Scientific Journal of Impact Factor (SJIF): 5.71

e-ISSN : 2348-4470
p-ISSN : 2348-6406

International Journal of Advance Engineering and Research Development

Volume 5, Issue 06, June -2018

# MULTITHREADING AN EXPERT METHOD FOR INCREASING PERFORMANCE OF A SYSTEM

[1]ISHA CHOUDHARY, [2]MR.PARAMVEER SINGH GILL

*Chandigarh Engineering College Landran Mohali(Punjab)*
*Assistant Lecturer , Chandigarh Engineering College Landran Mohali(Punjab)*

**ABSTRACT:-** *Simultaneous multithreading represents an attractive option for mainstream processors because it provides a smooth migration path from today's computer usage patterns. Because the impact on single-thread execution is low and the negative impact on performance in the worst case will be nearly unnoticeable. A user need only have multiple threads or applications running a small fraction of the time to overcome that cost. Simply threading the operating system may be enough to produce noticeable improvements in that scenario. This thesis discusses about the threading tool which evolves total eight machines with multiple number of cores with different system specifications for calculating execution time. Six different tasks i.e. LU Decomposition, Ordinary Differential Equation, Fast Fourier Transform, sparse, 2-D and 3-D are performed on all the machines and results has been presented. It is observed that multithreading leads to tune the application performance considerably. In this thesis we have presented a comparative study of multicore and multiprocessor systems based on power and performance. Each type of architecture is suitable for different type of task execution. Architecture utilized by multicore processor could process multiple instructions at very fast speed of a single huge code and improving execution speed whereas a multi-processor is capable of executing multiple programs in parallel and speeding up execution.*

***KEYWORDS***-*Multithreaded Processor, Scheduling, Matching,*

## INTRODUCTION

### 1.1 Introduction

Multithreaded processors aim to combine control-flow and data-flow ideas to form an amalgam which exhibits many of the advantages of both paradigms whilst trying to avoid the disadvantages. The primary characteristics are that some form of control-flow execution is supported together with hardware mechanisms to assist concurrent execution.

### 1.2 Multiple Contexts

A simple approach to context switching quickly is to avoid having context to switch. For example, the T800 INMOS Transporter only has six registers worth of state - three words of evaluation stack, an operand register, a work space pointer (WP) and a program counter (PC). Furthermore, most context switches can only occur at certain instructions (e.g. a jump) where, by definition, the workspace and operand registers may be discarded. Thus, only the WP and PC have to be saved which may be performed quickly. However, having so little state associated with a thread results in context being continually moved to and from memory rather than making efficient use of a closely coupled store. It simply takes the approach of loading state into a register file at the beginning of a micro thread and then saving it again before being rescheduled. However, in practice the size of a cached page able register file is severely limited because it has to have multiple data paths and still perform at the rate of the rest of the pipeline. For an efficient context switch, the micro thread's code (or text) must be available locally to the processor. One could hope that the code was still present in a local cache from a previous execution. However, assuming a hardware scheduler is present; it is possible to preload code before a process is queued for execution.

### 1.3 COMMUNICATION

Interprocessor communication may be supported by remote memory requests that utilize the usual memory access mechanism but memory may also be tagged with presence bits at each word to indicate whether the word is empty or full to assist synchronization. Communication mechanisms need to be efficient and are, therefore, often positioned very close to the processor to allow transfer of messages to and from the processor's register file and a message processor's input and output queues.

### 1.4 Synchronization and Scheduling

EM-4 and Monsoon utilize tagged memory to synchronies messages to dyadic micro threads in the usual data-flow manner. Whilst this is an efficient mechanism, it is still expensive when compared to the amount of work required to execute many dyadic micro threads.

Tetra [4] has four methods of using tagged memory:
1. Wait for full
2. Read and set empty
3. Wait for empty
4. Write and set full

When wait operations hit the memory the presence bit is returned to the particular processor's scheduling function unit (SFU) which polls the memory (up to a given maximum number of times) until the desired answer is returned. The SFU holds the process status word (PSW) and uses this information to reactivate a thread when the desired value has been returned by the memory. Whilst this is a simple mechanism, polling is inefficient even if a task has to wait just a few thousand clock cycles. HEP uses counter for synchronization and provide a join instruction which decrements a counter at a particular address and if the result is not zero then the thread is disc hauled. This mechanism is also simple but assumes that an atomic read/modify/write cycle can be performed on a counter. This is inefficient if one assumes the counter is stored in a memory with long access latency. MDFA also uses counters for synchronization but has a separate event coprocessor to manipulate a signal graph in a static data-flow manner. Like HEP, updating an event counter unfortunately assumes a low latency memory (e.g. as provided by a cache). However, having a separate signal graph is an interesting idea. Each node in the signal graph consists of an event counter, a reset value for the event counter, acknowledgement addresses for backward signaling, forward signaling addresses and a code pointer. When signals arrive at a node the event counter is decremented. Once the event counter reaches zero, it is reset to the reset value and the micro thread pointed to by the code pointer is executed. Upon completion of the micro thread a signal is returned to the node which then sends the forward and backward signals within the signal graph [5]. Does not attempt to use concurrency to tolerate latency so does not need to synchronies and schedule on memory accesses. However, it does need to synchronies on incoming messages from other processors, either using conventional interrupts or by polling the message coprocessor. Thus, matching messages to threads is a software overhead. If hardware support is provided for scheduling, then the prioritizing mechanism is usually just in the form of a few FIFO or LIFO queues. For example, the Transporter has two priorities of FIFO queue. This is inadequate for hard real time and multimedia applications.

**1.5 Memory**

Typically faster processors are used to tackle larger problems which require bigger memories which prevent memory latency from scaling with processor performance. However, it is possible to scale memory access frequency with processor performance provided a pipelined memory structure is used. Some multithreaded processors [2] take the control-flow solution of adding caches despite the side effect of temporal non determinism. Many processors all allow concurrency to be used to hide access latency to local and remote memories. Alewife [2], with its higher context switch overhead, only supports latency tolerance of remote memory. Memory protection and virtual address translation on most current multithreaded processors relies on a translation lookaside buffer (TLB). However, a TLB adds temporal non determinism and becomes inefficient when the number of threads reaches and exceeds the number of TLB entries. Tera [4] uses its memory latency tolerant characteristics to allow a pipelined memory protection and address translation mechanism to be efficiently used. Using pipelining allows the mechanism to be much larger than a conventional TLB. However, it still does not provide total memory coverage because the structure would be prohibitively large.

**1.6 Micro thread Size**

The data-flow oriented machines can perform efficiently with micro threads which are only one instruction long. Although HEP [3] is more control-flow oriented, it too deals with single instruction micro threads one instruction is picked in FIFO order from each of the runnable threads and is inserted into the processor's pipeline. Thus, if there are lots of threads then each stage of the pipeline will be executing an instruction from a different thread. However, if there are few threads then, assuming data dependencies are not violated; several instructions from the same thread may be in the pipeline at one time. Other processors all execute a single micro thread at a time which is usually several instructions long. The desirable length of a micro thread for a particular processor is dependent upon the efficiency of the context switch mechanism and whether a micro thread is forced to be descheduled to access local or remote memory. If a micro thread is forced to be short, then a large number of thread [4] are required to ensure that there is sufficient work for the processor when some threads are waiting due to memory access latency. However, if micro threads are larger than obviously fewer are required to hide memory latency. This is advantageous for the many algorithms which exhibit little parallelism.

**1.7 Multithreaded processor**

Multithreaded processor is designed to execute data driven micro threads, which can also be thought of as large grained data-flow where each data-flow node is a micro thread. Each micro thread consists of a control-flow routine with between 8 and 32 instructions. Micro threads have up to 16 input parameters which must be presented before execution can commence. An instance of a micro thread stores its parameters in an activation frame. Thus, there is a similar relationship between micro

threads and activation frames as there is between functions and stack frames on a control-flow processor. A matching store is provided for joining parameters to micro threads by writing them to the appropriate micro thread's activation frame and recording which parameters have been written. When an activation frame is full it is scheduled by the hardware.
A large sequential routine may be broken up into a number of sequentially ordered micro threads to form one logical thread of control. Only one activation frame is required when executing a single sequence of micro threads because only one micro thread is active at time so the activation frame may be reused. Communication with memory and I/O is supported by posting messages. Stores simply post a write request to the memory system. Loads do not stall but instead are performed split phase: one micro thread posts a memory request and specifies a destination micro thread using an address into its activation frame.

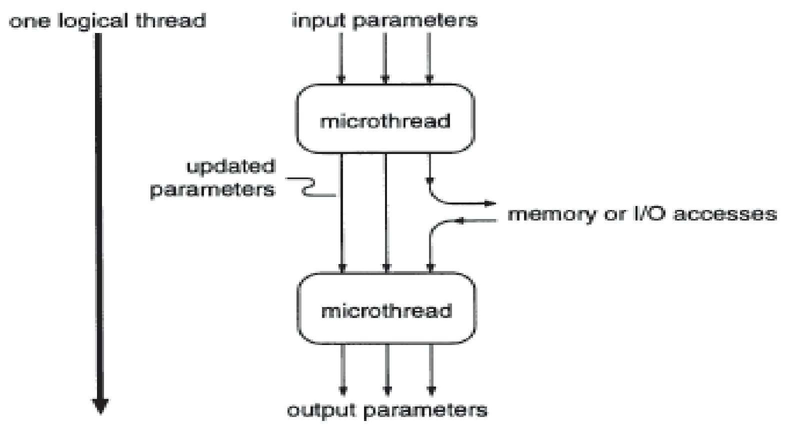

**Figure 1.1 Single logical thread constructed from a sequence of micro threads**

Thus, the data loaded is sent as the input parameter to an awaiting micro thread. Furthermore, the micro thread which initiates the transaction does not need to stall awaiting the memory response.
Conventional multithreaded programs may be constructed from micro threads. The Figure demonstrates the micro thread structure for one thread spawning two more threads and then waiting for them to complete before proceeding. In this instance, just three activation frames are used, one for each thread.

A more data-flow oriented style may also be supported. Figure 1.4 illustrates a data-flow styled bubble sort constructed from min/max micro threads which accept 10 parameters as input, and output the lowest 5 parameters to the left and the 5 highest parameters to the right.

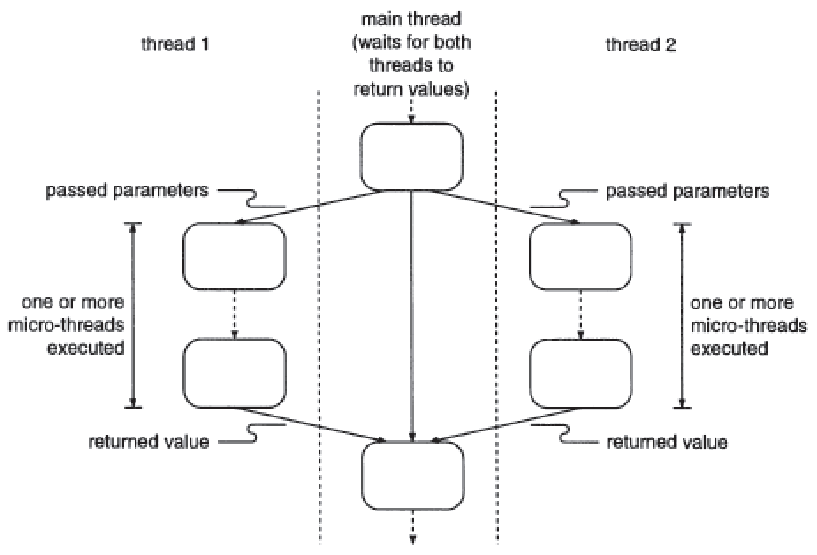
thread 1
main thread (waits for both threads to return values)
thread 2
passed parameters
passed parameters
one or more micro-threads executed
one or more micro-threads executed
returned value
returned value

**Figure 1.2 Example micro thread structure for forks and joins**

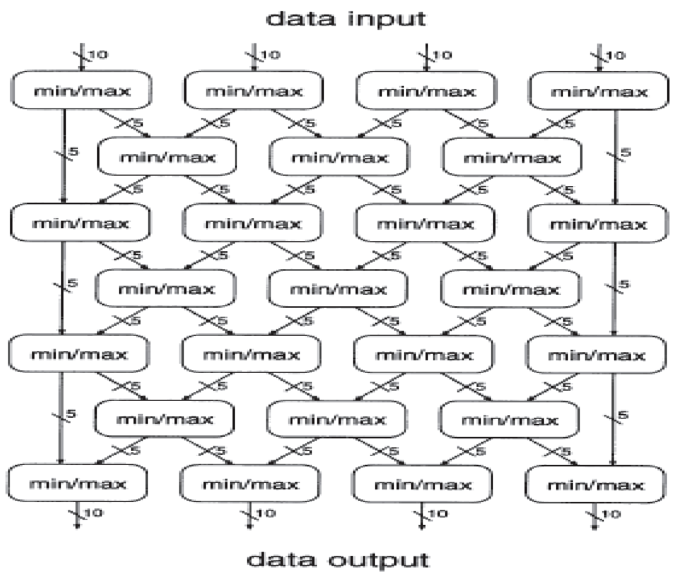

**Figure 1.3 Data-flow styled bubble sort**

**Related works-**

- **M.Shanthi & Dr.A.Anthony Irudhayaraj (2009)** presented the implementation of MI based image registration using parallel computing. An explicit multithreaded approach was developed for MI based image registration for prompt and proficient result. There research work could be easily implemented on multi core processor which are easily available in the market and produce tremendous result. Image Registration based on Mutual Information is time consuming processes as it is having several task to be executed. They have assigned individual task to each activated cores after maintaining the synchronization between them. After execution of the particular task assigned core/processor is free to execute another existing task in queue. Main advantage of there work is that we have parallelized all the time consuming steps of MI based image registration as a result we can see that the performance in terms of speed up of the developed approach is extensively excellent. Another major advantage of the used method was that this is able to work on multi core processor having no GPUs.

- **Hsunwei Hsiung & Sandeep K. Gupta (2015)** proposed a defect-tolerance methodology based on exploiting implicit redundancy in processors and use it to derive defect-tolerance approaches for datapath modules. There multilayered methodology identifies DT approaches that maximize the performance-per-area for processors by minimizing area and performance overheads of defect-free processors and maximizing the yield and performance benefits provided by the defective processors. They have demonstrated that our approaches provide substantial performance-per-area improvement. Averaging across the benchmarks, our first two approaches increase the performance-per-area 3.6% and 2.4% respectively for the multiplier and the ALUs.

- **Sangok Seok et. al. (2014)** presented a control system platform architecture developed for multi-degrees of freedom (DoFs) robots capable of highly dynamic movements. In robotic applications that require rapid physical interactions with the environment, it is critical for the robot to achieve a high frequency synchronization of data processing from a large number of high-bandwidth actuators and sensors. To address this important problem in robotics, They developed a control system architecture that effectively utilizes the advantages of modern parallel real-time computing technologies.

- **Jhi-Young Joo & Marija D. (2014)** proposed a novel set of methods for coordinating supply and demand over different time horizons, namely day-ahead scheduling and real-time adjustment. They illustrated the ideas by simulating simple examples with different conditions and objectives of each entity in the system. Mathematical conditions under which a system-level optimization of supply and demand scheduling could be implemented as a distributed optimization in which users and suppliers, as well as the load serving entities, are decision makers with well-defined sub-objectives.

- **Thakur and R. Thakur (2015**) emphasized on the performance and efficiency which could be achieved by using multicore together with parallel programming. Multicore technology offers more than one core that is used to execute multiple tasks at the same time. Whereas parallel programming offer the algorithm, which is used to distribute the complex task in smaller instructions. These instructions are than executed on different cores. Performance of the system depends upon how efficiently the parallel mechanism has been implemented in the multicore of system. Parallel programming in the multicore platform increases the operating efficiency and performance of a system and application to a greater extent.

- **Yaser Ahangari Nanehkaran and Sajjad Bagheri Baba Ahmadi (2013)** reported on the basic concept of multi-core processors, a sample of Dual-core Processors in Intel and AMD, and its advantages The most important aspects of challenge in this method. However, Before multicore processors the performance increase from generation to generation was easy to see, an increase in frequency. The proposed model broke when the high frequencies caused processors to run at speeds that caused increased power consumption and heat dissipation at detrimental levels. Adding multiple cores within a processor gave the solution of running at lower frequencies, but added interesting new problems. Multi-core processors are architected to adhere to reasonable power consumption, heat dissipation, and cache coherence protocols. But numerous issues remain unsolved. In order to use a multi-core processor at full capacity the applications run on the system must be multithreaded. There are relatively few applications written with any level of parallelism. And finally the memory systems and interconnection networks also need improvement.

- **Dina R. Salem et al. (2016**) proposed a resource scheduling algorithm along with a server consolidation algorithm is applied to multi-core processors. It has been presented experimentally that adding cores to the processors in data centers increases the system performance, decreases the power consumption, along with other benefits.

## METHODOLOGY

Test benches is designed for six different MATLAB tasks and their execution speed is compared. Six computational tasks will be ODE, FFT, LU, Sparse, 2D and 3D graphs. Matlab Distributed Computing Toolbox is used to run multiple copies of this stripped-down . A simulation environment that defines an implementation of a simultaneous multithreaded architecture; that architecture is a straightforward extension of next-generation wide superscalar processors, running a multi programmed workload that is highly optimized for single-threaded execution on our specified machine. For our multithreaded experiments, we assume support is added for up to eight hardware contexts. We simulate several models of simultaneous multithreaded execution. In most of our experiments instructions are scheduled in a strict priority order, i.e., context 0 can schedule instructions onto any available functional unit, context 1 can schedule onto any unit unutilized by context 0, etc. Our experiments show that the overall instruction throughput of this scheme and a completely fair scheme are virtually identical for most of our execution models; only the relative speeds of the different threads changes.

## RESULTS AND DISCUSSIONS

### 5.1 Multithreaded parallelism

In multithreaded parallelism multiple processors or cores, sharing the memory of a single computer, execute these streams. A schematic of a typical parallel computing cluster is
shown in figure 5.1. The gray boxes are separate computers, each with its own chassis, power supply, disc drive, network connections, and memory. The light blue boxes are microprocessors. The dark blue boxes within each microprocessor are computational cores. The green boxes are the primary memories. There are several different memory models. In some designs, each core has uniform access to the entire memory. In others, memory access times are not uniform, and our green memory box could be divided into two or four pieces connected to each processor or core.

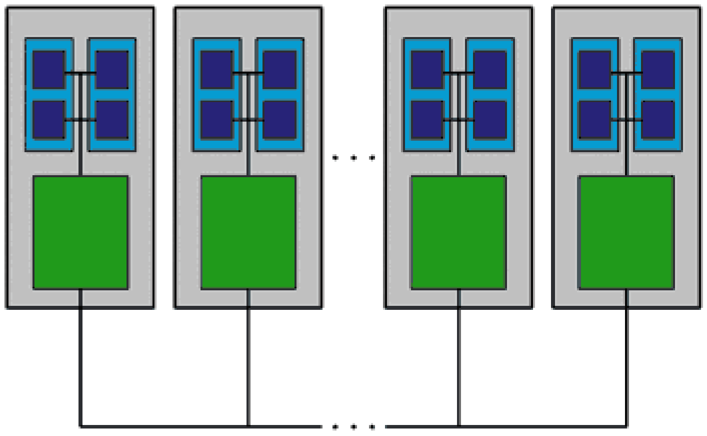

**Figure 1.4. Show a multithreaded parallel computing cluster.**

**5.2 Specifications of Machines**
In the presented work total eight machines are used with multiple number of cores with different system specifications for calculating different execution time . So, that analysis of the different machines could be done. In this work Mat lab software is utilized for the calculation of execution time of all the machines.

**Table : shows the details of different machines used in the work.**

| **Details of Machines used for performance comparison** | |
|---|---|
| Machine 1 | Windows 7, Intel Xeon E5-1650 v3 @ 3.50 GHz |
| | 12 cores, Windows 7 Enterprise, Intel Xeon CPU E5-1650 v3 @ 3.50 GHz, 64 GB RAM, NVIDIA Quadra K620 |
| Machine 2 | Surface Pro 3, Windows 8.1, Intel Core i5-4300U @ 1.9 GHz |
| | 2 cores, Windows 8.1 Enterprise, Intel Core i5-4300U @ 1.9 GHz, 8 GB RAM, Intel HD 4400 Integrated GPU |
| Machine 3 | Mac Book Pro, OS X 10.12.1, Intel Core i5 @ 2.6GHz |
| | 2 cores, OS X 10.12.1, Intel Core I5 @ 2.6GHz, 8 GB RAM, Intel Iris 1536 MB Integrated GPU |
| Machine 4 | Windows 10, Intel Xeon X5650 @ 2.67 GHz |
| | 12 cores, Windows 10, Intel Xeon X5650 @ 2.67 GHz, 24 GB RAM, NVIDIA Quadra FX 380 GPU |
| Machine 5 | iMac, OS X 10.10.5, Intel Core i7 @ 3.4 GHz |
| | 4 cores, OS X 10.10.5, Intel Core i7 @ 3.4 GHz, 16 GB RAM, AMD Radon 6970M GPU |
| Machine 6 | Windows 8, AMD A8-6410 APU @ 2.00 GHz |
| | 4 cores, Windows 8.1 Enterprise, AMD A8-6410 APU @ 2.00 GHz, 6.94 GB RAM, AMD Radeon R5 GPU |
| Machine 7 | Windows 7 Ultimate Intel (R) Core (TM) i3 -4010CPU @ 1.70GHz |
| | 3 cores, Windows 7 Ultimate, Intel (R) Core (TM) i3 -4010CPU @ 1.70GHz 4.00GB RAM |
| Machine 8 | Linux, Intel Xeon CPU W3690 @ 3.47 GHz |
| | 12 cores, Ubuntu 16.04 LTS, Intel Xeon CPU W3690 @ 3.47 GHz, 24 GB RAM, NVIDIA Quadro 400 GPU |

In the presented work total eight machines are used with multiple number of cores with different system specifications for calculating execution time. Six different tasks i.e. LU Decomposition, Ordinary Differential Equation, Fast Fourier Transform, sparse, 2-D and 3-D are performed on all the machines and results has been presented. The used tasks are explained below.

**5.3 Various Tasks Involved**

**5.3.1 LU Decomposition**
A procedure for decomposing an N×N matrix A into a product of a lower triangular matrix L and an Upper Triangular Matrix U.

$$LU = A$$

LU decomposition is implemented the Wolfram Language as LU Decomposition. Written explicitly for a 3×3 matrix, the decomposition is

$$\begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix}\begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & & u_{33} \end{bmatrix}=\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

**5.3.2 Ordinary Differential Equation (ODE)**

Ordinary differential equation (ODEs) is the function ode45. This function implements a Runge-Kutta method with a variable time step for efficient computation. Ode45 is designed to handle the following general problem

$$\frac{dx}{dt} = f\left(t,x\right), \qquad x\left(t_0\right) = x_0 \tag{5.2}$$

Where t is the independent variable, x is a vector of dependent variables to be found and f(t, x) is a function of t and x. The mathematical problem is specified. When the vector of functions on the right-hand side of Eq. (5.2), f(t, x), is set and the initial conditions, x = x (0) at time t (0) are given.

**5.3.3 Fast Fourier Transform (FFT)**

It's an algorithm which computes the discrete Fourier transform of a sequence or its inverse (IFFT). Fourier analysis converts a signal from its original domain (often time or space) to a representation in the frequency domain and vice versa. An FFT rapidly computes such transformations by factorizing the DFT matrix into a product of sparse (mostly zero) factors. As a result it manages to reduce the complexity of computing the DFT from, which arises if one simply applies the definition of DFT to where is the data size. FFT is widely used.

**5.3.4 Sparse**

A sparse matrix or sparse array is a matrix in which most of the elements are zero. By contrast, if most of the elements are nonzero, then the matrix is considered dense. The number of zero-valued elements divided by the total number of elements (e.g., m × n for an m × n matrix) is called the sparsity of the matrix. Theoretically, sparsity corresponds to systems which are loosely coupled. Consider a line of balls connected by springs from one to the next: this is a sparse system as only adjacent balls are coupled. By contrast, if the same line of balls had springs connecting each ball to all other balls, the system would correspond to a dense matrix. The concept of sparsity is useful in combinatory and application areas such as network theory, which have a low density of significant data or connections. When storing and manipulating sparse matrices on a computer, it is beneficial and often necessary to use specialized algorithms and data structures that take advantage of the sparse structure of the matrix. Operations using standard dense-matrix structures and algorithms are slow and inefficient when applied to large sparse matrices as processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus require significantly less storage. Some very large sparse matrices are infeasible to manipulate using standard dense-matrix algorithms.

**5.3.5 Two Dimensional (2-D)**

A 2-D shape is any shape that has two dimensions. It means to have two dimensions for a moment. If we had only one dimension to work with, we could only move backwards or forwards in a line. A line is one-dimensional. If we had two dimensions, on the other hand, we could go forwards and backwards in a line and turn in any direction to start a new line. We are essentially able to travel anywhere on a flat surface. In mathematics, a flat surface is called a plane.

**5.3.6 Three Dimensional (3-D)**

Three-dimensional computer graphics (3-D) computer graphics, in contrast to 2D are graphics that use a three-dimensional representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering 2D images. Such images may be stored for viewing later or displayed in real-time 3D computer graphics rely on many of the same algorithms as 2D computer vector graphics in the wire-frame model and 2D computer raster graphics in the final rendered display. In computer graphics software, the distinction between 2D and 3D is occasionally blurred; 2D applications may use 3D techniques to achieve effects such as lighting, and 3D may use 2D rendering techniques. 3D computer graphics are often referred to as 3D models. Apart from the rendered graphic, the model is contained within the graphical data file. However, there are differences: a 3D model is the mathematical representation of any three-dimensional object. A model is not technically a graphic until it is displayed. A model can be displayed visually as a two-dimensional image through a process called 3D rendering or used in non-graphical computer simulations and calculations. With 3D printing, 3D models are similarly rendered into a 3D physical representation of the model, with limitations to how accurate the rendering can match the virtual model.

**Table : Different Tasks to performed over the machines**

| S.No. | Name of Task | Short form used | Use |
|---|---|---|---|
| 1 | LAPACK | LU | Floating point, Regular Memory Access |
| 2 | Fast Fourier Transform | FFT | Floating point, Irregular Memory Access |
| 3 | Ordinary Differential Equations | ODE | Data Structure and Functions |
| 4 | Solve Sparse System | Sparse | Sparse Linear Algebra |
| 5 | 2-D Lissajous plots | 2-D | Animating line Plots |
| 6 | 3-D SURF (Peaks) and HGTransform | 3-D | 3-D Surface Animation |

### 5.4 Results for different Tasks

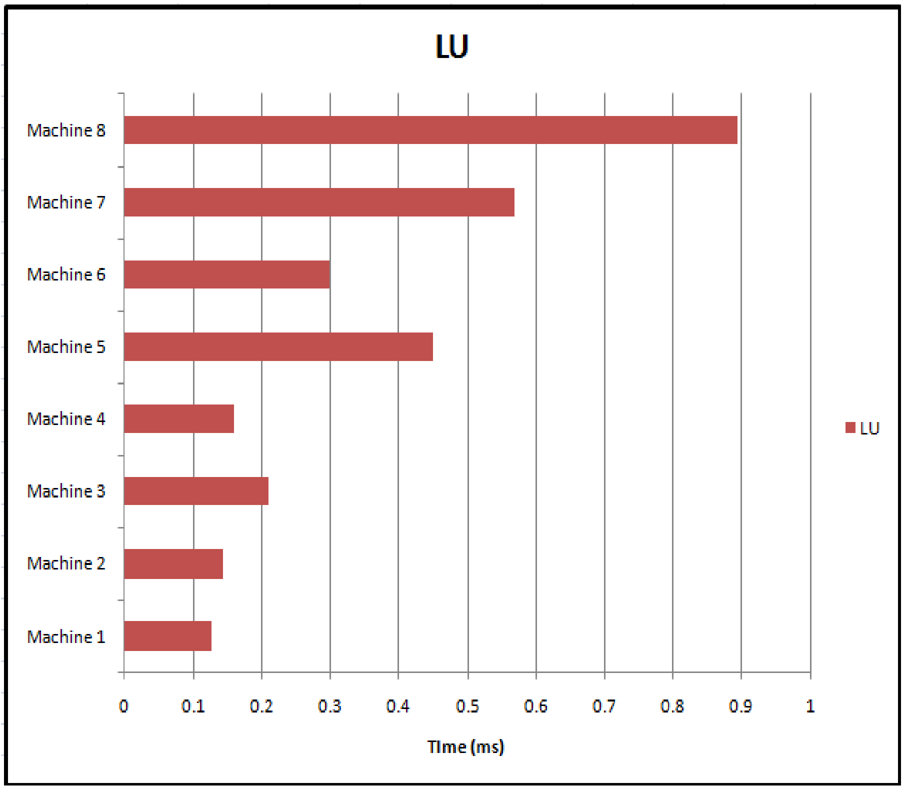

**Fig: show the execution time for LU decomposition over the machines**

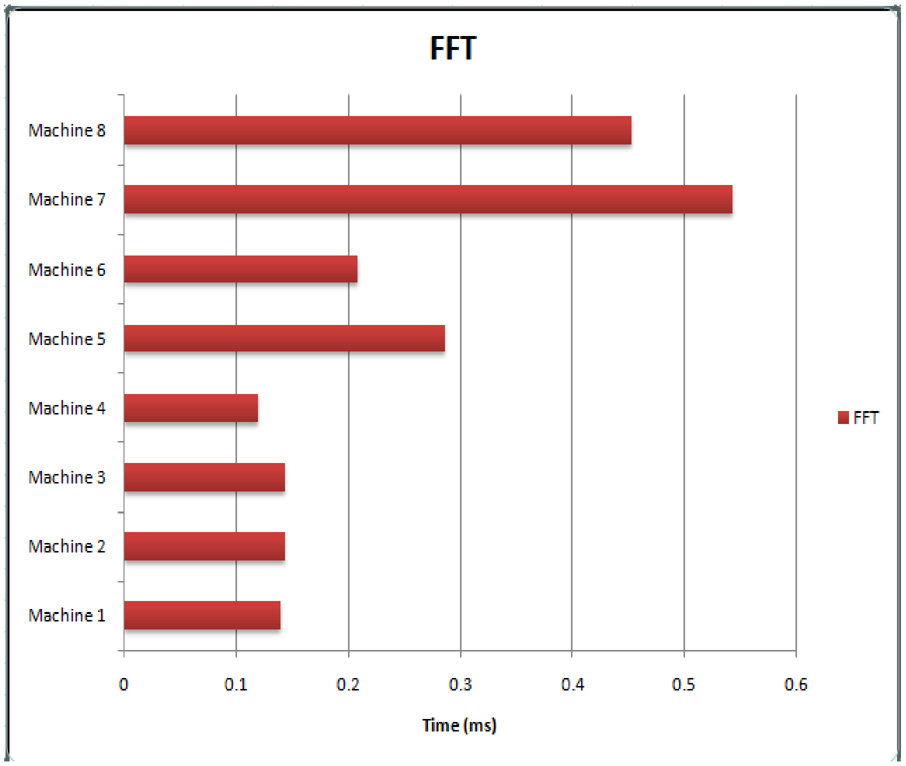

**Figure : Show the execution time for Fast Fourier Transform to Execute over the machines**

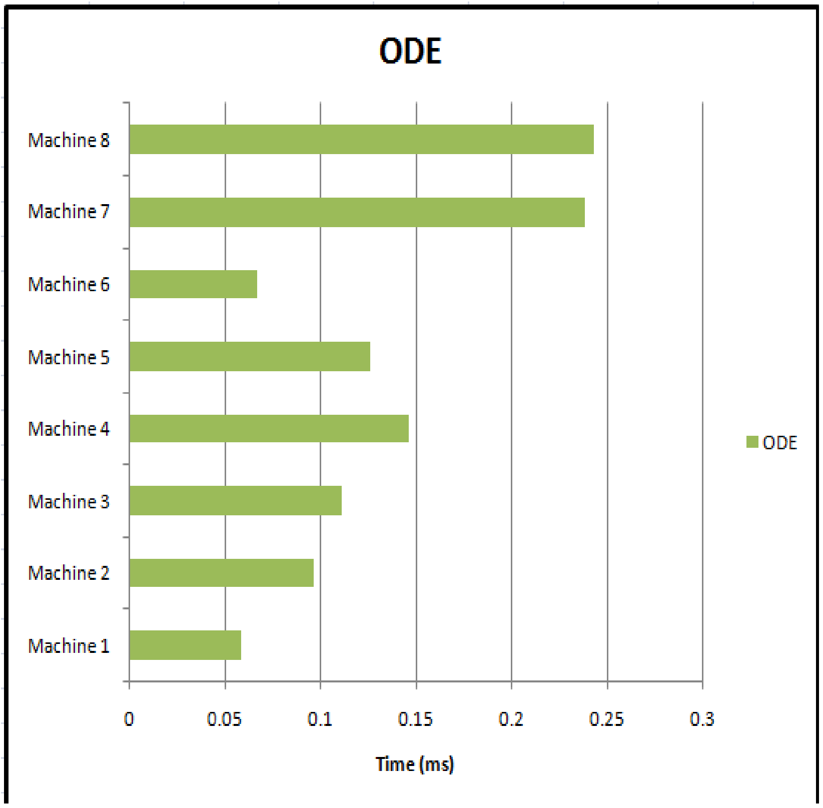

**Figure : Show the execution time for Ordinary Differential Execution over the machine**

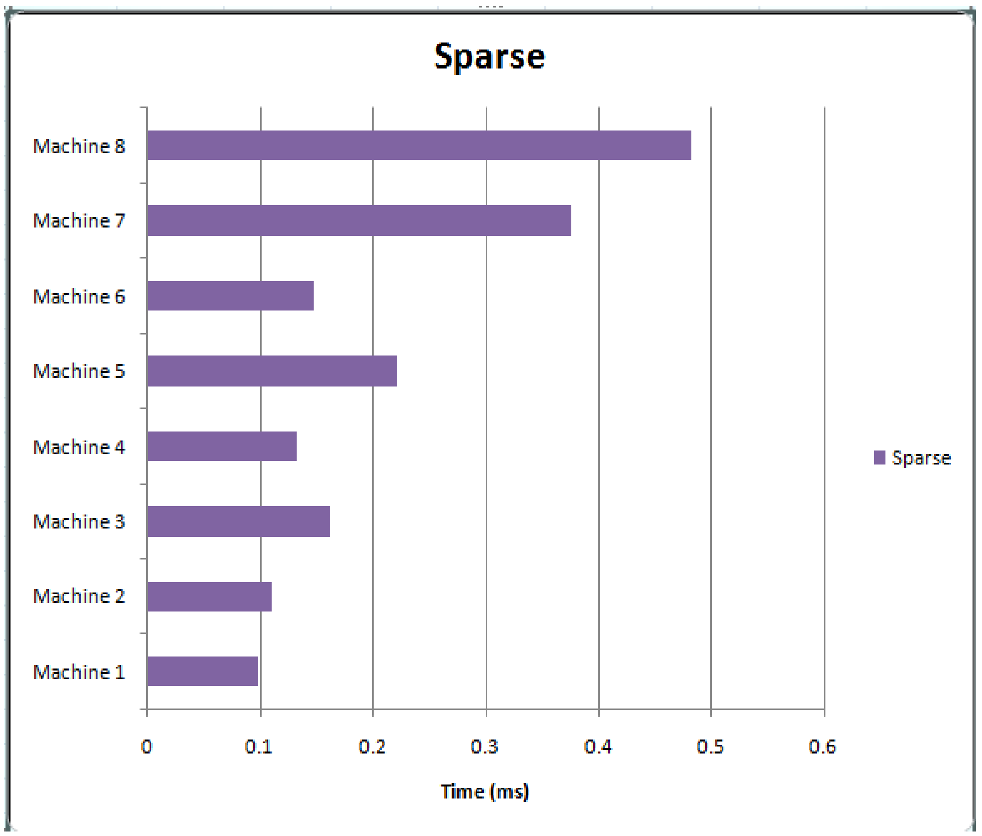

**Figure : Show the execution time for Sparse Fast Fourier Transform to Execute over the machines**

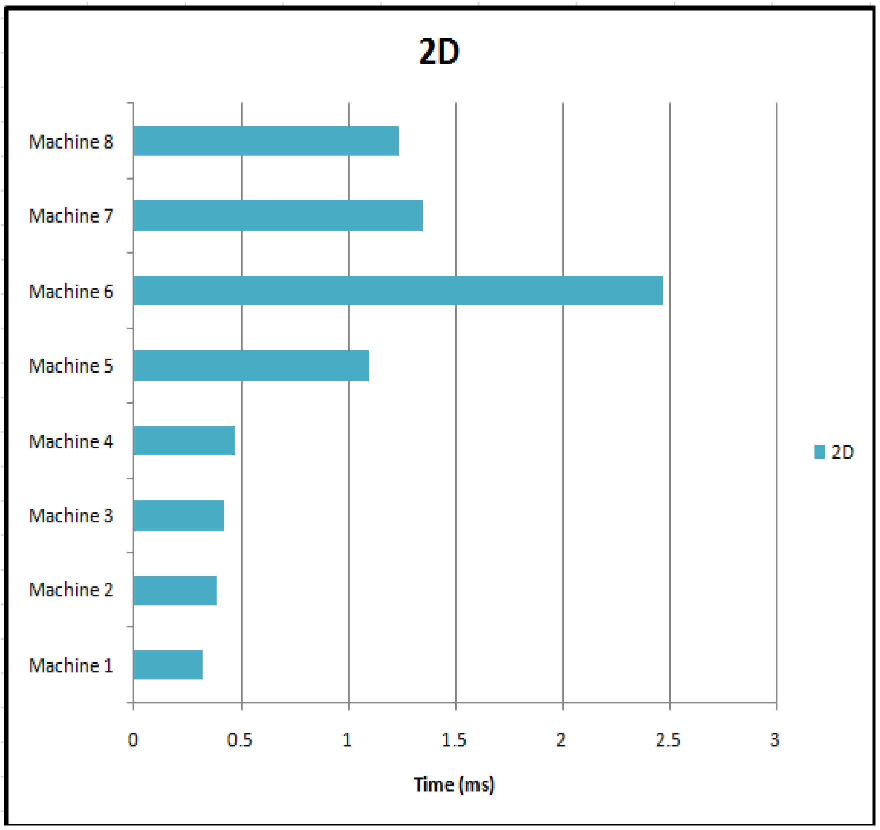

**Figure : Show the execution time for 2-D to Execute over the machines**

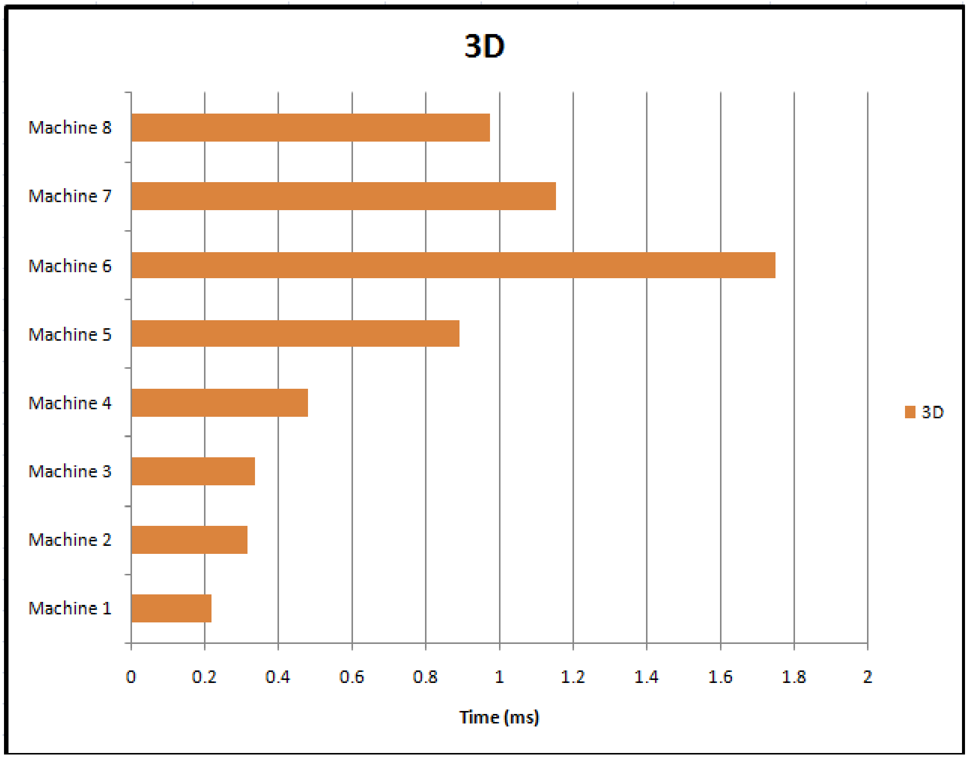

**Figure : Show the execution time for 3-D to Execute over the machines**

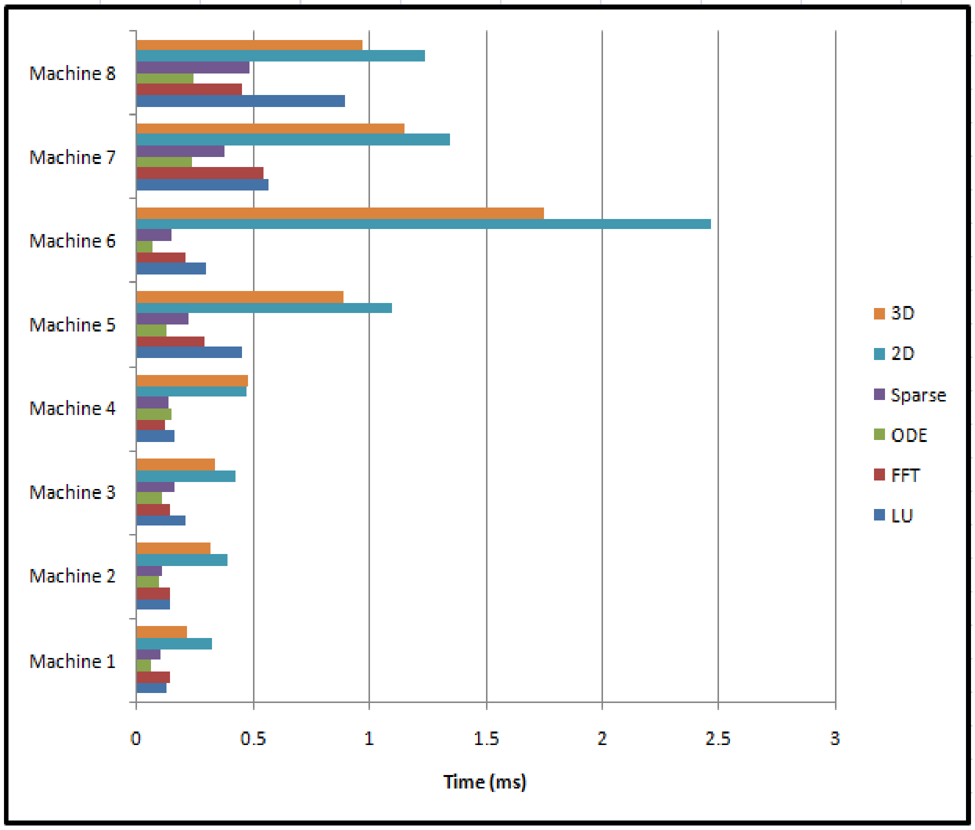

**Figure : Show the Comparison of all the Tasks for execution time to Execute over the machines**

**Table 5.3 Shows the Comparison of all the Tasks for execution time in seconds over the machines**

| | LU | FFT | ODE | Sparse | 2D | 3D |
|---|---|---|---|---|---|---|
| **Machine 1** | 0.1256 | 0.1393 | 0.0581 | 0.0981 | 0.3219 | 0.2165 |
| **Machine 2** | 0.1426 | 0.1433 | 0.0964 | 0.1104 | 0.3875 | 0.316 |
| **Machine 3** | 0.2094 | 0.143 | 0.1111 | 0.162 | 0.4236 | 0.3344 |
| **Machine 4** | 0.1607 | 0.119 | 0.1461 | 0.1319 | 0.4716 | 0.4773 |
| **Machine 5** | 0.4508 | 0.2861 | 0.1256 | 0.2213 | 1.0987 | 0.8906 |
| **Machine 6** | 0.2991 | 0.2086 | 0.0669 | 0.1474 | 2.4693 | 1.7504 |
| **Machine 7** | 0.5681 | 0.543 | 0.2378 | 0.3752 | 1.3463 | 1.1507 |
| **Machine 8** | 0.8945 | 0.4533 | 0.2429 | 0.4822 | 1.2382 | 0.971 |

A final bar chart shows speed, which is inversely proportional to time. Here, longer bars are faster machines, shorter bars are slower

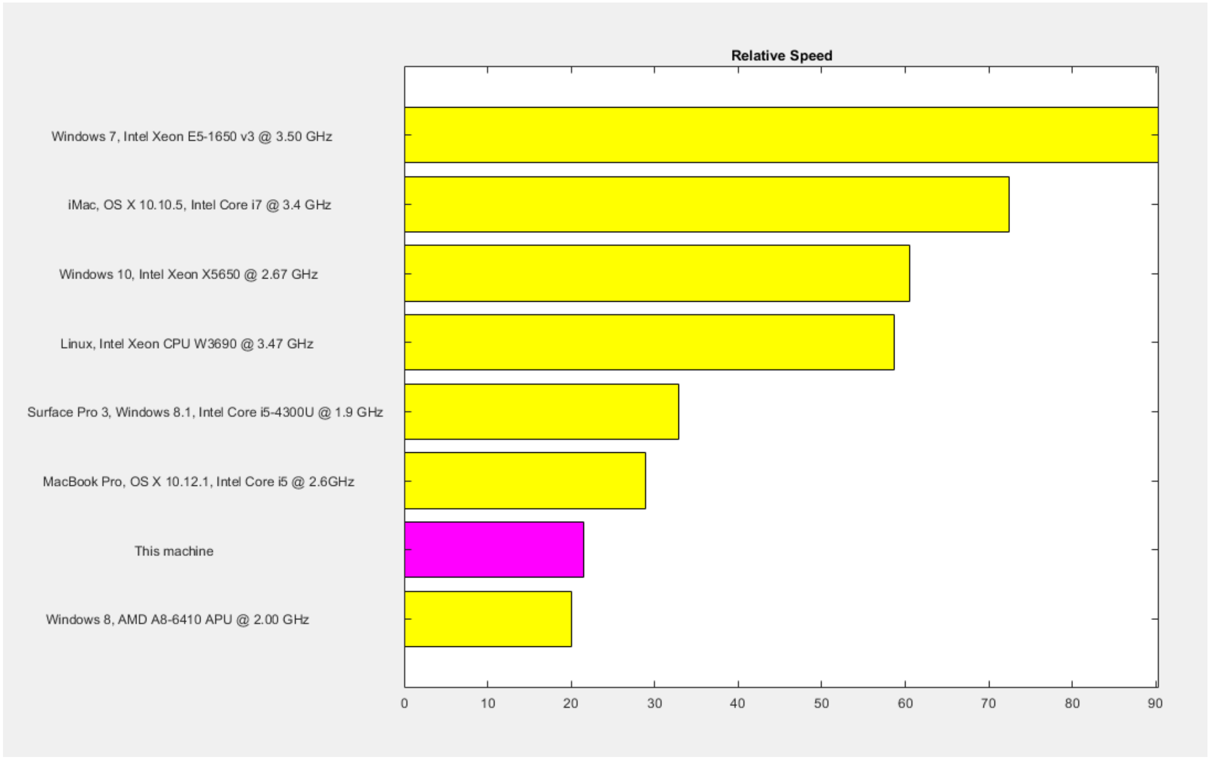

**Figure : Shows the overall Performance Comparison of the Machines**

## CONCLUSION AND FUTURE SCOPE

### Conclusion

Simultaneous multithreading combines a superscalar processor's ability to exploit high degrees of instruction-level parallelism with a multithreaded processor's ability to expose more instruction-level parallelism to the hardware by via inter-thread parallelism. Existing multithreading architectures provide the ability to hide processor latencies; simultaneous multithreading adds to that the ability to tolerate low levels of single-stream instruction parallelism. In this thesis we have presented a comparative study of multicore and multiprocessor systems based on power and performance. Each type of architecture is suitable for different type of task execution. Architecture utilized by multicore processor could process multiple instructions at very fast speed of a single huge code and improving execution speed whereas a multi-processor is capable of executing multiple programs in parallel and speeding up execution. The main application of multi-core processors is found in embedded systems, data, web server or web commerce signal processing.

**Future Scope**

Many techniques already exist such as data, instruction and thread level parallelism and simultaneous multithreading which enhance the performance of different processors. Yet the complete performance throughput can be realized only when the challenges multi-core processors facing in present are fully addressed. A lot of technological breakthroughs are expected in this area of technology together with new multi-core programming language software to port legacy software or multi-core software programs. While it has been one of the most challenging technology to adopt. In future of research could be carried out to utilize multi-core processors more efficiently.

## *References*

1. *J.S. Kowalik (editor). Parallel MIMD computation: the HEP supercomputer and its applications. MIT Press, 1985.*
2. *R. Kalla, B. Sinharoy, and J. M. Tendler. IBM POWER5 Chip: a dual-core multithreaded processor. IEEE Micro, 24(2), 2004.*
3. *C. Batten, "Simplified Vector-thread Architectures for Flexible and Efficient Data-parallel Accelerators, " PhD. Thesis, Massachusetts Institute of Technology, 2010.*
4. *M. Ju, H. Jung, and H. Che, "A performance analysis methodology for multicore, multithreaded processors, " IEEE Transactions on Computers, vol. 63, no. 2, pp. 276-289, 2014.*
5. *Ran Zhang and Hui Guo, "Evaluation of Multi-Threaded Processor Designs for Energy Efficient Embedded Systems," 16th International Conference on Computational Science and Engineering, IEEE 2013.*
6. *Jian Fu, Qiang Yang, Raphael Poss, Chris R. Jesshope, Chunyuan Zhang, "Rethread: A Low-cost Transient Fault Recovery Scheme for Multithreaded Processors," 9th International Conference on Availability, Reliability and Security, IEEE 2014.*
7. *M. Wickramasinghe, Hui Guo, "Energy-Aware Thread Scheduling for Embedded Multi-threaded Processors: Architectural Level Design and Implementation", VLSI (ISVLSI) 2014 IEEE Computer Society Annual Symposium on, pp. 178-183, July 2014.*
8. *Joseph M. Arul, Han-Yao Ko and Hwa-Yuan Chung, "VHDL Implementation of Scheduled Dataflow Architecture and the Impact of Efficient Way of Passing of Data," World Congress on Computing and Communication Technologies, IEEE 2014.*
9. *Mahanama Wickramasinghe and Hui Guo, "Effective Hardware-Level Thread Synchronization for High Performance and Power Efficiency in Application Specific Multi-Threaded Embedded Processors," 33rd IEEE International Conference on Computer Design (ICCD), 18-21 Oct. IEEE 2015.*