Parallel Computing in Multicore Architecture

Taslima Yeasmin
Department of Computer Science And Engineering
Department of Computer Science And Engineering, Jahangirnagar University Savar Dhaka, Bangladesh
Email- subah_cse@yahoo.com

Abstract- Many Applications are ranges from signal processing to astronomy includes FFT (Fast Fouries Transform) to increase the speed of computation. The computational demands of software continue to outpace the capabilities of processor and memory technologies especially in scientific and engineering programs. The parallel FFT computation offers a new advancement to increase the speed of computation is the objective of this research. This research deals with various approaches for FFT computation in multicore processor in application of two-dimensional FFT and DCT.

Keywords – DFT, FFT, Multicore Architecture, Pmatlab, FFT Performance

I. INTRODUCTION

The Fast Fourier Transform (FFT) is important to a Variety of applications, from signal processing to spectral methods and for making the solutions of mathematical equations. The computationally challenging nature of the FFT has made it a staple of benchmarks for decades. The FFT benchmark is a simple program that stresses both the local memory bandwidth and the network bandwidth of a parallel computer. It is typical of a class of problems that can perform well on canonical parallel computers, provided there is sufficient network bandwidth. The FFT was used for measuring the bandwidth between processors on a network. FFT is also a good example of a communication-intensive parallel application with complex communication patterns and allows the demonstration of a number of useful parallel coding techniques, in particular, how to write a distributed array program that succinctly and efficiently implements these complicated patterns. A serial FFT algorithm is usually limited by the memory bandwidth. The FFT provides an excellent illustration of the kinds of complex communication patterns that can perform well on canonical parallel computers, provided there is sufficient network bandwidth. This paper illustrates both coding and optimization approaches for efficiently implementing the FFT algorithm in pMatlab.

II. SYSTEM ARCHITECTURE

The Cell Broadband Engine (or the Cell/B.E.) is a novel high-performance architecture designed by Sony, Toshiba, and IBM (STI), primarily targeting multimedia and gaming applications. The Cell/B.E. consists of a traditional microprocessor (called the PPE) that controls eight SIMD co-processing units called synergistic processor elements (SPEs), a high-speed memory controller, and a high-bandwidth bus interface (termed the element interconnect bus, or EIB), all integrated on a single chip. The Cell is used in Sony’s PlayStation 3 gaming console, Mercury Computer System’s dual Cell-based blade servers, IBM’s QS20 Cell Blades, and the Roadrunner supercomputer.

In this paper we present the design of an efficient parallel implementation of Fast Fourier Transform (FFT) on the Cell/B.E. FFT is of primary importance and a fundamental kernel in many computationally intensive scientific applications such as computer tomography, data filtering, and fluid dynamics. Another important application area of FFTs is in spectral analysis of speech, sonar, radar, seismic, and vibration detection. FFTs are also used in digital filtering, signal decomposition, and in solution of partial differential equations. The performance of these applications relies heavily on the availability of a fast routine for Fourier transforms.

In our design of Fast Fourier Transform on the Cell (FFTC) we use an iterative out-of-place approach to solve 1D, 2D FFTs with 1K to 16K complex input samples. In this paper we will focus on better performance. In
implementation we will use range of complex inputs and also this paper illustrates both coding and optimization approaches for efficiently implementing the FFT algorithm in pMatlab.

**Literature Review**

In commenting on Charles Van Loan’s seminal book [1] Computational Frameworks for the Fast Fourier Transform, David Bailey wrote in the journal SIAM Review that “the FFT has found application in almost every field of modern science, including fields as diverse as astronomy, acoustics, image processing, fluid dynamics, petroleum exploration, medicine, and quantum physics. It is not an exercise in hyperbole to say that the world as we know it would be different without the FFT.” [2]. This observation, made over a decade ago, is ever more valid today. The application area of interest to us involves detection, classification and tracking of underwater targets [3]. To fully exploit the information contained in data measured by these novel devices requires use of unconventional algorithms that exhibit growing computational complexity (function of the number of acoustic channels and the number of complex samples in each observation window). For example, signal whitening has long been recognized as an essential stage in processing sonar array data [4]. The unconventional twice whitening paradigm includes the inversion of the spatiotemporal covariance matrix for ambient noise via unitary Fourier factorization [3]. The computational challenge one faces in such an endeavor stems from the following considerations. An array consisting of Ne acoustic channels, capturing Nt complex samples per time window per channel, (Ne ~ 103, Nt ~ 104) yields a spatio-temporal covariance matrix of size 107 × 107 for diffuse ambient noise. Its inversion involves 3 × 103 16K -point complex FFTs [3]. This, in turn, translates into computational throughput that cannot readily be met with conventional hardware. In such a context, the emergence of streaming multicore processors with multi-SIMD architectures (e.g., the IBM Cell [5], the NVIDIA Tesla [6]), or with ultra-low power operation combined with real-time compute and I/O reconfigurability (Coherent Logix “HyperlX” [7]), opens unprecedented opportunities for executing many sophisticated signal processing algorithms, including FFTs, faster and within a much lower energy budget. Discrete Fourier transforms are primarily used in signal processing. The discrete Fourier transform can be computed efficiently using the FFT algorithm. FFTs are also used in scientific and statistical applications, such as detecting periodic fluctuations in stock prices and analyzing seismographic information to take “sonograms” of the inside of the Earth [8]. Due to the vast usage of FFT different algorithms have been developed over time. We will discuss some of the FFT algorithms which are currently being used.

**Designing FFT on multicore architecture**

**Fast Fourier Transform**

FFT is an efficient algorithm that is used for computing the DFT. Some of the important application areas of FFTs have been mentioned in the previous section. There are several algorithmic variants of the FFTs for parallel processors and vector architectures. In our design we utilize the naïve Cooley-Tukey radix-2 Decimate in Frequency (DIF) algorithm. Fig. 1 shows the butterfly stages of this algorithm for an input of 16 sample points (4 stages). The total computations in all stages are N log N which makes the total FLOP count for the algorithm as 5N log N.
Parallel Computing in Multicore Architecture

Figure 1. Batteries of the ordered DIF FFT algorithm

On the basis of such considerations, the algorithm uses a different color image multiplied by the weighting coefficients of different ways to solve the visual distortion, and by embedding the watermark, wavelet coefficients of many ways, enhance the robustness of the watermark.

Multicore Architecture

Four parameters are of paramount importance when evaluating the relevance of emerging computational platforms for time-critical, embedded applications. They are computational speed, communication speed (the I/O and inter-processor data transfer rates), the power dissipated (measured in pico-Joules (pJ) per floating point operation), and the processor footprint. For each of these parameters, one can compare the performance of an algorithm for different hardware platforms and software (e.g., compilers) tools. Multicore processors of particular relevance for many computationally intensive maritime sensing applications include the IBM Cell [5], the Coherent Logix HyperX [7], and the NVIDIA Tesla, Fermi, Ion and Tegra devices [6]. In this paper, we focus on the IBM Cell.

Implementing FFT on multicore architecture:

This section first reviews the fundamentals of the FFT algorithm including its computation, data movement and data preparation. These three components provide the framework for high-performance design of FFTs. These techniques have been proven to provide significant performance improvement in many FFT implementations. The same paradigm can be similarly applied to the development of other types of multicore applications. The principles of automatic FFT generation leverage the strategies for efficient design of an FFT over the available computational resources. And finally, a case study of a large 3-D FFT across a large cluster of multicore systems is presented.

Computational Aspects of FFT Algorithms

A Fast Fourier Transform is one of the effective algorithm for computing Discrete Fourier transforms (DFTs). The size-N Discrete Fourier transform is defined by the formula following where \( x_0, x_1, ..., x_{N-1} \) are complex numbers.

\[
X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi nk}{N}} \quad k = 0, ..., N-1 \quad (2.1)
\]
In the case of radix-2, decimation-in-time, Fast Fourier Transform, the Discrete Fourier transform formula is re-expressed as two size-N/2 Discrete Fourier transform from the even-number-indexed elements \( E_j \) and the odd-number-indexed elements \( O_j \).

\[
X_k = \sum_{m=0}^{N-1} X_{2m} e^{-\frac{2\pi i}{N} (2m)k} + \sum_{m=0}^{N-1} X_{2m+1} e^{-\frac{2\pi i}{N} (2m+1)k}
\]

\[
= \sum_{m=0}^{M-1} X_{2m} e^{-\frac{2\pi i}{M} mk} + e^{-\frac{2\pi i}{N} k} \sum_{m=0}^{M-1} X_{2m+1} e^{-\frac{2\pi i}{M} mk}
\]

\[
= \begin{cases} 
E_k + e^{-\frac{2\pi i}{N} kO_k} & \text{for } k < M \\
E_{k-M} - e^{-\frac{2\pi i}{N} (k-M)O_{k-M}} & \text{for } k \geq M
\end{cases}
\]

Reason of the periodicity of the origin of unity, it holds true that \( E_{k-M} = E_k \) and \( O_{k-M} = O_k \). This equation provides a recursive way to explain the original O\((N^2)\) Discrete Fourier transform computational complexity and yield an O\((N \log_2 N)\) complexity. The two size-N/2 Discrete Fourier transform come from the even and odd elements, respectively, along the time series. This type of Fast Fourier Transforms is called Decimation-in-time (DIT). There are another form of Fast Fourier Transform is called decimation-in-frequency (DIF) [20]. Which has exactly the similar type of computational complexity? Regardless of the choice of Fast Fourier Transform, Decimation-in-time or Discrete Fourier transform, the computation consists of repeatedly applying a simple computational kernel over an array of elements as defined by the equation no [2.2]. This computation kernel is customarily represented as a buttery data flow graph as shown in Figure 2. By combining these butties with their complex roots of unity \( W \) (traditionally referred to as twiddle factors [22]), a complete Fast Fourier Transform is achieved.

**Figure 2. Radix-2 buttery dataflow graph.**

**FFT Performance**

For modern multicore processors which support fused multiply-add (FMA) instructions, there is a good choice. The simplest radix-2 FFT algorithm not only offers the highest usage of FMA instructions, its simplicity in implementation also serves as a good vehicle to demonstrate a practical optimization approach for multicore platforms [25]. The basic DIT radix-2 buttery is used as the building block for most implementations, where \((a, b)\) is the input, \((C, D)\) is the output and \(W_i\) is the twiddle factor. Without FMA instructions, a buttery requires 10 floating point real number add or multiply operations. With FMA instructions, the buttery is reduced to 6 floating-points multiply-add operations.

\[
D = (a - b) \ast W_n^n - b \ast W_n^n
\]
Parallel Computing in Multicore Architecture

\[ D_i = (a_i - b_i * W^w_{NR}) - b_r * W^w_{Ni} \quad (2.4) \]

\[ C_r = 2.0 * a_r - D_r \quad (2.5) \]

\[ C_i = 2.0 * a_i - D_i \quad (2.6) \]

Subscripts \( r \) and \( i \) denote real and imaginary parts of a complex number, respectively.

Data Movement and Preparation of FFT Algorithms

Another aspect that affects the FFT performance is the overhead of moving and preparing data for the butters. Such overhead is not directly expressed in the formulas in equation (2.2) and has often been ignored by many FFT studies. Because the size of the input array of an FFT is usually larger than that of the processor's register file, the data have to spill over more than one level of memory hierarchy. Moving data among the levels of the memory hierarchy is inevitable. This data movement overhead can affect final performance more than the computational portion on many of today's computer systems.

Multicore FFT Performance Optimization

Because data movement is a significant factor affecting performance, it is important to optimize the FFT stages according to the system's memory hierarchy. At each level in the hierarchy, one should exploit all available hardware parallelism and at the same time minimize the effect of data access to the next level of memory hierarchy. By using this heuristic approach, one can achieve a performance close to the upper bound previously established.

One hardware parallelism technique proven to be effective is exploiting concurrency between the computational operations and data movement operations. Such a feature is common in modern CPU design. By allowing the core to move data while computing on another block of data, the overhead of data movement can be mostly hidden behind the time the computational instructions are being executed. Effectively, this results in virtually no overhead for data movement and near-optimal performance is achieved. Such concurrency exists in all levels of the memory hierarchy of Cell/B.E. architecture [24]. At the register level, the Synergistic Processing Element (SPE) load and store instructions are executed on an instruction pipeline which is different from the one used by the computational instructions. For large FFTs that cannot completely fit into one level of memory, the principle of general factorizations of the Cooley-Tukey FFT algorithm can be applied to allow the larger FFT to be divided into a set of smaller sub-FFTs for execution on the specific memory hierarchy level. Because the computational complexity of FFT is \( O(N\log_2N) \), the larger the \( N \) that can execute on a particular level, the less data, \( O(1/\log_2N) \), that need to be transferred per computation unit. Therefore, it is best to execute as large an FFT as possible on each memory level.

To benchmark our performance, we have also implemented the FFT algorithm in pMatlab (pMatlab version is pMatlab_v2.0.1). Both coding and optimization steps for efficiently implementing the FFT algorithms are discussed briefly in appendix. Figure 3 shows the multicore performance on an eight-core system. The first test is to compare the performance of running the program using \( NP = 1 \) with distributed arrays turned off (PARALLEL=0) and distributed arrays turned on (PARALLEL=1). Here, \( NP \) is the number of copies or instances of the running program.

Absolute launch time, relative allocation, computation, communication, and run times, as well as the relative performance for the FFT benchmark using an eight-core processor. All values are normalized to column 2 in which \( NP = 1 \) with the distributed arrays turned off (PARALLEL=0). Columns 3-6 are the values with distributed arrays turned on (PARALLEL=1). The notation \( 1*8 \) means that \( NP = 8 \), and all PID's were run on the same processing node with the same shared memory. Each program is assigned a unique processor ID which is denoted PID, which ranges from 0 to \( NP - 1 \).
If using distributed arrays with NP = 1 slows things down significantly, then this must be addressed. It is very difficult to get any benefit with NP > 1 if the NP = 1 case is slower than the base serial code. The launch time row shows a generally increasing launch time with NP. Ideally, it would be a constant value that is independent of NP. However, depending upon the underlying launch mechanism, this time can be an increasing function of NP. Typically, launch time will not be a large fraction of the overall computation, but it can easily become significant when NP is large. For example, suppose launch time was proportional to NP and each additional PID added 1 second. If NP = 100, then the launch time would be 100 seconds. If this is the Amdahl fraction of the program, then the program will need to run for at least N2P seconds in order to achieve a 50x speedup over the serial case. The relative allocation time row shows that allocation generally decreases with increasing NP because each PID need only allocate N= NP data elements. The measured decrease in allocation time is not linear with NP because there are some overheads associated with allocation. Typically, allocation time will not be a large fraction of the overall computation. However, for cases in which NP is large, these overheads can become important, and so it is always good to measure allocation time and make sure it is reasonable. The relative compute time generally decreases linearly with NP because each core is working on a smaller problem. The relative communication time is computed with respect to the NP = 1 compute time. Thus, for NP = 2, the relative communication time is equal to the compute time, resulting in a net decrease in overall performance compared to the NP = 1 case. As NP increases, the communication time decreases and a net decrease in the relative compute time is observed. The relative performance row indicates that a speedup of 2.7 with NP = 8 is achieved. This is not unexpected given the large amount of communication that is incurred. In some instances, this performance improvement might be advantageous; in other circumstances, it may not be worth the added programming complexity. In general, the parallel FFT often falls in this middle performance regime. The relative bandwidth row shows that as NP increases, the bandwidth among the processors increases. In fact, increasing the NP by a factor of 4 (i.e., going from NP = 2 to NP) results in a bandwidth increase of 3.3. This increase in bandwidth with NP is the only reason that the FFT achieves any parallel performance. If the bandwidth did not improve, then the overall communication time would remain nearly the same as the NP = 1 computation time.

The new parallel resources of the multicore architectures introduce a new programming challenge that has not been well supported by today's traditional programming languages. New language constructs are required for developers to specify performance design knowledge. Such knowledge is critical for a multicore compiler to map an algorithm and schedule operations effectively on the available computational resources. A successful implementation depends upon a deep understanding of the data access patterns, computation properties and available hardware resources. Applications like FFTs with regular data access and computation patterns can readily take advantage of generalized performance planning techniques to produce successful implementations across a wide variety of multicore architectures.
III. CONCLUSION

In future I want to improve my system where the user can improve the performance of the FFT by increasing any number of processors. And in the system users can perform 2D as well as 3D mathematical equations. The general FFT program can works with any number of input points. To achieve better timings, systems can be implemented in such a way so that that computation speed will increase and decrease the timing.

REFERENCE

[7] M. Stolka (Coherent Logix), personal communication; see also: www.coherentlogix.com

ISSN 2277-1956/V3N1-37-43