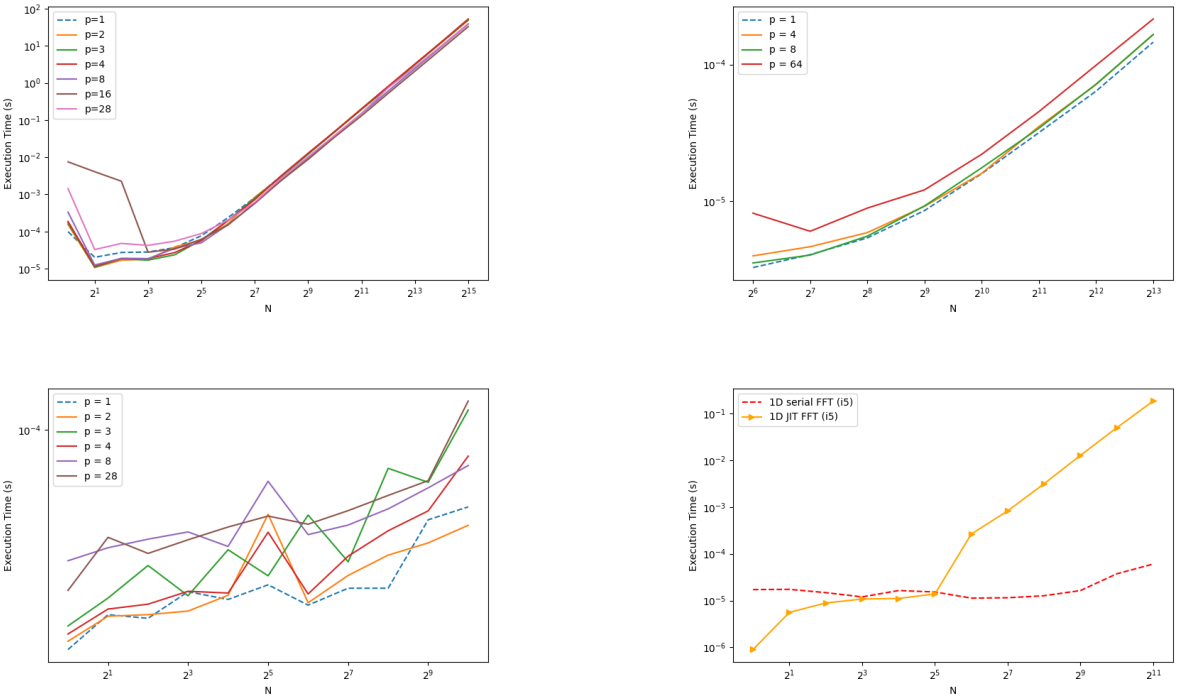

Figure 5. Scalability of parallelization methods for 1D FFT. Dashed line represents serial implementation. Top left: OpenMP; top right: MPI; bottom left: Multiprocessing; bottom right: JIT. OpenMP usage is tuned with dynamic scheduling on the BC4 hardware; MPI uses the BC4 hardware; Multiprocessing uses the i5 hardware; JIT is facilitated by Numba on the i5 hardware.

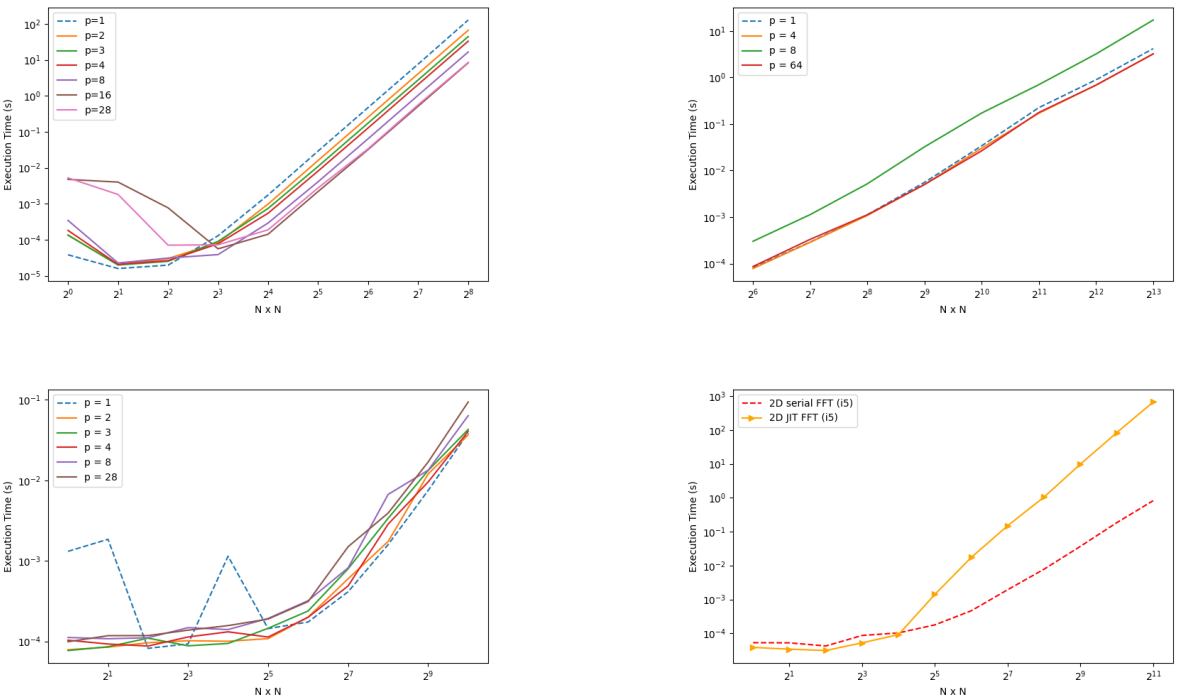

Figure 6. Scalability of parallelization methods for 2D FFT. Dashed line represents serial implementation. Top left: OpenMP; top right: MPI; bottom left: Multiprocessing; bottom right: JIT. OpenMP usage is tuned with dynamic scheduling on the BC4 hardware; MPI uses the BC4 hardware; Multiprocessing uses the i5 hardware; JIT is facilitated by Numba on the i5 hardware.

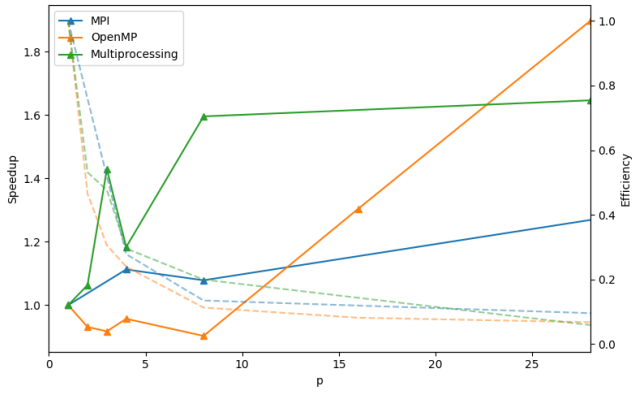

Figure 7. The speedup and efficiency of OpenMP, MPI and multiprocessing methods, against number of threads/processes for the 1D FFT computation. Solid line represents the speedup and dashed line represents the efficiency.

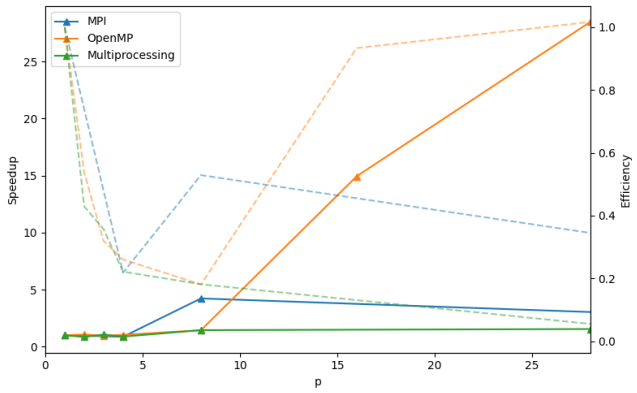

Figure 8. The speedup and efficiency of OpenMP, MPI and multiprocessing methods, against number of threads/processes for the 2D FFT computation. Solid line represents the speedup and dashed line represents the efficiency.

$$\text{Speedup (S)} = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \tag{7}$$

$$\text{Efficiency (E)} = \frac{S}{p} = \frac{T_{\text{serial}}}{p \cdot T_{\text{parallel}}} \tag{8}$$

Figures 7 and 8 present the speedup and efficiencies for OpenMP, MPI and multiprocessing in both the 1D and 2D FFT computation.

The speedup and efficiency evolution with p (number of threads/processes) is as predicted by Amdahl's Law which predicts diminishing returns proportional to the serial portion of the parallelized code. Speedup increases with p but returns diminish as p grows, largely due to the serialised matrix transposition that takes place within the FFT. For small problem sizes, speedup grows slowly due to overhead caused by communication/synchronisation dominating the execution. Efficiency therefore decreases sharply with p, as the workload per thread/process becomes too small to offset these overheads. The speedup and efficiency patterns illustrated in figures 7 and 8 underscore the dominance of overheads in creating bottlenecks that limit performance gains for all parallelization methods.

5 DISCUSSION

5.1 Multidimensional FFTs, Open-source Parallel FFT packages & Future Improvements

The implementation of parallelized FFT in this report demonstrates basic functionality, but struggles with the overhead caused by communication, synchronisation, and unparallelizable serial operations associated with MPI, OpenMP, and all FFT computations, respectively. The methods discussed also use standard all-to-all collective communication of contiguous memory, a notoriously costly communication type as this requires data realignment steps.

`mpi4py-fft` is an open-source, Python package specifically optimised for parallel, multidimensional, FFT computations (Dalcin et al. 2021). It is built on top of `mpi4py`, which was utilised for the distributed memory parallelization in this report. The package features specialised optimisations for data communication and memory handling, addressing the overhead and contention issues described previously.

The use of basic MPI communication patterns (e.g. `MPI_Scatter()`, `MPI_Gather()`) introduced significant overhead when communicating data for transpose-heavy FFTs, as discussed in Section 4.4. `mpi4py-fft` uses *pencil decomposition* - a technique that divides the data into smaller, contiguous arrays (i.e. 'pencils') across independent processes. This therefore minimises the IPC required during matrix transpose operations, alleviating the dominating communication overhead that was seen to bottleneck our performance.

Incorporating a 'pencil decomposition' element to the code would be essential in improving efficiency for distributed-memory systems, as it restructures data distribution to reduce communication volume and frequency during multi-dimensional FFTs, enabling better scalability for larger problem sizes and higher-dimensional FFTs.

6 CONCLUSION

Advancing efficient parallelization of the radix-2 Cooley-Tukey Fast Fourier Transform algorithm demands a focus on minimising communication overhead, optimising memory access, and well-planned load balancing. As a cornerstone of modern-day signal processing applications and beyond, the FFT requires scalable solutions to compute for higher-dimensions. Through analysing various methods of parallelization upon different memory and hardware systems, our analysis concludes that matrix transpositions present a continual challenge to achieving maximised parallelism.

REFERENCES

Amdahl G. M., 1967, AFIPS Conference Proceedings, 30, 483
Aziz Z. A., Abdulqader D. N., Sallow A. B., Omer H. K., 2021, Academic Journal of Nawroz University (AJNU), 10, 345
Cooley J. W., Tukey J. W., 1965, Mathematics of Computation, 19, 297
Dalcin L., Mortensen M., Keyes D. E., 2019, Journal of Parallel and Distributed Computing, 128, 137
Dalcin L., Mortensen M., Keyes D. E., 2021, Extreme Computing Research Center, King Abdullah University of Science and Technology
Farber R., 2011, CUDA Application Design and Development. Morgan Kaufmann
Gropp W., Lusk E., Skjellum A., 1999, Using MPI: Portable Parallel Programming with the Message Passing Interface. MIT Press
Harris C. R., Millman K. J., 2024, NumPy: The fundamental package for array computing with Python, https://numpy.org/doc/

McClanahan R., Sterling T., 2020, Parallel and High-Performance Computing. Manning Publications

Numba Development T., 2024, Numba: A High Performance Python Compiler, https://numba.pydata.org/numba-doc/dev/index.html

Orszag S. A., 1970, Journal of Fluid Mechanics, 41, 363

Python Software F., 2024, Python multiprocessing — Process-based parallelism, https://docs.python.org/3/library/multiprocessing.html

Saltz J., Berryman H., Wu J., 1990, Nasa contractor report, Multiprocessors and Runtime Compilation. Institute for Computer Applications in Science and Engineering, NASA Langley Research Center

Zhang J., Liu Z., Du Z., Wang J., 2021, Processes, 9, 1386