Acta Scientific COMPUTER SCIENCES

Volume 3 Issue 2 February 2021

Review Article

Parallel Fast-fourier Transform

Ola Nadim Halawi*

Department of Computer Science, Lebanon

***Corresponding Author:** Ola Nadim Halawi, Department of Computer Science, Lebanon.

Received: November 30, 2020

Published: January 22, 2021

Abstract

The Fast Fourier Transform (FFT) is an algorithm that is used to compute the Discrete Fourier Transform (DFT) of N points. It is extensively applied in many domains that make use of sinusoidal signals. Some of the well-known FFT algorithms include Radix-2, Radix-4, and Split Radix algorithms, their time complexity is O (n log n). The Fourier Transform algorithm is computationally intensive since it includes many additions, multiplications, and trigonometric functions. Thus, parallelizing the serial FFT algorithms would be very advantageous in terms of code complexity and execution time. This paper aims to provide a means for users to efficiently perform FFT of a polynomial over a finite field, and discusses ways of parallelizing FFT to reduce the communication overhead. Cache-oblivious sub-routines will be used to minimize page faults incurred regardless of the underlying cache-hierarchy, and parallelization will serve in reducing the real time needed by the algorithm. Also, the user will be able to compute the discrete Taylor expansion of a polynomial in a finite field (this functionality is a prerequisite to being able to compute the FFT). To achieve this, a data type will be provided for the user so that he/she can easily input the required polynomial. The project will use Open MPI directives to allow some of the program's segments to run in parallel in order to speed up the computation process. While this provides an efficient parallelized FFT, there is a significant communication overhead that should be considered.

Keywords: Fast Fourier Transform (FFT); Discrete Fourier Transform (DFT); Algorithm

Introduction

Fourier Transform plays a key role in computational science; it has many real life applications in science and engineering. It is often used in digital signal processing applications such as voice recognition and image processing. Also, it is used for solving partial differential equations and quick multiplication of large integers.

The Discrete Fourier Transform is a specific kind of Fourier Transform; it maps a sequence over time to another sequence over frequency and vice-versa. DFT is very computationally exhaustive since it is based on summing a finite series of products of input signal values and trigonometric functions. The widespread use of DFTs is mainly due to the existence of fast algorithms, known as Fast Fourier transform (FFT), which compute the DFT of an input of size N in O (N log N) time instead of the O(N^2) time needed by DFT.

In 1965, Cooley and Tukey published their famous FFT algorithm in a paper. Their algorithm uses a recursive approach of solving DFT of any size N by dividing the DFT into smaller sub- problems which subsequently reduce the time complexity. Since then, many variations of the algorithm have appeared. Fourier Transform is needed in areas where high speed of computation is very necessary, so it was crucial to come up with faster algorithms. A convenient solution is to modify the sequential FFT algorithms into parallel ones.

At first, it was difficult to develop competent and portable parallel FFTs because of the existence of many different parallel architectures. Recently, parallel programming has become less machine dependent, so parallel FFTs algorithms have appeared. This paper discusses the parallelization of these algorithms in order to produce efficient FFT with the least communication overhead.

Objectives and organization

- Performing an efficient Fast Fourier Transform over a finite field of length n = 2^m where m is a power of 2, and developing a parallel version to speed up the runtime of the algorithm.
- Overcoming the high communications overhead imposed by the distributed-memory parallelism of small problems.

The rest of this work is organized as follows. Chapter 2 gives background information on FFT, Chapter 3 describes the FFT algorithm used in this project, experimental work is discussed in Chapter 4, and Chapter 5 covers the parallelization approach while Chapter 6 displays and discusses all the numerical results. Finally chapter 7 is devoted for concluding points and future orientation.

Background Information

The DFT transforms N discrete samples x_0, x_1... x_{N-1} from a spatial domain to the frequency domain according to the formula:

$$X_r = \sum_{l=0}^{N-1} X_l w_N^{rl}$$

Where r = 0, 1... N-1. Implementing DFT requires matrix multiplication and takes O (N^2) time complexity.

Radix-2 FFT algorithm divides the DFT of N discrete points (where N is a power of 2) into two equal parts, the first part computes the Fourier transform of the even index numbers while the other part computes the Fourier transform of the odd index numbers. Finally, it merges them to obtain the Fourier Transform of the whole input. This algorithm runs in O (N log N) by using the divide and conquer approach.

The computation involving each pair of data is called a butterfly; every butterfly consists of one complex addition, one complex subtraction, and one complex multiplication.

Radix-4 FFT algorithm takes an input of length N where N is a power of 4. It divides the sequence into 4 equal parts, computes the Fourier transform of each part, and then it merges them to get the Fourier Transform of the whole input. This approach reduces the number of complex multiplications and additions to (3N/8) log N which is more computationally efficient than Radix-2 FFT algorithm.

Split Radix FFT algorithm, part of the input is computed using Radix-2 algorithm and other part is computed using Radix-4 algorithm. It uses the property that all the even numbered points of the DFT can be performed independent to the odd numbered points. The Split Radix FFT combines the advantages of the Rdix-2 FFT and the Radix-4 FFT.

In brief, while Radix-4 FFT is more efficient than Radix-2 FFT, it has the drawback that the length of the input should be a power of 4 which is not very convenient. From the three above algorithms, Radix-2 is the slowest, Radix-4 is the fastest, and Split radix performs somewhere in the middle.

Finite field is a collection of elements that are closed under addition and multiplication, and it has the property of commutative and associative arithmetic operations.

Cache oblivious algorithm is an algorithm that is designed to take advantage of a CPU cache without having the size of the cache or the length of the cache lines as an explicit parameter. It is designed to perform on multiple machines with different cache sizes, or for a memory hierarchy with different levels of cache having different sizes. It intends to minimize the number of page faults incurred regardless of the underlying cache-hierarchy. In this project, the cache oblivious algorithm will be used in transposing a matrix.

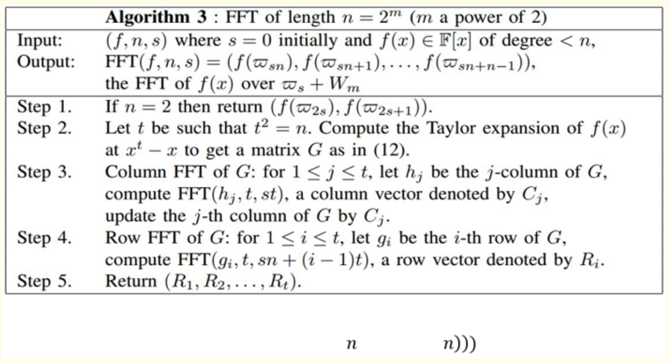
Message Passing Interface (MPI) is a library used by multiple processors to send messages back and forth using send and receive commands. This approach provides a significant increase in performance. The problem is split into parts, each of which is performed by a separate processor in parallel and the final result is gathered at the end. MPI provides an interface for the processors to communicate when they work in parallel.

In the development of parallel FFT algorithms, the two most used ones are the Binary Exchange algorithm and the Parallel Transpose algorithm. The major difference between these two approaches is the way of handling communication between different nodes. In the first, data is distributed equally among p processors and only the first log p stages of the computation require data exchange, with the remaining stages. The second algorithm attempts at solving the internode communication problem. An input of size n is conceptually represented as a √n x √n matrix wherein only one phase of communication is needed in between computations.

The algorithm

In 2010 Shuhong Gao and Todd Mateer published a paper "Additive Fast Fourier Transforms over Finite Fields" that presents an algorithm that computes the FFT of a polynomial over a finite field. Their algorithm makes use of a finite Taylor expansion of the polynomial. This project will be producing a direct implementation of the algorithms presented in their paper. The algorithm works as follows:

- Reduce a problem of size n to 2 n problems of size n
- For an input of degree less than n, obtain T such that $T=\sqrt{n}$.
- Compute the Taylor expansion of the polynomial at x^T-x. This yields T polynomials of degree less than T, which are used to construct a $T\times T$ matrix.
- Use matrix transposition to avoid passing over the matrix column-wise.
- Perform T FFT's of size T to update the columns.
- Another matrix transposition and T FFT's over the rows.
- Final Result is the concatenation of the row FFT's.
- Apply technique recursively till base case, n=2.



```
Algorithm 3 : FFT of length $n = 2^m$ ($m$ a power of 2)
Input:   $(f, n, s)$ where $s = 0$ initially and $f(x) \in \mathbb{F}[x]$ of degree $< n$,
Output:  FFT$(f, n, s) = (f(\varpi_{sn}), f(\varpi_{sn+1}), \ldots, f(\varpi_{sn+n-1}))$,
         the FFT of $f(x)$ over $\varpi_s + W_m$
Step 1.  If $n = 2$ then return $(f(\varpi_{2s}), f(\varpi_{2s+1}))$.
Step 2.  Let $t$ be such that $t^2 = n$. Compute the Taylor expansion of $f(x)$
         at $x^t - x$ to get a matrix $G$ as in (12).
Step 3.  Column FFT of $G$: for $1 \leq j \leq t$, let $h_j$ be the $j$-column of $G$,
         compute FFT$(h_j, t, st)$, a column vector denoted by $C_j$,
         update the $j$-th column of $G$ by $C_j$.
Step 4.  Row FFT of $G$: for $1 \leq i \leq t$, let $g_i$ be the $i$-th row of $G$,
         compute FFT$(g_i, t, sn + (i-1)t)$, a row vector denoted by $R_i$.
Step 5.  Return $(R_1, R_2, \ldots, R_t)$.
```

n $n)))$

Figure a

This algorithm takes as input a polynomial of degree less than n, a number s which is initially zero and a number m which represents the size of the finite field, and it returns the FFT of the polynomial over the finite field. The evaluation is a list of polynomials that have only one term with the exponent being 0.

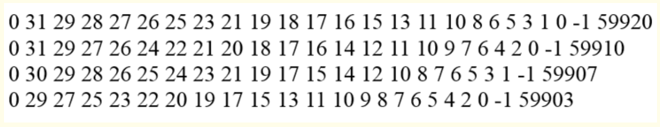
The algorithm makes use of a Taylor expansion of the given polynomial at x x) where n =. It returns a list of t polynomials whose degree is at most t-1. The results are then stored in a matrix, where each term corresponds to a coefficient. An FFT is then performed over each column of the matrix. This would incur many page faults, especially on large matrices. Therefore, the matrix transposition module is used here to transpose the matrix. This makes it possible to traverse the transposed matrix's rows, which is more 'cache-friendly'.

It is a recursive algorithm, with the base case occurring when n = 2. At that point, the algorithm evaluates (using an evaluation subroutine) the polynomial at two points (specified by s), inserts them in a list and returns the list. The final list is a concatenation of all the polynomials returned by the base case.

Experimentation: Preparation and Discussion

Input

Below is an example of the input



```
0 31 29 28 27 26 25 23 21 19 18 17 16 15 13 11 10 8 6 5 3 1 0 -1 59920
0 31 29 27 26 24 22 21 20 18 17 16 14 12 11 10 9 7 6 4 2 0 -1 59910
0 30 29 28 26 25 24 23 21 19 17 15 14 12 10 8 7 6 5 3 1 -1 59907
0 29 27 25 23 22 20 19 17 15 13 11 10 9 8 7 6 5 4 2 0 -1 59903
```

Figure b

Each line of the input file starts with a zero then with a descending order of beta polynomial and ends with the degree in x.

The first line can be read as follows:

$(\beta$^31+ β^29+ β^28+ β^27+ β^26+ β^25+ β^23+ β^21+ β^19+ β^18+ β^17+ β^16+ β^15+

β^13+ β^11+ β^10+ β^8+ β^6+ β^5+ β^3+ β^1+ β^0) X^59920

Tools

- C language

Reason: Still widely used by the scientific community and it supports Open MPI frameworks.

- Computer cluster in order to observe the effects of parallelization on multiple cores.

Methods

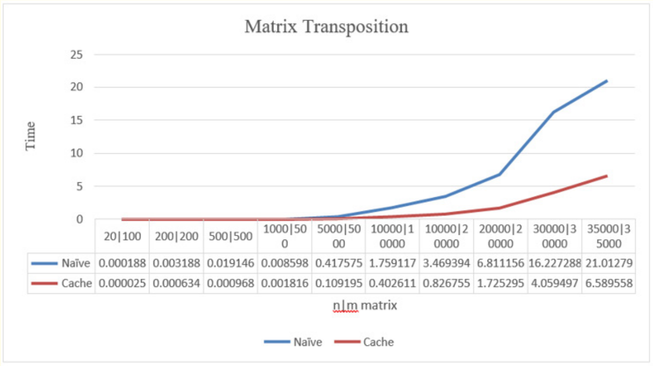
- Cache Oblivious techniques
- Open MPI library to parallelize the program and distribute it over a computer cluster.

Matrix transposition

This is the first module in the project. The algorithm proposed here assumes a one dimensional representation of a two dimensional matrix, meaning that the 2d matrix is implemented as a standard array, with row and column boundaries to be inferred by the programmer. It is a cache- oblivious matrix transposition, so it is asymptotically optimal in performance with respect to the number of page faults incurred. The implementation is a generic one; it can transpose any given matrix of any data type. The performance of this module was extensively tested on several data types.

Below is a graph illustrating the difference between the cache-oblivious implementation and a standard naive matrix transposition algorithm for different matrices sizes. The runtimes plotted are in seconds.

- **Input:** A 1d array representing the generic n x m source matrix to be transposed, a 1d array onto which the source is to be transposed: it represents the m x n destination matrix, number of rows and columns of the source matrix, the indices of the row and columns of the matrix, and the size of the data type of the matrix.
- **Output:** The transpose of the source matrix in the destination matrix.



	20|100	200|200	500|500	1000|500	5000|5000	10000|10000	10000|20000	20000|20000	30000|30000	35000|35000
Naïve	0.000188	0.003188	0.019146	0.008598	0.417575	1.759117	3.469394	6.811156	16.227288	21.01279
Cache	0.000025	0.000634	0.000968	0.001816	0.109195	0.402611	0.826755	1.725295	4.059497	6.589558

Figure c: Comparison between standard and cache-oblivious matrix transpositions.

Polynomial data type

The input will be a polynomial in x whose coefficients are also polynomials in another variable β where the β coefficients are either 0 or 1. Here the polynomial is assumed to have a large degree.

Two choices are available to represent the polynomial data input either by dense representation or by sparse one.

In the dense representation, to represent the β polynomial we will need to represent only their exponents. A dynamic array can be used to represent the polynomial's exponent where each exponent would be stored at an index. However, this would take up 4 bytes of space for every term, so in the worst case a polynomial would need to be represented by (4 * 64=256) bytes, assuming the Beta polynomials are at most of degree 63.

Another approach would be to use the polynomial powers as indices to an array of bits and accordingly represent it as such. So if a polynomial contains the term 1, bit 0 would be set to 1 (Since $\beta^0 = 1$, where 0 is the index). Similarly, the presence of β^2 would be indicated by setting bit (2) to 1. Using this representation would take up 64 bits = 8 bytes for the whole polynomial. However, there are no guarantees about the order in which the bits are stored, which can produce unexpected or unwanted results on different machines.

In a sparse representation, no space is allocated for a term in the β polynomials unless it is present and so, the coefficients are represented using a dynamic array of numbers with the degrees in x also represented as numbers. Since they are indistinguishable from one another, a marker is placed at the beginning of each term to mark the position of the degree in x.

Using the dense representation saves a huge amount of memory because one could represent the polynomials in β using 64 bits of memory storage. The downside to using the dense representation is that it would restrict the degree of the polynomials in β to 63. On the other side, if the polynomial is sparse many flops will be wasted to check empty bits. Also, the sparse representation would take up more space than the dense one, since every power of β would need an integer (32 bits = 4 bytes).

Finally, after comparing the pros and cons of the two approaches, sparse representation was chosen since it would be more efficient for the common applications of the program.

To use matrix transposition, each element of the matrix has to have the same size, which was not possible with our chosen representation. Therefore, a conversion subroutine is developed that converts from a given representation to another.

Beta division

In a given finite field whose size is, the maximum degree a polynomial in the field can achieve is m-1. When multiplying two polynomials in the field, the degree of the obtained polynomials may exceed the set limit. To ensure the multiplication remains within the field, we take the polynomial modulo a primitive polynomial of degree.

This module accepts a primitive polynomial (As a linked list), a power of β, and a Boolean array of size m, where the answer is stored. If the array[k] is set to true, this means that the term β is in the result. To reduce polynomials, one must reduce each term individually and add them to obtain the result.

Data representation

To implement the algorithms, a polynomial representation is needed. A linked list representation would cause the polynomial to be stored in chunks across memory, which would increase the number of page faults incurred, especially when dealing with large polynomials. Therefore, we decided to represent the polynomial using a data structure called 'pol'. This structure was implemented as a dynamic array of integers, each term of the polynomial can be represented in the following manner:

(N,,,…,, E)

N: Number of elements in the term (k+1). This is used to mark the position of the exponent in this term, since they are all integers and are otherwise indistinguishable. The degree of the β variable present in the coefficient of the term. These should be stored in descending order. In a finite field of size=, K can be at most m. E: The exponent of this term.

To add two polynomials, we merge the terms of the two polynomials in descending order of exponents. In the case where two exponents are the same, their coefficients are also merged in descending order as well. In the case where the two coefficients share a term β, this term is not included in their merge. If the merge of the two coefficients yields 0 (the two coefficients are equal), their exponent is not added in the merge of the two polynomials.

In addition to the array of integers, the data structure has two other members, one which represent the size of the array in memory and the other representing the last filled index.

Data type to hold list of polynomials

The algorithms we are discussing return a list of polynomials rather than a single one. Therefore, we need to define a data type that is a list polynomials, along with the ability to concatenate two lists. This method accepts two pointers to polynomial lists and returns a pointer to a new, dynamically allocated list that is the concatenation of the supplied lists. The new list contains the elements of the first list, followed by the elements of the second list.

Taylor series expansion

This method accepts a pointer to a polynomial, an integer n and another integer t, where the polynomial is of degree less than n. It returns a pointer to a list of polynomials that represent the Taylor expansion of the polynomial at x x).

This expansion is a finite expansion, and has exactly L =ceiling (n/t) polynomials. To obtain the original polynomial, the first polynomial is multiplied by x x), the second is multiplied by x x) … the polynomial is multiplied by x x). This algorithm is recursive with the base case occurring when n is less than t. At that point, the algorithm inserts the polynomial in a list and returns the list. The final answer is a concatenation of the lists obtained from the base case.

Evaluation

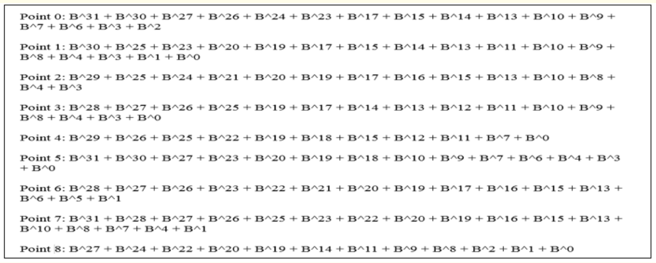
This subroutine is a crucial part of the FFT algorithm. It takes a polynomial, an integer and a field size m and evaluates the polynomial at a certain point specified by s and the primitive polynomial associated with the given field. To evaluate the polynomial, we need to make use of a list of β polynomials which can be generated from a primitive polynomial. These tables, along with the primitive polynomials, are stored in a header file "tables. h". The tables are created on the heap at the beginning of the program and are deallocated at the end.

The evaluation step requires multiplying two polynomials in β. After the multiplication, we use the Beta division module to obtain the appropriate element in the finite field as an array of Booleans. The final step of this subroutine converts the Boolean array representation to our chosen representation.

Structure of the program

The user will create a polynomial using the representation that we have provided and call the method that we have defined to compute the FFT. If the degree of the polynomial is sufficiently small, a list of polynomials will be returned. If not, the algorithm will use the Taylor expansion of the polynomial to create a matrix and then recursively call itself on the columns and the rows of this matrix. The results from each row and each column are concatenated together and returned back to the user as a list of polynomials.

Sample output



```
Point 0: B^31 + B^30 + B^27 + B^26 + B^24 + B^23 + B^17 + B^15 + B^14 + B^13 + B^10 + B^9 +
B^7 + B^6 + B^3 + B^2

Point 1: B^30 + B^25 + B^23 + B^20 + B^19 + B^17 + B^15 + B^14 + B^13 + B^11 + B^10 + B^9 +
B^8 + B^4 + B^3 + B^1 + B^0

Point 2: B^29 + B^25 + B^24 + B^21 + B^20 + B^19 + B^17 + B^16 + B^15 + B^13 + B^10 + B^8 +
B^4 + B^3

Point 3: B^28 + B^27 + B^26 + B^25 + B^19 + B^17 + B^14 + B^13 + B^12 + B^11 + B^10 + B^9 +
B^8 + B^4 + B^3 + B^0

Point 4: B^29 + B^26 + B^25 + B^22 + B^19 + B^18 + B^15 + B^12 + B^11 + B^7 + B^0

Point 5: B^31 + B^30 + B^27 + B^23 + B^20 + B^19 + B^18 + B^10 + B^9 + B^7 + B^6 + B^4 + B^3
+ B^0

Point 6: B^28 + B^27 + B^26 + B^23 + B^22 + B^21 + B^20 + B^19 + B^17 + B^16 + B^15 + B^13 +
B^6 + B^5 + B^1

Point 7: B^31 + B^28 + B^27 + B^26 + B^25 + B^23 + B^22 + B^20 + B^19 + B^16 + B^15 + B^13 +
B^10 + B^8 + B^7 + B^4 + B^1

Point 8: B^27 + B^24 + B^22 + B^20 + B^19 + B^14 + B^11 + B^9 + B^8 + B^2 + B^1 + B^0
```

Figure d

Parallelization

At the first step the data is divided equally among all MPI processes (sent to appropriate indices to eliminate need for transposition at the first step). Data is distributed over all MPI nodes, which then perform the FFT on their data (columns) and update their values. Then they all transpose their own matrices (makes sending easier, chunks instead of individual) before sending data to each other. Finally, they perform the final FFTs over their data; all processors send their lists to processor 0 which then prints out the ending result.

Approaches to parallelization

While computing FFT, data elements are exchanged very frequently in order to compute the butterfly relation. Because of this, parallel computing models (except for shared memory) must take care of handling communication delays across multiple processors.

There are many forms of parallel architectures available. In shared memory multiprocessors all the processes can access all memory while in distributed parallelization different processes have access to different portions of memory. Message based multi-processors is also a form of parallelization, recently the Message Passing Interface (MPI) has become a very popular form of writing parallel programs for massively parallel multi-processors.

One of the approaches to parallelize FFT is to perform all butterfly computations in parallel, and to gather the results at the end of each stage. The issue with this approach is the communication overhead because at the end of each stage all the processors have to communicate with each other, and the system will not proceed to the next stage until all the output from the earlier stage is produced. This is a very good candidate to be implemented in a shared memory system.

Another approach is to increase the scope of parallelization with stages, in the first stage only one processor will be active while all the other available processors are idle. However, the number of processors used will be increased in the following stages.

In order to reduce the communication overhead, the "Communicate twice" algorithm is used. Since the next processor is aware of the data that it is waiting for from the earlier processor, it could perform the same computation that the previous processor is performing and then use the result for its stage computation. Some processors could be performing similar and repeated computations, but this approach is efficient for fast processor nodes.

This can be implemented using any number of processors. When we cannot divide the data elements at any stage, then the processor that was supposed to divide the data set will continue to perform computations on the entire data set from the previous stage. In this approach, we map out the entire communication for each stage for all the processors before beginning to compute the FFT.

Blocking MPI communication was used instead of non-blocking communication as it will not make any difference in performance in this case and just increase the amount of memory required because of the requirements of MPI COMM_WORLD.Isend() method.

Comparison of the Parallelizing Algorithms

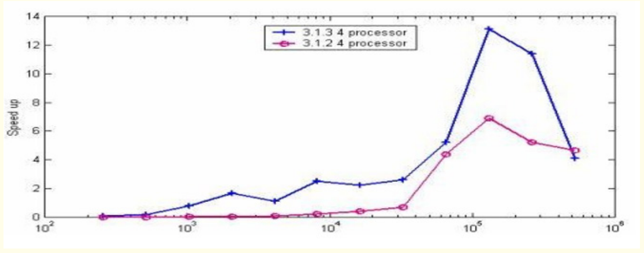


Figure e: Comparison of the algorithm implemented in 4 processor system with the "Communicate twice algorithm" and with "Scope of parallelization increases with stages".

Clearly, the Communicate twice algorithm beats the other one.

The table below describes a static heuristic to determine the number of processors to use given the input size in a MPI based cluster.

Input size range	Number of processors to use
1-256	1
256-16384	2
16384-1.5×10^5	4
1.5×10^5-1×10^6	8

Table 1

Empirical Results

Serial Program Runtime

Degree	FFT Length	Run Time in seconds
6	2^4	0.01
17	2^8	0.18
53	2^8	1.14
110	2^8	2.05
250	2^8	2.13
520	2^{16}	563.01
1200	2^{16}	738.60
2000	2^{16}	881.39
3000	2^{16}	886.68

Table 2

The above able shows the run time of the serial program (without parallelization) given different degrees and FFT lengths. It's evident that the running time increases as the degree increases.

Runtime after segmentation

The above results doubt the code implementation in the first place, so time segmentation was needed.

Evaluation is broken down into preparation time (omitted here) and calculation time. Calculation time is further broken down into addition and multiplication. Only multiplication is shown here as it is the dominant factor. The tested degrees are 1200, 1500, and 2000.

Degree in x: 1200 GF: 2^{32}
Total runtime = 74.3082 seconds

	Total time	Percentages
Evaluation	81.9	93%
Total Calculation Time		79.03
Multiplication	68.66	78.35%
Transposition	0.05	0.06%
Taylor Expansion	0.983	1.12%

Table 3

Degree in x: 1500, GF: 2^{32}
Total runtime = 87.63 seconds

	Total time	Percentages
Evaluation	81.9	93%
Total Calculation Time		79.03
Multiplication	68.66	78.35%
Transposition	0.05	0.06%
Taylor Expansion	0.983	1.12%
Conversion	0.43	0.49%

Table 4

Degree in x: 2000 GF: 2^{32}
Total Runtime = 88.13 seconds

	Total time	Percentages
Evaluation	82.1	93.158%
Total Calculation time		79.4
Multiply	68.66	77.908%
Transposition	0.05	0.057%
Taylor Expansion	0.983	1.115%
Conversion	0.43	0.488%

Table 5

Runtime for Large Degrees after Further Segmentation.

The time was segmented further. The tested degrees are 10,000|20,000|30,000|40,000|50,000 "n" is the number of points the polynomial will be evaluated at, so the degree of the polynomial must be less than n.

Degree of the polynomial ($x^{10,000}$): n= 65,536, GF: 64

Total time: 89.468451 seconds
Evaluation: total time: 79.018496 seconds
Convert to a general polynomial for multiplication: 1.330156 seconds
Arithmetic: 73.741317 seconds

- Getting first term to multiply with: 1.934307 seconds
- Power (degree): time spent: 0.992673
- Multiplication time: 6.943383 seconds
- Division time: 39.841526
- Reduce time: 18.955932 seconds
- Evaluation end time: 1.292979 seconds

Taylor: 2.391822 seconds
Conversion: 0.644655 seconds
Transposition: 0.047351 seconds
FINAL STEP TIME (concatenation): 4.998192 seconds

Degree of the polynomial ($x^{20,000}$): n= 65,536 GF: 64

Total time: 75.681523 seconds
Evaluation: total time: 66.522444 seconds
Convert to a general polynomial for multiplication: 1.216757 seconds
Arithmetic: 61.587403 seconds

- Getting first term to multiply with: 1.426823 seconds
- Power (degree): time spent: 0.869765
- Multiplication time: 5.780981 seconds
- Division time: 33.482148
- Reduce time: 15.814538 seconds

Evaluation end time: 1.125878 seconds
Taylor: 1.678628 seconds
Conversion: 0.551574 seconds
Transposition: 0.045514 seconds
FINAL STEP TIME (concatenation): 4.877057 seconds

Degree of the polynomial ($x^{30,000}$) n= 65,536 GF: 64

Total time: 77.489933 seconds
Evaluation: total time: 68.152884seconds
Convert to a general polynomial for multiplication: 1.223030seconds
Arithmetic: 63.179807 seconds

- Getting first term to multiply with: 1.467617seconds
- Power (degree): time spent 0.893853
- Multiplication time: 5.927357 seconds
- Division time: 34.356698
- Reduce time: 16.216129 seconds

Evaluation end time: 1.146678seconds
Taylor: 1.787188 seconds
Conversion: 0.560490 seconds
Transposition: 0.046106 seconds
FINAL STEP TIME (concatenation):4.893375 seconds

Degree of the polynomial ($x^{40,000}$) n= 65,536 GF: 64

Total time: 76.134272 seconds
Evaluation: total time: 66.935502 seconds
Convert to a general polynomial for multiplication: 1.219473 seconds
Arithmetic: 61.980878 seconds

- Getting first term to multiply with : 1.440934 seconds
- Power (degree) : time spent 0.876509
- Multiplication time: 5.821588 seconds
- Division time:33.688455
- Reduce time: 15.914761seconds

Evaluation end time: 1.130903 seconds
Taylor: 1.787188
Conversion: 0.560490
Transposition: 0.046106
FINAL STEP TIME (concatenation):4.893375

Degree of the polynomial ($x^{50,000}$) n= 65,536 GF: 64

Total time: 80.527825 seconds
Evaluation: total time: 70.878010 seconds
Convert to a general polynomial for multiplication: 1.226665 seconds
Arithmetic: 65.875284 seconds

- Getting first term to multiply with : 1.520289seconds
- Power (degree) :0.927680
- Multiplication time: 6.183681 seconds
- Division: 35.848519
- Reduce time: 16.905305 seconds

Evaluation end time: 1.167658seconds
Taylor: 1.996446
Conversion: 0.586650 seconds
Transposition: 0.045994 seconds
FINAL STEP TIME (concatenation): 4.895934 seconds.

This table shows the percentages of the time consumed by every method.

Method	Percentage
Evaluation	88.59%
Evaluation end time	1.459%
Taylor	2.49 %
Conversion	0.73%
Transposition	0.057%
Concatenation	6.1199%

Table 6

The time segmented over the evaluation time can be summarized by the below table which shows the percentages of the time consumed by every method at the arithmetic phase.

Method	Percentage
First Term	2.3%
Power	1.408%
Multiplication	9.386%
Division	54.4187%
Reduce	25.654%

Table 7

Performance of the program

The below graphs and tables show the FFT serial algorithm performance with and without matrix transposition.

Keywords

- L1 cache miss ratio= L1-dcache-loads / L1-dcache-load-misses
- Branch misprediction ratio= branch-misses/branch-instructions

Explanation

During the execution of certain programs there are places where the program execution flow can continue in several ways; these are called branches, or conditional jumps.

The CPU uses a pipeline which allows several instructions to be processed at the same time. When the code for a conditional jump is read we do not know yet the next instruction to execute and insert into the execution pipeline. This is where branch prediction comes in. Branch misprediction occurs when a CPU mispredicts the next instruction to process in branch prediction.

Case 1

Polynomial of degree 520 n=65536

GF= 64

Event-name	Without matrix transposition	With matrix transposition
CPU-cycles	2,193,468,010,014	2,194,145,260,022
instructions	3,698,446,719,817	3,697,179,220,419
cache-references	366,886,017	343,884,206
cache-misses	15,817,918	15,393,933
branch-instructions	563,043,373,262	562,765,944,034
branch-misses	2,558,673,550	2,589,657,405
bus-cycles	56,322,337,940	56,332,772,079
L1-dcache-loads	2,017,858,910,433	2,017,026,179,171
L1-dcache-load-misses	384,228,167	386,480,719
L1-dcache-stores	323,665,757,681	323,353,365,872
L1-dcache-store-misses	229,960,900	231,344,155

Table 8

Case 2

Polynomial of degree 250 n=256

GF= 64

Case 3

Polynomial of degree 110 n=256

GF= 64

Case 4

Polynomial of degree 53 n=256

GF= 32

Event Name	Without matrix transposition	With matrix transposition
CPU-cycles	8,161,909,071	8,189,436,498
instructions	13,686,272,124	13,735,624,755
cache-references	1,542,568	1,105,916
cache-misses	29,468	17,258
branch-instructions	2,077,395,578	2,079,959,524
branch-misses	8,981,676	9,030,479
bus-cycles	209,993,408	210,745,762
L1-dcache-loads	7,527,538,026	7,560,537,057
L1-dcache-load-misses	1,143,744	1,044,672
L1-dcache-stores	1,170,401,934	1,173,295,772
L1-dcache-store-misses	669,412	651,673

Table 9

Event Name	Without matrix transposition	With matrix transposition
CPU-cycles	826,615,299	837,354,461
instructions	1,464,547,751	1,463,360,275
cache-references	119,248	134,029
cache-misses	26,921	26,059
branch-instructions	219,821,118	219,156,654
branch-misses	2,927,846	3,000,674
bus-cycles	21,462,852	24,158,737
L1-dcache-loads	692,798,833	708,064,919
L1-dcache-load-misses	131,347	180,819
L1-dcache-stores	150,852,880	155,195,655
L1-dcache-store-misses	116,660	139,699

Table 10

Case 5

Polynomial of degree 17 n=256

GF= 64

Case 6

Polynomial of degree 6 n=16

GF= 32

Event Name	Without matrix transposition	With matrix transposition
CPU-cycles	731,899,581	717,187,838
instructions	1,293,806,536	1,266,444,766
cache-references	83,502	86,941
cache-misses	18,060	11,162
branch-instructions	192,885,703	191,895,217
branch-misses	2,606,660	2,642,066
bus-cycles	20,384,651	19,039,016
L1-dcache-loads	617,259,028	633,458,627
L1-dcache-load-misses	106,955	116,621
L1-dcache-stores	130,129,567	131,271,138
L1-dcache-store-misses	62,530	73,300

Table 11

Event Name	Without matrix transposition	With matrix transposition
CPU-cycles	4,421,989,961	4,425,418,090
instructions	7,432,963,707	7,436,855,738
cache-references	1,148,443	877,996
cache-misses	24,471	12,639
branch-instructions	1,122,114,787	1,118,566,337
branch-misses	4,892,571	4,890,994
bus-cycles	113,900,786	114,354,419
L1-dcache-loads	4,059,008,253	4,069,216,932
L1-dcache-load-misses	621,772	655,535
L1-dcache-stores	639,134,629	647,647,211
L1-dcache-store-misses	381,899	416,739

Table 12

The table below compares the run times of FFT with and without matrix transposition. An improvement in the running time can be noticed.

Note GF=64.

Event Name	Without matrix transposition	With matrix transposition
CPU-cycles	10,781,711	20,070,172
instructions	18,251,289	24,653,384
cache-references	39,575	69,115
cache-misses	6,989	7,534
branch-instructions	3,751,765	1,194,815
branch-misses	80,789	36,551
bus-cycles	1,946,153	814,768
L1-dcache-loads	12,552,270	20,827,214
L1-dcache-load-misses	17,995	40,723
L1-dcache-stores	2,635,962	8,029,156
L1-dcache-store-misses	1,135	60,796

Table 13

Degree	Time for Naïve FFT in seconds	Time for Transposed FFT in seconds
10,000	87.076441	86.545085
20,000	74.410854	74.172684
30,000	75.786633	75.315532
40,000	74.287920	74.063398
50,000	78.769958	78.412049

Table 14

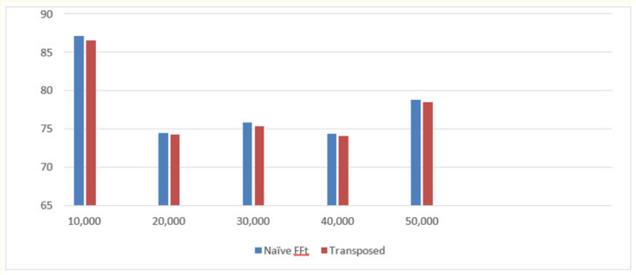


Figure f

Serial Vs Parallel

FFT of length 2^{16}, field of size 2^{64}

Degree	Time of serial program	Time of parallel (4 nodes) program
10,000	86.54	21.45
20,000	74.17	18.61
30,000	75.31	18.81
40,000	74.06	18.78
50,000	78.41	19.41

Table 15

Using the serial program, the maximum time consumed is approximately 90 seconds, so for a bigger matrix of size 65,536 65,536 the time consumed will be approximately 5,898,240 seconds (68.266 days).

Using the parallel program, the maximum time consumed is approximately 20 seconds. An FFT of size 2^{32} which is approximately 4 billion would have to perform 2^{16} FFTs of size 2^{16} on the columns and the same number over the rows. In parallel, the work will take only half an hour (26 times faster than the serial FFT) [1-17].

Thus, there is a clear improvement in the run time when the program is parallelized.

Parallel program runtimes

Processes	Time
2	44.10
4	21.3
8	10.8
16	7.86
32	5.62

Table 16

Run time of the parallelized FFT Program is decreasing as the number of processes increases.

Chapter 7 Conclusion and Future Work

Fast Fourier Transform (FFT) is used widely in many scientific, engineering and mathematical applications. In some cases, it is used to analyze a huge set of input data. Hence, parallel FFT algorithms are desirable.

This paper establishes that FFT algorithms can be parallelized and we can also reduce the execution time. The "communicate twice" algorithm reduces runtime considerably than the other algorithms used in, and it can be implemented using any number of processors. But, the engineering compromise here is that it is not a good cluster citizen because it increases the load of the processor and increases the cumulative CPU cycles spent on the operation.

The project used the FFT algorithm proposed by Shuhong Gao and Todd Mateer which runs in O(log n x log (log n)) time. The implementation was divided into several modules. Matrix transposition was the first module to be implemented; there was a clear improvement in the running time of the cache-oblivious matrix transposition compared to the naïve one. Sparse representation was used to represent the polynomial data type, and a converter to convert between sparse and dense representation was created. Also, a data type to hold list of polynomials was developed.

The paper discussed some approaches to parallelization and chooses the "communicate twice" algorithm as the best one of them. The runtime of the serial FFT program was compared for different degrees and FFTs lengths. Further segmentation of the running time was done in order to run the program on high degrees. Also, the performance was compared with and without matrix transpositions. Indeed, with matrix transposition the performance was better. Finally, the run time of the serial FFT program was compared to that of the parallelized program for high degrees. It's evident that the time taken by the parallel program is much less than that taken by the serial one, and it decreases as the number of processes increases.

Future improvements of division and multiplication algorithms need to be taken into account if we need to intensively test FFTs of much higher degrees. Also, we need to work on compact data representation.

Bibliography

1. A Dubey., *et al*. "A general purpose subroutine for Fast Fourier Transform on a distributed memory parallel machine". *Parallel Computing* 20 (1994): 1697-1710.
2. Shuhong Gao and Todd Mateer. "Additive Fast Fourier Transforms over Finite Fields".
3. Bo Liu. "Parallel Fast Fourier Transform, 159.735 Studies in Parallel and Distributed System"
4. Brigham EO. "The fast Fourier transform and its applications". Englewood Cliffs, N, J.: Prentice Hall (1988).
5. C Van Loan. Computational Frameworks for the Fast Fourier Transform, SIAM, Philadelphia, PA, (1992).
6. Chu E and George A. FFT algorithms and their adaptation to parallel processing, Linear Algebra and its Applications 284 (1998).
7. Chu E and George A. "Inside the FFT black box: serial and parallel fast Fourier transform algorithms". Boca Raton, Fla.: CRC Press (1999).
8. Gray RM and Goodman JW. "Fourier transforms: an introduction for engineers". Boston: Kluwer Academic Publishers (1995).
9. Herbert Karner and Christoph W Ueberhuber. Parallel FFT algorithms with reduced communication overhead, Institute for Applied and Numerical Mathematics, Technical University of Vienna
10. M C Pease. "An adaptation of the fast Fourier transform for parallel processing". *Journal of the ACM* 15.2 (1968): 252-264.
11. Matteo Frigo., *et al*. Cache- Oblivious Algorithms, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139.
12. Michael Balducci., *et al*. "Comparative Analysis of FFT algorithms in sequential and parallel form". Mississippi State University.

13. P N Swarztrauber. "Multiprocessor FFTs". *Parallel Computing* 5 (1987): 197-210.
14. Quinn MJ. "Parallel programming in C with MPI and OpenMP". New York: McGraw-Hill Higher Education (2004).
15. Somasundaram Meiyappan, Implementation and performance evaluation of parallel FFT algorithms, School of Computing National University of Singapore.
16. W L Briggs and V E Henson. The DFT: An Owner's Manual for the Discrete Fourier Transform, SIAM, Philadelphia, PA (1995).
17. W Yang. "Parallel ordered FFT algorithms on distributed-memory multiprocessors". M.Sc. Thesis, Department of Mathematics and Statistics, University of Guelph, Guelph, Ontario (1996).