Parallel Fast Fourier Transform Literature Review

Ben Karsin

December 15, 2013

Abstract

The fast Fourier transform (FFT), developed in 1965, is widely considered one of the most prolific and significant algorithms of the 20th century. While the classic Fourier Transform algorithm converts data from time-domain to frequency domain in $O(N^2)$, the FFT reduces the complexity to $O(N\log N)$. This performance improvement has had a significant impact on many fields and has revolutionized the areas of digital signal processing and numerical convolution.

Since the FFT was introduced, a wide range of research has attempted to leverage and improve on the idea. One of the largest trends was in attempting to parallelize the FFT to further improve its performance. Despite the wealth of research, the basic FFT algorithm remains mostly unchanged over the near half-decade since. This paper presents an overview of the research centered around the Fast Fourier Transform, with a focus on parallelization. We review the works that led to major contributions to the field and discuss how the field has evolved over the years.
1 Introduction

In 1811 Joseph Fourier submitted a paper to the French Academy of Sciences. While the paper was not immediately published, it had an impact and, unbeknownst to Fourier, would have a lasting impact on a range of problem domains, many of which did not yet exist. The paper presented by Fourier introduced the concept of the Fourier Transform, a method of transforming a finite set of evenly-distributed samples from its original domain (e.g., time) to the frequency domain. The mathematical concept of the Fourier transform is applied to the infinite set of complex numbers and involves integration. This continuous type of Fourier transform has many applications in physics and engineering, but a Discrete form of the Fourier transform exists that can be easily performed by computer systems [53]. This Discrete Fourier Transform (DFT) and its uses in Computer Science led to the advent of the Fast Fourier Transform (FFT) in 1965 by James Cooley and John Tukey [18]. Using a divide-and-conquer method, FFT reduces the computational complexity of the DFT from $O(N^2)$ to $O(N \log(N))$. With such a significant decrease in computational complexity, the FFT made the Fourier Transform a viable solution to many difficult problems. Due to the implications of the FFT on the areas of digital signal processing and numerical convolution, it is considered one of the most influential algorithms of the 20th century [22]. Because it is utilized by so many applications and systems, the FFT is one of the most commonly utilized algorithms by modern computers [22].

Modern computers have become exponentially faster over the past several decades, allowing us to solve larger and more complex problems. However, some large-scale problems still present difficulties. Even with the $O(N \log(N))$ complexity of the FFT, some large-scale problems remain infeasible with modern computers. One way to overcome computation deficiencies is through the use of parallelization. By using many computers to solve a problem in parallel, we can greatly decrease the overall computation time. This is an increasingly commonplace strategy with the prevalence of multi- and many-core architectures. The distributed nature of the FFT algorithm makes it an obvious candidate for parallelization. Since the introduction of the FFT algorithm in 1965 [18], its parallelization has been (and remains) a rich area of research.

This survey of the Parallel Fast Fourier Transform is organized as follows: in Section 2 we present general information about the FFT algorithm and the most prominent variants. Section 3 contains general information about the most widely adopted parallel FFT algorithm. In Section 4 we outline the development of the various parallel FFT algorithms by the corresponding landmark papers. In Section 5 we discuss the pro’s and con’s of the various methods and present opinions on current and future research areas. Finally, in Section 6 we present conclusions.
The Sequential Fast Fourier Transform Explained

The Discrete Fourier Transform is an operation performed on a series of elements to convert the underlying domain (e.g., time) to frequency, or vice-versa. The result has many useful applications and is one of the most widely used algorithms of the 20th and 21st centuries [22]. The typical DFT operation performed on \( N \) elements \( x_1, x_2, \ldots, x_N \) is defined as:

\[
X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}
\]

Every element of the result array \( X \) requires an additive contribution of every element of the input array \( x \). Thus, the DFT operation on an \( N \)-element input array is \( O(N^2) \). This becomes problematic when trying to compute the DFT of a large array, which is the case for many applications. Therefore, reducing the time-complexity of the DFT algorithm is desirable.

The Fast Fourier Transform algorithm, introduced in 1965 by Cooley and Tukey [18], utilizes the divide-and-conquer approach to substantially reduce the complexity of computing the Discrete Fourier Transform. Using the FFT method, we are able to compute the DFT of any input array in \( O(N \log N) \). The original paper that introduced the FFT is very brief [18], but its impact was immediately apparent, particularly in the field of numerical convolution. Several papers published shortly after present more detail and further generalize the FFT algorithm [28]. However, for many years after its introduction, few technical advances were made to the "Cooley-Tukey algorithm", and most research was focused on analyzing the details of the FFT algorithm or theoretical propositions of FFT variants.

2.1 The Cooley-Tukey Algorithm

The Cooley-Tukey algorithm, as introduced in [18], uses a divide-and-conquer approach to reduce the computation (both additions and multiplications) needed to calculate the Discrete Fourier Transform. The original Cooley-Tukey algorithm, outlined here, is radix-2 DIT (decimation-in-time) and serves as our baseline for the general FFT algorithm. The basic steps of the algorithm are:

1. **Decimate** - Split the original input into even or odd sets, creating two (i.e. radix-2) smaller DFTs to solve.

2. **Multiply** - Multiply each element by its complex root of unity (called twiddle factors [28]).

3. **Butterfly** - Add each element of each of the smaller DFTs with a corresponding element of the other (see Figure 1).
Figure 1: Diagram of the recursive process of a radix-2 DIT Cooley-Tukey FFT algorithm. Example calculates the DFT of an 8-element array in 3 \((\log(n))\) recursive stages, with 8 \((n)\) addition and multiplications operations at each stage. Note that the output array \((B)\) is not in the correct order (the indices of the output array elements are jumbled) and must be re-ordered to generate the correct DFT result.

4. **Recurse** - Repeat from step 1 for each new DFT.

When using decimation-in-time, each DFT is split (decimated) by even and odd input (time) indices. Twiddle factors are data-independent trigonometric constants that are multiplied at each stage of recursion. They are determined by the data size, radix, and recursion stage. The "Butterfly" pattern at each stage of recursion is outlined in Figure 1.

As Figure 1 illustrates, the pattern of element pairings at each stage (along with the correct twiddle factors) allows us to calculate the DFT in just \(\log(n)\) recursive stages. At each stage, the algorithm adds the results of the two smaller DFTs (based on even and odd indices), multiplied by the corresponding twiddle factor. The key point of the Cooley-Tukey algorithm is that the dissemination at each stage results in each element contributing to the final product of every resulting element. Due to the distributive properties of multiplication and addition and use of twiddle factors, the FFT is able to shortcut the N-element summation used to determine the DFT of each element described above.
One significant downside to the Cooley-Tukey FFT algorithm is that it jumbles the order of the elements. Either the input array \(A\) or the output array \(B\) must be re-arranged (sorted) to get the correct DFT result. Sorting is a rich research field of its own and there are many different methods used to rearrange the input, the best of which are (in the general case) \(O(n \log n)\). Note that, since the FFT and sorting stages have the same computational complexity, the overall operation remains \(O(n \log n)\). Research in the area of sorting is beyond the scope of this paper, except where algorithms combine the FFT algorithm with the sorting stage to improve overall performance. We refer the interested reader to well-known sorting algorithms [19].

2.2 Other Sequential FFT Algorithms

All variants of the Cooley-Tukey FFT algorithm use the same divide-and-conquer approach to calculating the DFT, though the details of operation are different. Below are several common variations of the basic Cooley-Tukey FFT algorithm, along with the advantages/disadvantages of each.

- **Prime-factor**: Originally proposed by [30] and cited by [18]. Only functions on certain input sizes, but requires no twiddle factors.

- **Decimation-in-frequency (DIF)**: Introduced by [28], the algorithm is essentially performed in reverse compared to decimation-in-time (DIT). DFTs are divided by odd and even indices of the output (frequency). There are several variations of DIT and DIF FFT algorithms that result in different input/output array orders, although these play no significant role in the focus of this paper.

- **Mixed-radix**: Proposed by [52]. Based on the input size, different stages of recursion utilize different radii. This results in a reduced number of recursion steps. However, the algorithm must be tailored to the specific data input size.

- **Split-radix**: Involves decimating the DFT into unequally-sized problems, using different radii and is discussed in detail in Section 4.1.1. The algorithm has the same computational complexity \(O(n \log n)\), but requires as little as half the number of multiplications as Cooley-Tukey [64, 23].

Many of these variations can be combined (e.g., Split-radix DIF), leading to many different Cooley-Tukey-based FFT algorithms. In addition to the above variants of the traditional Cooley-Tukey FFT algorithm, there are several different algorithms that achieve comparable performance in computing the DFT of a given input. Each the following sequential algorithms provides some benefits and drawbacks, compared to Cooley-Tukey.

- **Bruun’s FFT algorithm** [14]: A similar process to Cooley-Tukey but utilizes different coefficients at each stage, with only real coefficients until the last stage of computation. Bruun’s algorithm was proposed to apply
more easily to real data, but has been shown to be inferior in simplicity and accuracy to Cooley-Tukey.

- **Rader’s FFT algorithm [48]**: Computes the DFT of only prime sizes by re-expressing it as a convolution. Traditional radix-2 Cooley-Tukey, however, does not function on prime sizes. This algorithm has been applied to other transforms (e.g., Hartley transform) but is not heavily used as it only functions on prime sizes.

- **Winograd’s FFT algorithm [62]**: An extension of Rader’s, is able to compute DFTs with a greatly reduced number of multiplications (O(N)). However, the greatly increased number of additions makes it infeasible for large input sizes.

- **Bluestein’s FFT algorithm [12]**: Uses a similar technique to Rader’s algorithm, but functions on any size input. However, it is less efficient than traditional Cooley-Tukey. Like Rader’s, Bluestein’s algorithm has several other applications as well.

The details of each of these sequential algorithms (or variants) is outside the scope of this paper, as the focus is the parallelization of FFT algorithms.

### 3 Parallel Fast Fourier Transform

In the previous section, we explained how the various flavors of the Fast Fourier Transform algorithm are able to reduce the computational complexity required to compute the DFT of an input from O(n^2) to O(n log n). This has had a significant impact on many fields and revolutionized the areas digital signal processing and numerical convolution [22]. However, as technology continues to advance and data becomes more dense, more problems are limited by the time and memory required to compute the FFT. Additionally, many applications operating on large datasets require the use of distributed-memory systems, providing another argument for parallelization. Examples of these large-scale FFT applications include:

- **3D graphics processing** - many graphics processing algorithms utilize the FFT. As graphics become more detailed and complex, the demand for high-speed FFT processing is required. A good deal of research has been done in this field, including efficiently applying the FFT to graphics processing units (GPUs) [17]. We discuss this area of research in Section 4.4.1

- **Large-scale spectral transforms** - Spectral transformation is used to analyze a wide range of data (e.g., image, electromagnetic, radiation). As measurements become more dense and precise, the scalability of this process becomes more important [20],
• Data compression - Many efficient data compression algorithms (e.g., JPEG) utilize the FFT. Formatting and compression is often needed to be done on-the-fly on increasingly large files (e.g., images, sound files) [17], and

• Polynomial Multiplication/Convolution - multiplication of arrays and polynomials can be an inefficient \(O(N^2)\) operation that can be improved with the FFT. By transforming the arrays with the FFT, we can perform the multiplication element-wise (i.e., in linear time) before transforming back to the original domain with the FFT. For large-scale arrays, this alternative can significantly decrease the time required [17].

Clearly, it has become important to leverage all resources to tackle these large-scale problems. For the past several decades, parallel processing has been increasingly helpful in achieving greater performance from existing computer hardware. Since the 1960s, every top supercomputer has utilized multi- and many-core technology [2]. Today, most commodity computers are multi-core, allowing users to achieve performance gains through parallelization.

3.1 Inherent Parallelism of the FFT

One key aspect of the Fast Fourier Transform is that it operates on large data structures (i.e., arrays). Conveniently, this lends itself to easy parallelization: we simply assign a portion of the input data to each thread/core. In the case of shared-memory systems, this allows for almost perfect parallelization [39, 55]. However, large-scale FFT problems typically exceed the memory capacity of most systems. Therefore, the majority of research focuses on distributed memory systems, where each node holds a specific subset of the input data and communicates over a network to other nodes when needed. Due to the complexity of the Butterfly stage of the FFT, this communication can be time-consuming. Figure 2 illustrates the communication pattern of an 8-node distributed-memory system running a radix-2 DIT FFT algorithm, where each node has 1 element of the input array (similar communication patterns are seen when each node contains a subset of larger arrays).

As seen in Figure 2, the Butterfly pattern requires communication from varying pairs of nodes. Note from the patterns in Figure 2 that, at each stage \(s\) of the radix-2 algorithm, each node communicates with the node at index \(+/- 2^s\). This makes the FFT algorithm perfectly suited for the Hypercube network topology [13], which we discuss in further detail in Section 4.2.1.

Despite being seemingly well-suited for parallelization, parallelizing FFT computation over a distributed-memory system introduces some difficulties. FFT, being a relatively data-algorithm \(O(n \log n)\) means that memory management and communication may become bottlenecks and reduce parallel efficiency. Several strategies have been developed to overcome these difficulties, as detailed hereafter.
3.2 Memory and Communication

Distributed-memory parallel systems are capable of providing much more resources (CPU, memory, storage) than traditional systems, but leveraging those resources can prove difficult. In addition to the communication phases required by the FFT algorithm (see Figure 2), distributing memory and unscrambling data provide an added communication requirement. Thus, the typical parallel version of the Cooley-Tukey algorithm is considered as the 6-step framework shown in Figure 3 [21]. Since, with current (and past) technology, network communication speed is far slower than memory access, the bottleneck typically lies in the 3 communication phases seen in Figure 3.

The 3 communication phases seen in Figure 3 are as follows: bit-reversal, FFT, and memory distribution (labeled cyclic to block) and are labeled as stages 1, 3, and 6, respectively. As discussed in Section 2.1 either the input or output array must be sorted as part of the Cooley-Tukey algorithm. Thus, the first communication phase (bit-reversal) involves a global re-ordering. The second phase is the typical FFT algorithm, with its required communication at each recursive stage. Finally, the third communication phase (stage 6 of Figure 3) is required to collect the results for further use. Several variations to this method
have been proposed that reduce the cost of the first and third phases, but the parallel FFT algorithm remains communication bound. Recent advances, such as the Fast Multipole Method [24], detailed in Section 4.3.3, attempts to reduce communication time using approximation and increased computation.

3.3 Legacy of Research

The significance and inherent parallelism of the Fast Fourier Transform has prompted a rich body of research spanning decades. Starting immediately after the introduction of the FFT algorithm, parallel FFT research has continually been a rich field of research.

From the theoretical research of the '60s and '70s [45, 9, 39] to the supercomputer and network topology research of the late '80s and '90s [55, 56, 47] to the large-scale application and hardware research of the 2000s and 2010s [16, 7, 50], for the past four decades the parallel FFT field has been continually evolving and growing. In the next section, we introduce the more significant ideas and papers in the development of the parallel (and sequential) FFT.
4 Algorithmic Development

In this section, we present the major contributions to the field of parallel FFT computation over the past half-century. We demonstrate how the field has evolved since its inception, in conjunction with other fields. Figure 4 illustrates how the contributions of the papers discussed in this section are related. Figure 4 is divided into 4 columns, indicating the general focus of each paper and how the trends of the field changed over the years. Each of the colored regions in Figure 4 represents an era of research, and is described in detail in its corresponding section (as labeled). A chronological list of these publications and their respective contributions/weaknesses are presented in Table 1 at the end of this paper.

4.1 Preliminary FFT Work in the ’60s and ’70s

After the introduction of the sequential FFT algorithm in 1965 [18], research on parallelization techniques involving the FFT immediately began. In 1968 (just 3 years after the Cooley-Tukey paper), the first parallel FFT paper was introduced, outlining a parallel FFT algorithm and proposing hardware to implement it [45]. Though it did not present any empirical data, the idea gained some popularity and it opened the field of parallel FFT algorithms. Shortly after, Bergland published two theoretical papers that extended the idea of a parallel Fast Fourier Transform algorithm [9, 10]. [9] demonstrated that the FFT could be applied to a parallel shared-memory system. Three years later, [10] introduced an implementation of such a parallel algorithm, including details of mapping addresses to processors and correctly ordering the input/output arrays.

Concurrently, important research was being done in the field of sequential FFT algorithms. [52] introduced the mixed-radix algorithm and [64] published preliminary work on a simplistic variant of the split-radix FFT algorithm. These papers helped lead to the work in [23] that introduced the final Split-radix algorithm (and coined the term), which remains one of the most efficient and widely-used sequential FFT algorithms [17].

4.1.1 Split-Radix FFT

The central concept behind the Split-Radix algorithm is to decimate the DFT into several DFTs of different size at each stage. Specifically, each DFT (of size $N$) is split into 3 smaller DFTs of size $N/2$, $N/4$, and $N/4$. Figure 5 illustrates the concept behind the Split-radix algorithm.

Recall from Section 2 that a DFT operation performed on $N$ elements is defined as:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi k \frac{n}{N}}$$
Figure 4: Major papers that contributed to the field over the past half-century. Papers are placed in one of the 4 columns based on the general focus of the research. Each region represents an era of research and is discussed in its corresponding section. More details about each paper are found in Table 1. Red-bordered nodes represent papers that focus on sequential FFT algorithms, while blue-bordered nodes involve parallelism.
Figure 5: Diagram illustrating the Split-radix FFT algorithm. Even elements are split by radix-2, while odd elements are split by radix-4, resulting in different-sized DFTs. The algorithm typically reduces the number of multiplications and additions by 20% over the standard Cooley-Tukey algorithm.

The Split-radix function expresses this as the following 3 summations:

\[ X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} e^{-i2\pi \frac{nk}{N/2}} + e^{-i2\pi \frac{k}{N}} \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} e^{-i2\pi \frac{nk}{N/4}} + e^{-i2\pi \frac{3k}{N}} \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} e^{-i2\pi \frac{nk}{N/4}} \]

The justification behind this decomposition is that the first summation of even elements has the advantage of not requiring a multiplication using radix-2, while odd elements (the 2nd and 3rd summations) benefit from a radix-4 decimation. Figure 5 illustrates how this decimation occurs at each stage of the Split-radix algorithm.

The advantages of Split-radix over the classic Cooley-Tukey algorithm are significant. While a general lower-bound for the number of operations of a classic FFT algorithm is not known, the Split-radix algorithm requires \(4N\log N - 6N + 8\) real additions and multiplications. In practice, this is a 20% decrease in the average number of operations required by the classic radix-2 Cooley-Tukey algorithm [23]. The Split-radix algorithm has since been improved slightly [33], reducing the required operations to approximately \(\frac{3}{4}N\log N\).

The major drawback of the Split-radix algorithm is that input sizes must be a multiple of 4. However, since it is a recursive process (like Cooley-Tukey), it can be combined with any other FFT algorithm. Split-radix remains the algorithm with the lowest average number of operations that can be practically applied. As such, it is widely used in many domains [27, 10, 33].
4.2 Early Parallelization in the ’80s

In the mid 1970s, shared-memory systems began to increase in popularity with the availability of vector machines. Datasets were generally still quite small and distributed memory had yet to gain wide-spread popularity, so early parallel FFT works generally involved shared-memory systems. The works of [39], [55] and [47] all presented empirical results for Cooley-Tukey-based parallel FFT algorithms running on shared-memory vector machines. While they all achieved good performance gains through parallelism, the trend in the field quickly shifted away from shared-memory vector machines as technology advanced.

In the mid 1980s, there were many advances in networking and hardware was becoming much cheaper. Due to these advances, the field of hardware-specific parallel FFT implementations quickly became a hot topic of research, with at least 6 significant papers being published over the next decade [32, 8, 34, 66, 5, 29]. The work in [32] was published in 1986 and proposed shared-memory FFT algorithms on a specific SIMD (single instruction, multiple data) system. The authors proposed algorithms for FFTs of both 1- and 2-dimensional data, focusing mainly on memory partitioning methods. Similarly, the work in [8, 34, and 5] all focused on memory partitioning methods on specific hardware platforms. The authors of [34] obtained good results on the Cray-2 environment, while the results in [5] demonstrated “near-linear speedup” with a parallel Cooley-Tukey implementation on a custom-built cluster of PCs.

One common weakness among the papers published during that time is the focus on hardware-specific memory partitioning schemes and optimizations. Despite (mostly) good results for the specific hardware being used, the research had limited impact on future research. This is underscored by the lack of application-focused papers published during that time, implying that the research results were not having a significant impact on other fields. Despite this trend, the introduction of the hypercube network topology led to some of the most significant advances in parallel FFT algorithms.

4.2.1 Leveraging the Hypercube

The Hypercube network topology was pioneered by Danny Hillis with his Connection Machine in the early 1980s and his work became available in 1982 [13]. 5 years later, [56] presented a work that showed that the Hypercube network topology is the ideal topology for the parallel Cooley-Tukey algorithm. Figure [9] demonstrates how the Hypercube is perfectly suited for the parallel radix-2 FFT algorithm.

As illustrated in Figure [9] the Hypercube network topology allows all communication required by the radix-2 parallel FFT algorithm to be sent with just 1 hop each. For distributed memory parallel FFT computation, the Hypercube remains the best and most used network topology, with few attempts to improve on it. Additionally, since the Hypercube is a popular network topology for clusters and supercomputers, it has allowed efficient parallelization of FFT algorithms on existing hardware.
Figure 6: Illustration of how the hypercube perfectly suits parallel FFT algorithms. At each stage, the communication between nodes requires only 1 hop with a hypercube network topology.
In addition to demonstrating that the hypercube was well-suited for parallel FFT algorithms, the authors of [56] introduced 8 novel algorithms for vector multiprocessors that utilize a hypercube topology. This paper was a major advance for the field and laid the groundwork for future distributed-memory FFT research. Immediately following [56] and extending the work, [66] implemented and evaluated a parallel FFT algorithm on an iPSC/2 hypercube. By focusing on reducing communication overhead and efficiently utilizing the hypercube topology, the authors of [66] were able to achieve 80% parallel efficiency. Also working off [56] and leveraging the hypercube, [61] introduced several parallel FFT algorithms designed for massively parallel hypercubes.

4.3 Algorithmic Advances in the ’90s

The algorithms introduced in the 1980s designed for large-scale hypercube network topologies opened the door for a wealth of future parallel FFT work. However, parallel hardware was still somewhat limited and it was unclear how parallel FFT algorithms would scale. Thus, scalability analysis became an important topic and was addressed in the early 1990s.

4.3.1 Scalability Analysis

In 1993, two papers were published that both analyzed the scalability of parallel FFT algorithms [31, 44]. While both papers used the same measure, isoefficiency and used similar methodologies, [31] focused on network traffic and had a more significant impact on the field. The isoefficiency metric is defined as the function of efficiency with respect to the number of parallel processors used. Up to this point, the use of the Hypercube network topology had enabled near-linear speedup, even with a simple parallel Cooley-Tukey-based algorithm. In [31], the authors analytically compute how this efficiency will scale (and compare scalability with a Mesh network topology). While the Mesh is much cheaper to implement than a full Hypercube, the Mesh does not scale nearly as well, even for small problem sizes. They find that, even using a Hypercube, communication bandwidth is the biggest limiting factor to parallel scalability. If the data size (per processor) is relatively small, efficiency more quickly drops off. Figure 7, taken from [31], illustrates how scalability is dependent on the per-processor problem size. However, with network improvements since publication and the drastic increase we have seen in problem sizes, typical FFT algorithms scale quite well with large numbers of processors.

Overall, the work of [31] and [44] further confirmed that parallelization using a Hypercube network topology was a good solution to improve FFT performance. Their work helped researchers find ways of improving parallel FFT performance by focusing on communication bottleneck issues. Furthermore, these works confirmed, analytically, that the FFT algorithm is quite efficiently scalable, as long as hardware and software are adequately configured (problem size is large enough to limit the communications bottleneck).
4.3.2 General Accessibility of the FFT

While research done through the mid 1990s was significant for the supercomputing world, large-scale FFT computation remained inaccessible to many researchers. A major weakness of almost every paper published in the 1980s and early 1990s was that general usability was not a focus. The primary goal was to achieve high performance, and as a result most work was not accessible to people wishing to utilize these advances. This issue was remedied in the late 1990s when cheap, commercially available computers started becoming powerful enough to solve large-scale problems. Sequential FFT algorithms were getting more heavily used and in 1997 Frigo [27] presented the Fastest Fourier Transform in the West (FFTW), a high-performance open-source sequential FFT package. FFTW implemented a Cooley-Tukey sequential FFT algorithm with many machine-independent optimizations. Although FFTW did not support any form of parallelism, it demonstrated that machine-specific FFT algorithms were not required to perform large-scale FFT calculations. We discuss the FFTW package in more detail in Section 5.1.1.

Around this time, cheap, fast clusters were also more accessible, so parallel research started to focus more on machine-independent FFT algorithms as well. In 1998, [16] published an in-depth analysis of many existing (as well as 3 new) parallel FFT algorithms. [16] enabled researchers to quickly determine the best algorithm to use for their purposes, regardless of the underlying hardware. This increased accessibility helped contribute to the various application-focused
papers that began appearing in the early 2000s. While [16] provided a clear analysis of existing methods, the 3 novel algorithms presented did not provide lasting significance.

While the works of both [27] and [16] had greatly improved the accessibility of high-performance FFT algorithms for researchers, their algorithmic contributions were insignificant. Likewise, throughout the 1990s, few algorithmic advances were made. It was established that, despite communication bottleneck issues, Cooley-Tukey-based parallel FFT algorithms were the best-performing options, with few feasible alternatives. In 1999, however, a new algorithm was introduced that, unlike all previous algorithms, used approximation to drastically improve FFT performance.

4.3.3 Integrating FFT with the Fast Multipole Method

In 1992 the Fast Multipole Method (FMM) was introduced as a method of greatly reducing the computational complexity of the N-body problem using some clever approximations that minimally affects overall accuracy [49]. The central ideal behind the FMM is related to rank-deficient matrices. The rank of a matrix is defined as the number of linearly-independent columns (or rows) that it has, and if a matrices rank is lower than its total number of columns, it is considered rank-deficient. More efficient rank-deficient operators exist that can be used in such cases. Furthermore, if rows are considered to be “nearly” rank-deficient (i.e., some columns are nearly linearly-dependent), they can be approximated and rank-deficient operators can be used. The less accuracy required, the more columns (and rows) can be approximated as linearly-dependent.

In 1999 the work of [24] applied the ideas of the FMM to further improve the performance of parallel FFTs. In their work, [24] proves that FFT matrices are nearly rank-deficient and compressing the matrices can greatly reduce the communication required with minimal loss of accuracy. Furthermore, they show that the accuracy/communication trade-off can be easily controlled by a single variable (\(\epsilon\)).

This was one of the first attempts to improve FFT performance using approximation, rather than computing the exact DFT values. Since this work, the FMM has remained a useful way to drastically improve performance, though there are some drawbacks. The major drawbacks are the reduced accuracy and the increased computation required. The details of the FMM is beyond the scope of this paper, though it has become a useful tool to improve FFT performance when absolute precision is not necessary. We direct interested readers to a series of works regarding the FMM and its integration with FFT algorithms [49, 24, 58, 57].

4.4 Large-scale FFTs of the 21st Century

Work in the early 2000s began by building off the contributions of the late 1990s [27][16][24]. Extending [27], the authors of [41] introduce several optimization techniques to try and further improve FFTW. Several of the optimizations
become incorporated into FFTW and, although [41] claimed a 90% performance gain, the performance of FFTW improved only incrementally. During this time, several less successful papers were published that attempted to introduce novel parallel FFT algorithms. In 2000 and 2003, Takahashi et al., published two such papers. In [60], the authors attempted to introduce a complex multi-radius parallel FFT algorithm that utilizes all-to-all communication patterns to achieve good parallel performance. While tested on a large 1024-node cluster, lots of raw results are presented and contrasting results using other algorithms are not provided. In [59], on the other hand, the authors present an 3-D FFT algorithm designed to reduce cache misses on individual nodes. Like [60], the complexity of the algorithm and lack of comparative results are major weaknesses of [59].

Although these algorithm-based papers were unsuccessful at changing the state of parallel FFT computation, the contributions of the late 1990s had a significant impact. This impact can be seen in the number of application-focused papers published around this time [21, 36, 25, 65]. With general FFT algorithms more widely available, researchers began focusing on applications and application-specific optimizations.

The 2 most significant of these papers that utilized the algorithmic contributions of the late 1990s were [25] and [65]. The work in [25] extended previous parallel 3D volumetric FFT work by implementing and testing it on specific hardware (BlueGene/L) and obtained impressive performance gains, though many of the results presented were preliminary and optimizations were hardware-specific. The 2nd application-oriented paper of 2005, [65], utilized existing parallel FFT algorithms to improve performance of an electromagnetic field simulator. Though the paper had no direct contribution to the field of parallel FFT computation, it illustrates that the algorithmic improvements made in the late 1990s and early 2000s had an important impact on a wide range of other fields.

Along with the popularity of parallel FFT algorithms came a series of papers that provided little lasting contribution. The work of [20] introduced an unclear method of “deducing” algorithms, which they used to create a parallel FFT algorithm. The authors go on to make several unsubstantiated claims about the performance of their method and provide no empirical results. Similarly, [37] presented a novel parallel FFT algorithm that was obscure and hardware-specific, providing little significant contribution to the field. By the late 2000s, the popularity of parallel FFT algorithms began to wane and focus moved to more hardware oriented topics.

4.4.1 Vector Processors and GPUs

As explained in Section 4.2, early parallelization techniques were mostly related to shared-memory "Vector Machines." However, once datasets became large enough, distributed memory parallelization became necessary and most work of the '80s and '90s focused on that area. The advances in networking and distributed memory, such as the use of Hypercube network topologies, addressed many of the difficulties with distributed memory FFT calculation. Combined
with the introduction of high-performance Graphics Processing Units (GPUs) and embedded vector processors, this led to work in the mid-2000s focusing on using GPUs to improve FFT performance. The work of [7] was one of the earlier works where they attempted to leverage Graphics Processors (GPUs) for FFT computation. In [7], the authors utilize the IBM Cell processor, similar to the processing unit used by the Sony Play station 3 gaming console. Using this GPU, they were able to out-perform top commodity systems available at the time.

GPUs are well-suited for FFT evaluation because they are primarily designed to perform vector and matrix calculations in parallel using shared-memory. Many graphics applications require real-time matrix manipulation, for which many-core GPUs are designed. The Cooley-Tukey algorithm can be reduced to a Matrix operation and thus performed quickly by vector processors. Additionally, GPU designer nVidia has developed an FFT library with optimizations specifically designed to improve FFT performance on GPUs. However, leveraging GPUs for computation acceleration again puts the burden on memory requirements. Thus, some more recent works [42] focus on using both distributed memory and GPU acceleration to further increase overall FFT performance. We can expect that the combination of GPU acceleration and distributed-memory parallelization to be an on-going area of research.

In addition to being well-suited for FFT computation, this type of vector-based hardware can be applied to small embedded systems. [40] focused on reducing power consumption of parallel FFT execution on GPUs and achieved high efficiency. The novelty of this work is that it is one of the first works to focus on power consumption when evaluating FFT algorithms. [6] also aimed at lowering power consumption, though they proposed a more efficient use of under-utilized hardware components on existing hardware. Finally, [35] proposed several hardware optimizations to improve parallel FFT performance on embedded systems. They show that these optimizations increased parallel FFT performance by 15%, while decreasing the hardware footprint size by 30%.

Despite these papers that demonstrate the significance of new hardware to the field, many of the other hardware-focused parallel FFT papers published in the past several years have been less successful, both in presentation and content. [54], for example, while a novel approach to parallel memory management, did not present any empirical results. [63] also had a poor presentation and limited significant contribution. This trend toward less significant papers may be an indication that the field is becoming stagnant.

4.4.2 FFT in Current-day Supercomputing

While results with new GPU hardware are promising, top-performing FFT platforms remain distributed-memory supercomputers. The work in [50] focused on optimizations specifically tailored to the BlueGene/L hardware. The results presented in [50] are quite impressive, showing parallel FFT performance better than any other system available at the time. Supercomputers continue increasing in scale, with peta- and exa-scale becoming buzzwords in recent years.
As of November 2013, the top 26 supercomputers were peta-scale (over $10^{15}$ Flop/s) and exa-scale ($10^{18}$ Flop/s) is projected to be reached in 2018. Harnessing this new power for the purpose of very large-scale FFT computation is difficult, as memory and communication bottlenecks become ever more pronounced. We can expect that algorithmic advances will be required to leverage the power of next-generation high-performance computing systems.

The biggest practical parallel FFT-related contribution over the past several years is that of [46]. The widely-known FFTW [27] released a massively-parallel open-source version. This has made the use of high-performance parallel FFT much more viable for researchers in other fields. In the late 1990s, we saw how several contributions (including the release of FFTW) spurred research in the FFT applications area by making FFT more accessible. We may see the contributions by [46] have a similar effect on the field over the next several years.

5 Discussion

In the previous section, we presented an overview of the practical advances made in the field of parallel FFT computation (and related fields). The current state of the field is quite mature, with many studies and advances in the half-century since the FFT was introduced [18]. Despite the maturity of the field, there remain open questions and unsolved difficulties. In this section, we recap the common practices and high-points of the field, review the open issues, and propose some potential future areas of study.

5.1 State of the Field

Although the field of parallel (both distributed and shared memory) FFT computation has been heavily studied and is the primary means of achieving increased performance, many researchers opt to use sequential algorithms for convenience. As such, there are a series of highly efficient sequential FFT algorithms that are widely used. The most commonly-used sequential FFT algorithm used to achieve good performance is Split-radix, detailed in Section 4.1.1.

For many applications, sequential FFT algorithms are simply not powerful enough. Modern applications are using increasingly large datasets, causing a continually growing demand for fast FFT computation. This ever-increasing data deluge causes two difficulties that must be overcome: computational requirements and memory requirements. The first difficulty, the increasing need for computational power, can be overcome through parallelization. However, the second difficulty dictates that distributed memory systems are necessary. Therefore, common practice in high-performance FFT evaluation is to leverage large, distributed memory clusters or supercomputers. Since the late 1980s, this has been the focus of much of the high-performance FFT research [56, 66, 16, 50, 16].

Various optimizations have been introduced through the decades, but overall the Cooley-Tukey-based parallel FFT algorithm has changed little. For most parallel systems, network bandwidth tends to be the bottleneck, though the
use of a Hypercube network topology helps alleviate this, as explained in Section 4.2.1. It has been shown that parallel FFT algorithms scale quite well, as long as the system has adequate network bandwidth and the problem size is large [31] (relative to the number of processors). With many of these issues addressed years ago, the area of high-performance distributed-memory FFT algorithms has stagnated. Most researchers with access to large-scale distributed-memory systems are able to achieve adequate FFT performance using simple parallel variants of the Cooley-Tukey algorithm. Therefore, in recent years the field has split into three directions: FFT-FMM integration, GPU Acceleration, and Embedded systems.

FFT-FMM integration, discussed briefly in Section 4.3.3, looks at utilizing the Fast Multipole Method of approximation to greatly reduce the communication requirement of parallel FFT with minimal accuracy loss (and vise-versa). For researchers without access to powerful clusters or supercomputers, FFT-FMM integration is the best way to achieve the performance that some modern applications require. One major advantage of this method is that the FMM algorithm approximates the interactions between elements, greatly reducing the communication requirement of the FFT algorithm on distributed-memory systems.

GPU acceleration, discussed in Section 4.4.1 has been a hotbed of research in a range of fields for the past decade and is well-suited for FFT algorithms. The vector processing nature of GPUs enable them to accelerate FFT computation, and when combined with distributed-memory clusters, can be an economical way of achieving great performance. Despite several GPU-oriented FFT libraries [1], GPU hardware varies so greatly that hardware-specific code is often necessary.

With the boom of cell phones and other portable devices, embedded systems have become an increasingly important area of research. Unlike high-performance computing, the metrics of concern with embedded systems is commonly power consumption and size. This is a budding field and currently consists of hardware-specific research projects. In the years to come this could become a hotbed of research, as the need for smaller and more efficient FFT computation may increase.

5.1.1 Fastest Fourier Transform in the West

Ultimately, the most common use of high-performance FFT algorithms comes from users who are concerned with quickly and easily obtaining results. Therefore, the availability of open-source high-performance FFT libraries have a major impact on other fields. Currently, FFTW, detailed hereafter, is the most widely-used library of high-performance FFT algorithms.

In Section 4.3.2, we discussed how the FFTW open-source library, introduced in 1997 [27], provided the research community with access to high-performance FFT algorithms. FFTW is an open-source library of high-performance FFT algorithms that is continually being improved with the latest optimization techniques. The drawback of FFTW was that, for a long time, it focused only on sequential FFT execution, though it has consistently been the top-performing
A major advance with FFTW came in 2012 when a massively-parallel version of the library became available [46]. Using a high-performance MPI interface for distributed-memory systems, this extended the capabilities of the library and gave researchers in all fields access to high-performance parallel FFT algorithms. To date, the FFTW library contains many algorithms, options, and optimizations including:

- One-dimension and multi-dimensional transforms,
- FFTs of arbitrary input size,
- Efficient use of shared-memory systems using multi-threading,
- Ability to leverage distributed-memory systems using MPI communication,
- Portable across platforms through the use of a C compiler,
- Interface for Fortran or C, and
- Extensive documentation

With this recent introduction of a distributed-memory option, along with the wide range of optimizations available, FFTW will likely remain the top-performing open-source FFT option [46]. FFTW serves as a venue for advances in the high-performance FFT field to reach users, making the research done even more valuable. The major component missing from FFTW is any kind of FMM integration. With the large performance gains obtainable with FMM (discussed in Section 4.3.3), including it in FFTW would provide users with limited hardware resources the ability to tackle much larger problems, especially those using lower-end distributed-memory systems.

5.2 Open Issues

In terms of practical FFT performance, there is still no hard solution for the communication bottleneck frequently encountered on distributed-memory systems that allows. GPU acceleration, larger-scale distributed-memory systems, and ever-increasing dataset sizes all contribute to this difficulty. The use of FMM approximation can dramatically decrease communication requirements, but for some applications the loss of accuracy is unacceptable (and FMM approximation is not easily implemented). Further adding to the difficulties of having parallel FFT algorithms be communication-bound, Hypercubes are becoming too expensive to implement on very large-scale systems. As hardware systems reach "peta-" and "exa-scale", Hypercube networks will simply not be feasible and currently-used FFT algorithms will perform dramatically worse. Network topologies is a rich research field of its own and may help solve these issues, but currently this remains one of the primary practical difficulties with high-performance FFT computation.
5.3 Future Areas of FFT Study

As systems become too large to economically utilize hypercube networks, other network topologies may begin to gain popularity. Therefore, an important area of research may be implementing FFT-like algorithms that are designed with less expensive network topologies in mind. Many new, cheaper network topologies have been introduced in recent years, and creating FFT algorithms that can take advantage may prove useful. For example, recent studies have shown that random network topologies exhibit properties better than existing designs [38]. Dynamic algorithms that take advantage of these properties may out-perform existing algorithms on systems unable to utilize Hypercube topologies.

The area of study that seems to have the most potential is further refining the use of FMM approximation to improve FFT performance, as discussed in Section 4.3.3. Currently, it seems like FMM is the best way to overcome the communication bottleneck that is so common in parallel FFT algorithms. Additionally, the unique ability of FMM approximation to allow a trade-off between accuracy and performance enables users to tune the algorithm to fit their needs, based on available hardware resources. Further refining existing FMM-FFT algorithms may lead to faster, more accurate algorithms. This area has the added benefit of potentially improving FMM algorithms, further improving performance for a range of applications, such as the N-body problem [49].

Finally, the importance of FFT performance on embedded systems must be underscored. As discussed in Section 4.4.1, mobile phones and handheld devices are constantly using FFT algorithms and improving the speed (or reducing the power or size requirements) will have a significant impact on their capabilities. Designing small, efficient hardware dedicated to FFT algorithms is a field that will continue to grow. As this research reaches hardware manufacturer and becoming applied to commercial products, we will continue to see faster, smaller, more efficient devices.

6 Conclusion

The importance of the FFT algorithm in modern technology cannot be understated. From mobile phones to biological data analysis, the FFT has become pervasive across countless domains and is a necessary component of many tech-
nologies. In this paper, we reviewed the major works that led to, improved upon, and contributed to the current state of the field. One major thrust of research has been parallelization, which we focused on with this review.

The FFT algorithm was presented in 1965 [18] and had an immediate impact. By the end of the decade it was wide-spread and a body of research had began. There were many early improvements, but parallel implementations did not appear until 1979 [39]. The introduction of the Hypercube led to the capstone paper of parallel FFT algorithms in 1987 [56]. The next major advances to the field came in the 1990s when papers like [27] and [16] provided researches in other fields accessibility to the algorithm. This led to a wealth of application-oriented papers in the early 2000s, a trend that has continued since. Finally, the most recent trend has been in leveraging GPU-type hardware to further improve the performance of the FFT.

In the half-century since the FFT algorithm was developed, many of the most pressing problems have been solved. Thus, the common practices in use to day are well established and lead to good results for most applications. The current state of the field involves 4 general methods: 1) efficient sequential implementations using the Split-radix (or similar) algorithm (see Section 4.1.1), 2) parallel implementations based on the work in [56] and others, 3) GPU-accelerated implementations Section 4.4.1, and 4) Fast Multipole Method (FMM) approximation (discussed in Section 4.3.3). For the vast majority of applications, these methods provide adequate performance and accomplish the needed task. However, datasets are continuing to grow and new applications are continually being found. In Section 5.3, we discuss some future areas of research, such as further leveraging FMM and optimizing the parallel FFT for varying network topologies. The FFT algorithm will remain a keystone algorithm for a diverse set of applications, so advances in this field will inevitably have a wide range of implications.
<table>
<thead>
<tr>
<th>Citation</th>
<th>Year</th>
<th>Contribution</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Good-58 [30]</td>
<td>1958</td>
<td>Earliest paper to propose a type of FFT algorithm.</td>
<td>Limited to prime-factor length FFTs, implications not recognized until [15].</td>
</tr>
<tr>
<td>Cooley-Tukey-65</td>
<td>1965</td>
<td>Capstone paper credited for introducing FFT. Huge impact on many fields.</td>
<td>Very short, theoretical paper meant algorithm was not very accessible.</td>
</tr>
<tr>
<td>Pease-68 [15]</td>
<td>1968</td>
<td>Earliest parallel FFT algorithm proposed, along with a description of hardware to execute it.</td>
<td>No empirical results since parallelism was so new.</td>
</tr>
<tr>
<td>Yavne-68 [64]</td>
<td>1968</td>
<td>Initially introduced the &quot;Split-Radix&quot; algorithm: a faster, more versatile FFT variation.</td>
<td>Contribution was not recognized and Split-Radix was &quot;reintroduced&quot; later.</td>
</tr>
<tr>
<td>Singleton-69 [52]</td>
<td>1969</td>
<td>Extended the FFT by analyzing non-radix-2 FFTs and determined that they are viable (only powers of 2).</td>
<td>No novel algorithms presented, just further extend/analyze Cooley-Tukey.</td>
</tr>
<tr>
<td>Bergland-72 [10]</td>
<td>1972</td>
<td>First to propose a &quot;parallel&quot; FFT algorithm. Provided detailed algorithm and justification.</td>
<td>No empirical results (hardware was not available yet).</td>
</tr>
<tr>
<td>Korn-79 [39]</td>
<td>1979</td>
<td>Early parallel implementation of FFT on vector computer with empirical results.</td>
<td>Algorithm is circumstantial and somewhat hardware-specific.</td>
</tr>
<tr>
<td>Pan-83 [43]</td>
<td>1983</td>
<td>Proved that the FFT class of algorithms are, at least, O(n lg n).</td>
<td>No algorithms or results shown, no parallelism.</td>
</tr>
<tr>
<td>Duhamel-84 [23]</td>
<td>1984</td>
<td>Popularized the &quot;Split-Radix&quot; algorithm, a significant improvement over Cooley-Tukey for many applications.</td>
<td>Most of the material taken from [61].</td>
</tr>
<tr>
<td>Agarwal-86 [17]</td>
<td>1986</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jamieson-86 [32]</td>
<td>1986</td>
<td>Early paper to focus on memory requirements of parallel FFT algorithms (both 1- and 2-D).</td>
<td>No empirical results, sole focus on memory.</td>
</tr>
<tr>
<td>Swarztrauber-87 [56]</td>
<td>1987</td>
<td>First paper to utilize the Hypercube network for parallel FFT.</td>
<td>The 8 proposed parallel FFT algorithms had limited significance.</td>
</tr>
<tr>
<td>Citation</td>
<td>Year</td>
<td>Contribution</td>
<td>Weaknesses</td>
</tr>
<tr>
<td>---------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Zhu-90 [66]</td>
<td>1990</td>
<td>Extended the Hypercube work of [56] in a more thorough paper with more advanced hardware.</td>
<td>Proposed algorithmic improvements had limited significance.</td>
</tr>
<tr>
<td>Tong-91 [61]</td>
<td>1991</td>
<td>Introduced algorithms to reduce cost to re-order results. Results presented on very large connection machine.</td>
<td>Complex algorithm with incremental improvement.</td>
</tr>
<tr>
<td>Getov-92 [29]</td>
<td>1992</td>
<td>Thorough benchmarking of parallel FFT algs. on flexible platform and compare to theoretical estimate.</td>
<td>Performance only measured/estimated up to 16 nodes.</td>
</tr>
<tr>
<td>Foster-92 [26]</td>
<td>1992</td>
<td>Apply a parallel spectral transform to weather prediction to determine scalability.</td>
<td>Only theoretical estimates, poor results for most cases.</td>
</tr>
<tr>
<td>Patel-93 [44]</td>
<td>1993</td>
<td>Apply scalability measures to several 2D parallel FFT algs.</td>
<td>No empirical results or clear conclusions.</td>
</tr>
<tr>
<td>Sahay-93 [51]</td>
<td>1993</td>
<td>Present a &quot;computation schedule&quot; that interleaves computation and communication.</td>
<td>Questionable claims of &quot;hiding all communication cost&quot;.</td>
</tr>
<tr>
<td>Takahashi-00 [60]</td>
<td>2000</td>
<td>Introduces parallel FFT algorithm using all-to-all communication that keeps results in-order.</td>
<td>Poorly written with huge tables of data without explanation.</td>
</tr>
<tr>
<td>Mullin-02 [41]</td>
<td>2002</td>
<td>Present 4 FFT algorithm optimizations that can improve performance up to 90%.</td>
<td>No discussion of parallelization.</td>
</tr>
<tr>
<td>Katoh-02 [16]</td>
<td>2002</td>
<td>Introduce a method for multiple DNA sequence alignment based on FFT algorithm to reduce computation.</td>
<td>No use of parallelization or improvements to FFT.</td>
</tr>
<tr>
<td>Citation</td>
<td>Year</td>
<td>Contribution</td>
<td>Weaknesses</td>
</tr>
<tr>
<td>-----------------</td>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Takahashi-03</td>
<td>2003</td>
<td>Present novel parallel alg. for 3D FFT aimed at reducing cache misses on indi-</td>
<td>Evaluated on a cluster of only 8 nodes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vidual nodes</td>
<td></td>
</tr>
<tr>
<td>Cui-xiang-05</td>
<td>2005</td>
<td>Introduce general method of &quot;deducing&quot; sequential FFT algorithms and convert to parallel implementation.</td>
<td>Many unsubstantiated claims of performance and scalability.</td>
</tr>
<tr>
<td>Kim-05</td>
<td>2005</td>
<td>Apply known parallel FFT alg. to a network-on-chip architecture to reduce communication.</td>
<td>Algorithm is completely hardware-specific and outdated.</td>
</tr>
<tr>
<td>Eleftherion-05</td>
<td>2005</td>
<td>Implement and test volumetric 3D FFT alg. and evaluated on BlueGene/L with 2048 nodes and compare with FFTW.</td>
<td>Many results only preliminary and specific to hardware.</td>
</tr>
<tr>
<td>Bader-07</td>
<td>2007</td>
<td>Achieve very good performance using a parallel FFT alg. on the IBM Cell BE architecture (used in Playstation 3).</td>
<td>Results only presented for restricted dataset where it performs well.</td>
</tr>
<tr>
<td>Sabharwal-08</td>
<td>2008</td>
<td>Present 3 optimizations to improve parallel FFT performance on BlueGene/L. Results out-perform any system of the time.</td>
<td>Optimizations are specific to the BG/L hardware.</td>
</tr>
<tr>
<td>Berthold-09</td>
<td>2009</td>
<td>Define and evaluate skeletons for 6 different FFT alg.s.</td>
<td>Only preliminary results and no comparison with existing work.</td>
</tr>
<tr>
<td>Taboada-09</td>
<td>2009</td>
<td>Create FMM-FFT alg. and parallelization strategy to improve FMM performance. Results obtained for 0.5B unknowns.</td>
<td>No improvements to FFT, only FMM.</td>
</tr>
<tr>
<td>Taboada-10</td>
<td>2010</td>
<td>Utilize parallel FFT alg. to improve FMM performance on shared-memory system to solve 620M unknowns.</td>
<td>Incremental improvement over previous work.</td>
</tr>
<tr>
<td>Pachnicke-10</td>
<td>2010</td>
<td>Novel implementation of Split-step Fourier alg. on GPU to simulate fibre-optic communication.</td>
<td>Not FFT, but results compared to some FFT alg.s.</td>
</tr>
<tr>
<td>Lorentz-10</td>
<td>2010</td>
<td>Implement and evaluate parallel FFT alg. on a low-power GPU. Show high efficiency for low power.</td>
<td>Lower performance than existing studies, but much lower power consumption.</td>
</tr>
<tr>
<td>Sorokin-11</td>
<td>2011</td>
<td>Propose a parallel memory access scheme for power-of-2 mixed-radix FFT alg.s.</td>
<td>Short paper, hardware-specific implementation, and no results.</td>
</tr>
<tr>
<td>Wu-11</td>
<td>2011</td>
<td>Implement FFT on a heterogeneous multi-core system with a hierarchical network. Empirical results show better performance than other hardware systems.</td>
<td>Poorly written and hardware-specific.</td>
</tr>
<tr>
<td>Pippig-12</td>
<td>2012</td>
<td>Extend FFTW for massively-parallel cases. Extremely accessible library with great results on BlueGene/P.</td>
<td>No novel theoretical work.</td>
</tr>
<tr>
<td>Citation</td>
<td>Year</td>
<td>Contribution</td>
<td>Weaknesses</td>
</tr>
<tr>
<td>---------------------</td>
<td>------</td>
<td>------------------------------------------------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>Abdellah-12 [3]</td>
<td>2012</td>
<td>Implement and test the FFT-shift operation on a GPU. Good results showing GPU are well-suited.</td>
<td>Not a parallel FFT paper, similar performance implications.</td>
</tr>
<tr>
<td>Swartzlander-12 [35]</td>
<td>2012</td>
<td>Propose new hardware operation suited to FFT. Results in 30% smaller and 15% faster FFT operator.</td>
<td>Hardware-specific, not easily applicable except in rare cases.</td>
</tr>
</tbody>
</table>

Table 1

References


