Parallel Fast Fourier Transforms for Shared and Distributed Memory Systems

Bertie Thorpe

School of Physics, The University of Bristol
12 December 2023

Abstract

Parallelisation of the Cooley-Tukey radix-2 Fast Fourier Transform algorithm was investigated for shared, distributed, and hybrid memory systems. Evaluating the performance on multi-core processors and multi-node clusters, the study addresses practical considerations such as cache misses, load balancing, problem scaling, and communication overhead. The findings offer valuable insights into designing efficient parallel FFT algorithms for both shared and distributed memory systems.

Figure 1: *The Fourier Transform of Paul Rand's famous "Eye-Bee-M" logo. The FFT algorithm was rediscovered by James Cooley and John Tukey whilst the former worked at IBM [1].*

Introduction

The Fast Fourier Transform (FFT) was the key that unlocked the digital age. Its role in accelerating the computation of the Discrete Fourier Transform (DFT) has profoundly impacted various fields, and as a consequence, the world at large [2]. In signal processing, the FFT enabled the swift analysis and manipulation of signals, ensuring real-time processing in telecommunications, audio engineering, and image processing. The FFT is also instrumental in efficient and powerful lossy data compression, reducing the bandwidth requirements in communications further. There is no doubt that the internet as we know it would not be feasible without this single algorithm [3].

Beyond traditional applications, the FFT finds widespread use in scientific and engineering simulations; for fluid dynamics, structural analysis, quantum mechanics, and electromagnetic simulations. With high performance computing approaching the exascale, data and computational demands continue to escalate [4]. The need for not only fast, but scalable FFTs has never been more pressing.

The traditional Cooley-Tukey FFT algorithm was borne from a time when the uniprocessor was king, and it performs and scales excellently under single core execution. [1] However, with the break down of Dennard scaling around 2006, CPU clock-speeds couldn't be reliably pushed any higher. Manufacturers resolved to move to multicore processor architectures, and so algorithms must adapt to this paradigm [5].

This paper investigates and analyses various parallelisation strategies for the Cooley-Tukey radix-2 FFT algorithm, aiming to optimise computation time and scalability on the University of Bristol's BlueCrystal Phase 4 supercomputer. Successful parallelisation is rarely trivial and is highly dependent on the structure of a given algorithm, the microarchitecture of the systems it executes on, and the communication overhead of the parallelisation approach.

Fast Fourier Transform

The Cooley-Tukey FFT algorithm is an efficient computation of the Discrete Fourier Transform, reducing the compute complexity from $O(N^2)$ to $O(NlogN)$

[6]. To understand how this is achieved, it is necessary to understand the structure of the DFT.

The DFT of a 1D vector, typically a time domain signal, decomposes the binned signal into its frequency-domain representation, described by

$$X_k = \sum_{n=0}^{N-1} x_n e^{\frac{-i2\pi}{N}kn}, \tag{1}$$

where X_k is the complex number representing the phase of the k-th frequency component in the frequency domain, x_n is the input signal at the n-th time sample, and N is the size of the input signal. The DFT essentially decomposes the signal into a sum of sinusoidal functions, each associated with a specific frequency. It is helpful to look at the DFT as

$$\vec{X} = \boldsymbol{W}\vec{x}, \tag{2}$$

where $\vec{X}$ and $\vec{x}$ are the column vectors of X_k and x_n values. $\boldsymbol{W}$ is known as the DFT square matrix and equates to ω^{nk} where $\omega = e^{\frac{-i2\pi}{N}}$. The DFT is then clearly seen as a large square matrix multiplication, indicating $O(N^2)$ computational complexity. When expanded, the DFT matrix for a 4-point input signal looks like

$$\begin{aligned} \boldsymbol{W} &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}. \end{aligned} \tag{3}$$

It can then be seen that thanks to the periodicity of the exponential N-th roots of unity, the number of unique 'twiddle' computations can be drastically reduced. Further, when the input is divided into odd and even components of the sequence, the smaller DFTs can be combined to find the global transform, utilising the symmetric twiddles. This division of the problem can be described by

$$X_k = \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N}(2m)k} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}, \tag{4}$$

where $n = 2m$ and $n = 2m+1$ are the even and odd input indices, respectively. Factoring out the common exponent, the two sums become DFTs of the even and odd indices:

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k. \tag{5}$$

By applying this principle recursively, very large DFTs can be reduced to a sequence of radix-2 DFTs. Once the radix depth is reached, a bit reversal reordering of the input indices is calculated to efficiently order the butterfly operations, and the output is recombined following the data flow diagram in figure 2. Thus achieving a computational complexity of $O(NlogN)$.

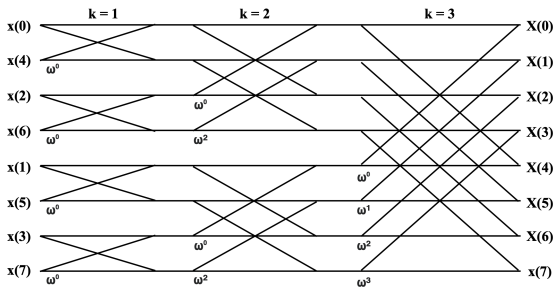

Figure 2: *The data flow diagram for the butterfly operations of an 8-point 1D FFT. k here is the stages of the butterfly recombinations.*

For the case of the 2D Fast Fourier Transform, the theoretical algorithm is trivially extended from the 1D FFT, as the dimensions are perfectly separable. 1D FFTs are calculated for each of the rows, and then similarly calculated for each of the columns of the 2D input array. The naive 2D square DFT complexity is $O(N^4)$, by the FFT this can be reduced to $O(N^2 logN)$. In practice however, the 2D FFT has data access limitations and so the performance is dependent on that. This will be discussed below.

Serial Implementation

The FFT implementation makes considerable use of contiguous memory which ensures better cache locality. When data is stored consecutively in memory, it increases the likelihood that adjacent data elements are fetched together into the cache. This is crucial for performance because accessing data in cache is significantly faster than fetching data from main memory. Matrix calculations often involve repeated access to neighbouring data points, making contiguous memory advantageous for exploiting cache efficiency[7].

A number of optimisations to the serial Cooley-Tukey algorithm can be made before the parallel models are implemented. Due to the repeated use of the twiddle factors at each stage of the butterfly operations and their expensive trigonometric calculation cost, these should be precomputed. The savings from this are significant for the 1D FFT but exceptional for the 2D FFT, where a vastly reduced amount of runtime twiddles need to be calculated due to them being identical for each row and column, as their values are based off the size of the input array.

Likewise, the input bit reversal reordering can be partially precomputed by calculating the bit reverse order of the input index and using this as an index for the actual bit-reverse swapping [8]. This makes no difference for the 1D case as the input reordering is only done once, but the 2D case will reorder every row and column based off this index. The proportion of these sub-algorithms against the actual butterfly operations can be seen in figure 3, where the optimised versions have been implemented, showing that they aren't insignificant and must not be overlooked when optimising for compute efficiency.

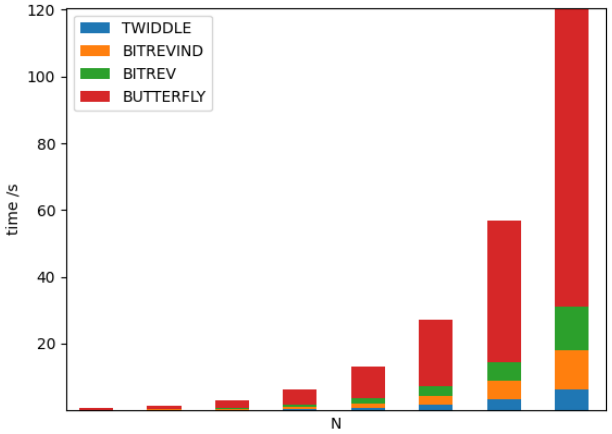

Figure 3: *The compute times for components of the serial 1D FFT, scaled with problem size. Precomputed twiddle factors have minimal impact on the overall compute time. The bit reverse indices calculation is shown separate from the bit reverse swapping function because the former will be precomputed for the 2D FFT.*

Having said this, there is a tradeoff to be made when precomputing values, as the speedup gained by not computing on-the-fly may be lost when fetching the values from memory. This may inhibit problem size scaling as values which don't fit in the memory will result in frequent and significant cache misses. Fetching data from the RAM is slow. For this reason it is preferable to keep the problem sizes small enough so they fit within the cache as much as possible. Most data calculations are performed in place for this reason [9].

The 2D FFT requires particular attention to the memory access of data. C adopts row-major indexing of arrays, so when the program proceeds to take the FFTs of the columns, the memory access time is greatly increased by striding the memory. A solution to this is a single transpose of the data matrix, so that the FFT of the columns can be accessed in row-major order. A final transpose to reorient the data is optional, though it has been implemented here. The matrix transpose complicates the parallelisation in distributed systems, which will be discussed.

Parallelisation

The parallelisation of the Cooley-Tukey algorithm will largely adhere to two approaches:

1. Shared Memory.
2. Distributed Memory.

For shared memory multi-core processors, the OpenMP API is used. For distributed memory systems, MPI (Message Passing Interface) is used.

Shared Memory - OpenMP

Shared memory parallelisation is possible thanks to the fast cache shared between cores on each processor socket. For BlueCrystal, this equates to a 35 MB L3 cache shared between 14 cores on the Broadwell Xeon E5-2680 CPUs. Each node on BlueCrystal contains 2 of these CPUs in a dual socket configuration. They are connected by a high bandwidth, high throughput bus, allowing shared memory parallelisation with OpenMP to extend to all 28 cores on the node, with minimal bus latency and bottleneck.

OpenMP adopts a fork-join model of parallelisation. The code follows the serial control flow on a master thread, until it reaches a parallel region. OpenMP then spawns worker threads and completes the parallel code before synchronising the threads, collapsing the threads, and returning to a serial control flow again. This synchronisation stage is the root of most load imbalances in the computation with OpenMP if the parallel code blocks are not distributed evenly across threads. The benefit of OpenMP's fork-join model is it can be implemented without disrupting the control flow of otherwise serial code, simplifying implementation and reducing development time [10].

The most significant advantage of OpenMP over a distributed memory system, is there are no costly communication overheads from message passing, though the one caveat to this is there are overheads associated with data access. Only one thread can access the same memory address at a time. While the overhead for OpenMP data access is significantly lower than message passing, it can still prove costly if excessive loop level recursion is employed.

Fortunately, the FFT algorithm can be designed in a way to separate most of the computation into successive loops which are suited to the fork-join model. Similarly, the butterfly operations can be mapped from the beginning and calculated iteratively rather than recursively. This sacrifices the flexibility of input size normally associated with divide-and-conquer recursion, but the tradeoff for optimal and predictable cache locality is well worth it, and quite necessary for effective parallelism.

Concerning the shared memory 2D FFT: using OpenMP with the collapse clause is generally not recommended for an in-place transpose operation. The collapse clause is typically used with nested loops to combine them into a single loop, but in the case of in-place transpose, you need to be careful about the order in which elements are accessed and updated. The reason is that the transpose operation involves swapping elements across the main diagonal, and the order of these swaps matters. Using collapse might lead to incorrect results because the combined loop might not preserve the order needed for an in-place transpose.

Distributed Memory - MPI

MPI (Message Passing Interface) is a message-passing standard for parallelisation on distributed memory architectures. Distributed memory parallelisation relies on communication between processor ranks. In theory this can scale to however many nodes available, the reality however is not so trivial. Successful distributed memory parallelisation depends on problem size, specific algorithms, and if there are any bottlenecks in the code that cannot be parallelised. These bottlenecks become limiting factors according to Amdahl's law, which states that the overall improvement in parallel computation is fundamentally limited by the fraction of the task that remains sequential. It underscores the importance of optimising the critical path to achieve meaningful performance gains.

2D FFTS are far easier to parallelise due to their nearly complete separability, only requiring two gathers and transposes in the complete FFT calculation. The major contention with this however, is as mentioned above, the matrix transpose itself. Whilst the FFT butterfly operations are embarrassingly parallel, the matrix transpose is all communication dominated overhead. Practically no ALU computation is performed for transposes as they are merely moving data between memory addresses. Further, to parallelise the transpose a global, point-to-point, all-to-all communicator is required. This is an expensive communicator[11].

Single program, multiple data (SPMD) implementations of 1D FFTs also require complex and extensive global communication patterns which can create excessive overhead. For large enough N for 1D data, it is not necessary to transpose and re-scatter the data for further parallel butterflies. Though not a computationally optimal method, it is suitable to finish the FFT butterfly operations on one thread because there are only $log(p)$ remaining steps, compared to $log(N/p)$ steps for the first part of the FFT [12][13]. It may even be preferable because of the significant global communication required for this last transpose if it were to become fully parallelised for every stage of the FFT, though this is unlikely to scale very well for HPC level problem sizes and parallelism.

Hybrid

The most significant advantage of OpenMP is also its downfall. The efficiency of the shared memory model is ultimately bounded by the size of the single multicore processor (or node). Increasing the number of cores on a CPU results in increased amount of cores that need to access data in the shared memory, access which is staggered if multiple cores are accessing the same data. And this is before considering the physical limitations of CPU size when concerned with heat dissipation and power consumption.

It is desirable to be able to combine the shared memory speeds of OpenMP with the theoretical scalability of MPI, potentially allowing fine-grained load balancing, at the loop level, and higher message passing level. For large datasets that require distributed-memory parallelisation, the combination of OpenMP and MPI can provide a powerful solution. The shared-memory parallelisation within each node (using OpenMP) ensures efficient computation, while MPI allows for communication between nodes, enabling the processing of datasets that exceed the capacity of a single node's memory.

Results

1D FFT

Figure 4 shows the 1D FFT performance on BlueCrystal, limited to a single node. As expected the serial implementation performs faster at small problem sizes, below N=8192 as it doesn't have to contend with communication overhead. For both OpenMP and MPI, eight threads/ranks proved to be the most stable at lower FFT sizes, the more threads there are, the more synchronisation conditions there are, particularly at lower problem sizes, as load balancing is affected drastically by otherwise small synchronisation delays.

The OpenMP implementation produces the best scaling, overtaking the serial code performance at N=8192 whereas the MPI implementation doesn't overtake the serial performance until N=2^{22}, when both are compared at eight processes. This too is expected because the benchmark only runs on one shared memory node. MPI's message passing communication overhead is far greater than OpenMP's thread synchronisation and data access overhead.

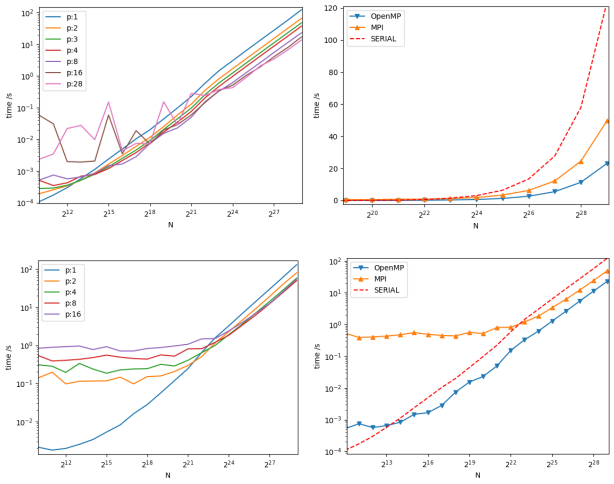

Figure 4: *Performance of the 1D FFT. Upper left: OpenMP size scaling performance with different number of threads. Lower left: MPI size scaling performance with different number of ranks. Upper right: linear time performance of both implementations against serial time, using 8 processes. Lower right: log time performance of both implementations against serial time, using 8 processes.*

2D FFT

Figure 5 shows the equivalent performance metrics of the 2D OpenMP and MPI implementations on a single node.

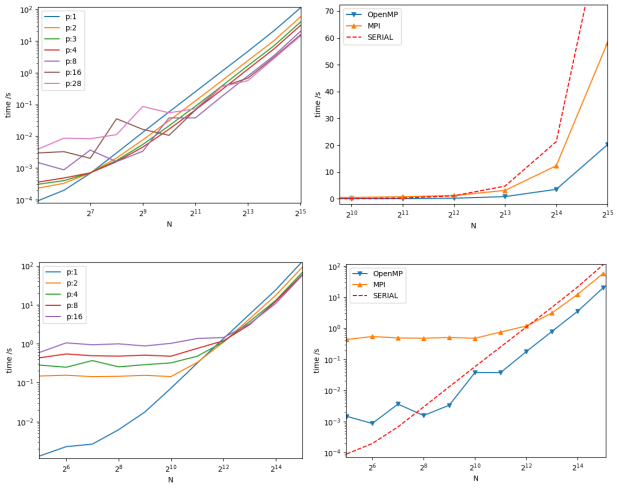

Figure 5: *Performance of the 2D FFT.Upper left: OpenMP size scaling performance with different number of threads. Lower left: MPI size scaling performance with different number of ranks. Right: 8 processes. Upper right: linear time performance. Lower right: log time.*

Likewise to the 1D implementations, the serial code performs faster at low N, and OpenMP overtakes the serial code earlier and provides better scaling at large problem sizes. The OpenMP code overtakes the serial code at N=256, whereas MPI doesn't overtake until N=4096. The MPI code also doesn't perform much better than serial at large N. This is most likely due to the serialised matrix transpose, which scales at $O(N^2)$, creating a ceiling for further compute optimisation.

One interesting thing to note is the improved stability at low problem sizes for the 2D FFT over the 1D FFT. The individual FFT rows are far smaller than the 1D FFT, and so require far less bit reverse indices and twiddle factors. Having been precomputed, these values are fetched from memory each time they are called. The smaller array sizes would allow for better cache locality and coherence, resulting in fewer unpredictable delays in memory access.

Speedup Efficiency

Figure 6 Compares the speedup efficiency of all the single node FFT implementations against the serial code performance, using 8 processors. The 2D parallel implementations display drastic efficiency improvements with scaling problem size. It takes the MPI code a significant amount of input size before it starts improving upon the serial performance. However the OpenMP implementations display great instability in their performance, which can be attributed to the synchronisation scheduling.

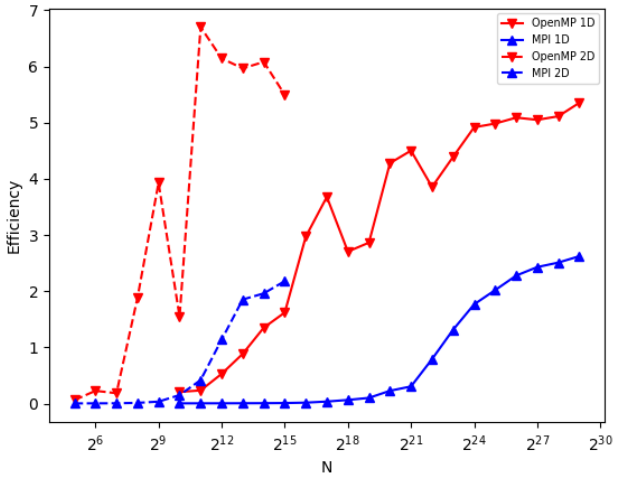

Figure 6: *The speedup efficiency is compared for each of the implementations at the optimal 8 process performance against the respective serial implementations.*

Hybrid Cluster

Figure 7 shows the 2D FFT performance across 4 nodes, utilising MPI and a hybrid MPI/OpenMP model with 4 MPI ranks, one for each node, and 16 OpenMP threads within each rank. Successful scaling is observed for the hybrid model, though the scope for further node scaling is inhibited most likely by the MPI ranks, and the fact the matrix transpose is performed serially on the pure MPI code and only

on one node for the hybrid code. Having said that, the efficiency gain in the hybrid model is significant enough to warrant further research.

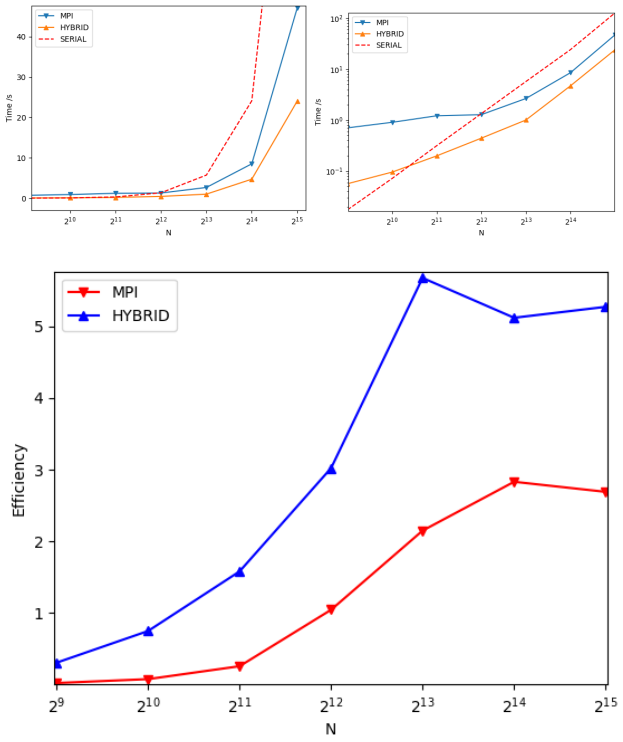

Figure 7: *Four node performance of MPI and Hybrid OpenMP/MPI 2D FFT across 64 tasks. Upper left: linear process time. Upper right: log process time. Below: the compute speedup over the serial FFT.*

Future Improvements

There are a number of limitations to the above implementations, particularly concerning the poor scaling to higher FFT sizes. To be able to make full use of large scale multi-node clusters, this poor scaling needs to be addressed.

MPI Transpose The chief culprit causing the poor scaling is the MPI transpose. Parallelising the transpose can be achieved by block-wise communicators, by recursively subdividing the input matrix into four subarrays, transposing the local subarrays and assigning the subarrays with an MPI type creation, then mapping point-to-point communication for the global transpose [11]. This is an expensive communication but scales well, as you wouldn't have to calculate the very expensive $O(n^2)$ transpose on one core (or one node if using hybrid system). A similar MPI Alltoall communicator can be used for the 1D FFT too, 'transposing' the ranks' array allocation among processors. This should reduce the serial execution of the FFT for the final $log(p)$ stages of the butterflies [14].

3D FFT The relevance of HPC scale 2D FFTs is debatable. Most use cases of 2D FFTs are for image processing which have rare need for exascale performance. 3D FFTs however require significantly more computational power, and there is still need for bigger sized models. Fluid and molecular dynamics simulations are underpinned by their FFT libraries, with a lot of room for parameter tuning[4].

Conclusion

Parallelising the Cooley-Tukey radix-2 FFT for various memory systems requires careful tuning, emphasising cache optimisation, load balancing, and communication efficiency. The FFT's pivotal role in signal processing, data compression, and scientific simulations, especially in the era of exascale computing, underscores the need for fast and scalable FFTs. With an in-depth exploration of the memory access considerations and tradeoffs in precomputing values, it is evident that successful parallelism is held back by global memory communication for transposing matrices.

References

[1] Michael T Heideman, Don H Johnson, and C Sidney Burrus. "Gauss and the history of the fast Fourier transform". In: *Archive for history of exact sciences* (1985), pp. 265–277.

[2] James W Cooley, Peter AW Lewis, and Peter D Welch. "Historical notes on the fast Fourier transform". In: *Proceedings of the IEEE* 55.10 (1967), pp. 1675–1677.

[3] G Ganesh Kumar, Subhendu K Sahoo, and Pramod Kumar Meher. "50 years of FFT algorithms and applications". In: *Circuits, Systems, and Signal Processing* 38 (2019), pp. 5665–5698.

[4] Andrei Gorobets et al. "Hybrid MPI+ OpenMP parallelization of an FFT-based 3D Poisson solver with one periodic direction". In: *Computers & fluids* 49.1 (2011), pp. 101–109.

[5] Robert H Dennard, Jin Cai, and Arvind Kumar. "A perspective on today's scaling challenges and possible future directions". In: *Handbook of Thin Film Deposition*. Elsevier, 2018, pp. 3–18.

[6] James W Cooley and John W Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[7] Markus Kowarschik and Christian Weiß. "An overview of cache optimization techniques and cache-aware numerical algorithms". In: *Algorithms for memory hierarchies: advanced lectures* (2003), pp. 213–232.

[8] Christian Knauth et al. "Practically efficient methods for performing bit-reversed permutation in C++ 11 on the x86-64 architecture". In: *arXiv preprint arXiv:1708.01873* (2017).

[9] Jeffrey J Rodriguez. "An improved bit-reversal algorithm for the fast Fourier transform". In: *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*. IEEE Computer Society. 1988, pp. 1407–1408.

[10] Franz Franchetti et al. "Discrete Fourier transform on multicore". In: *IEEE Signal Processing Magazine* 26.6 (2009), pp. 90–102.

[11] Lisandro Dalcin, Mikael Mortensen, and David E Keyes. "Fast parallel multidimensional FFT using advanced MPI". In: *Journal of Parallel and Distributed Computing* 128 (2019), pp. 137–150.

[12] Ramesh C Agarwal, Fred G Gustavson, and Mohammad Zubair. "A high performance parallel algorithm for 1-D FFT". In: *Supercomputing'94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. IEEE. 1994, pp. 34–40.

[13] David Culler et al. "LogP: Towards a realistic model of parallel computation". In: *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*. 1993, pp. 1–12.

[14] Rami Al Na'mneh, W David Pan, and Seong-Moo Yoo. "Efficient adaptive algorithms for transposing small and large matrices on symmetric multiprocessors". In: *Informatica* 17.4 (2006), pp. 535–550.