**Chapter 1**

**Multi Core Computing**

* 1. **Introduction**

In 1945, mathematician John Von Neumann proposed an architecture in which a program is a sequence of instructions stored sequentially in the computer’s memory. The program’s instructions are executed in a linear single-threaded fashion, one after the other.

The 1960s saw the advent of time sharing operating systems. Run on large mainframe computers, these operating systems first introduced the concept of concurrent program execution. Multiple users could access a single computer simultaneously and submit jobs for processing. From the program’s perspective, it seems to be only process in the system. The operating system handled the details of allocating CPU time for each individual program. At this time, concurrency existed at the process level, and the job of task switching was left to the systems programmer.

With the success of dual core and the advantages it brought, the battle for quad core and beyond has been raging. In the x86 world, quad core processors are shipping today (though some disagree on whether the current quad core processors from the Intel are truly quad core). AMD quad core will be out shortly. Sun is already shipping its 8-core Niagara chip; 8-core will likely come to the x86 market by 2008 or 2009. We are clearly moving down the path to tens or even hundreds of cores. The key to leveraging this hardware trend will of course be in the software. Multi-cores are only as valuable as the multithreading software running in them.

Multi-core processors can deliver significant performance benefits for multithreaded software by adding processing power with minimal latency, given the proximity of the processors. The most significant benefits will be seen in applications such as larger databases, CRM (customer relationship management), ERP (enterprise resource planning), ecommerce amd virtualisation. The more threaded applications will clearly get more benefits. Over time, this trend is beginning to and will shape the future of software development towards parallel processing.

* 1. **Need of Multi-Core Computers**

Today multi-core computers are using everywhere and in the last few years, multi-core CPUs have become a standard component in nearly all sorts of computers. In multi-core technology processors that have two or more working processor chips which also called as cores are working simultaneously as one system. Also the idea of multi-core computers can be used to make parallel computing possible. This technology is used even in the desktop and the laptop PCs for consumers as well as servers and high end workstations. Also game consoles nowadays usually use multi-core technology with CPUs. When considering the development history of this technology in 1965 Gordon Moore predicted that the numbers of transistor that can be cost corrected that assumptions to a period of two years. Nowadays this period is frequently assumed to be 18 months. Increase in transistor density does not always lead to an equal in increase in computing speed but Moore’s projection has more or less been accurate up to today and users have gotten used to the constant speedup of computer hardware. In order to implement the exponential increase of integrated circuits, the transistor structures have to become gradually smaller. The extra transistors were used for the integration of more and more specialized instruction sets on CISC chips and smaller transistor sizes lead to higher clock rates of the CPUs, because due to physical factors, the gates in the transistor could perform faster state switches. Multiple processing cores could be placed in a single processor die, since the CMOS technology met its limits for the additional increases of the CPU clock frequency and the number of transistors that could be integrated on a single die that allowed for it. Significantly increases in the clock speed of processors will not be seen longer. The power consumed by the fastest possible processors generates too much heat to dissipate effectively in known technologies. Instead processor manufactures are adding multiple processors cores to each chip. In 2006, Intel released the core microprocessors, a die package with two processors cores with their own level 1 caches and shared level 2 caches. In 2006, AMD releases the Athlon X2, a processor with similar architecture to the Core platform. The concept of also sharing a CPU- integrated memory controller among the cores is additional feature of this Athlon X2 technology.

Using multi-core processors, Intel can dramatically increase a computer’s capabilities and computing resources, providing better responsiveness, improving multithreaded throughput, and delivering the advantages of parallel computing to properly thread mainstream applications.

While manufacturing technology continues to improve, reducing the size of single gates, physical limits of semiconductor-based microelectronics have become a major design concern. Some effects of these physical limitations can cause significant heat dissipation and data synchronization problems. The demand for more capable microprocessors causes CPU designers to use various methods increasing performance. Some instruction-level parallelism (ILP) methods like superscalar pipelining are suitable for many applications, but are inefficient for others that tend to contain difficult-to-predict code. Many applications are better suited to thread level parallelism (TLP) methods, and multiple independent CPUs is one common method used to increase a system’s overall TLP. A combination of increased available space due to refined manufacturing processes and the demand for increased TLP is the logic behind the creation of multi-core CPUs.

How to increase the performance of multi-core systems?

* Performance of a processor can be increased by increasing clock speed and bus speed.
* To increase the speed of processor we need a large cache memory.
* We need transistors for the performance of a processor.

According to MOORE’s law “The number of transistor that can be integrated on single chip keep increasing exponentially and a processor is consider as better speed by using as many minimum number of transistors.”

**A FUNDAMENTAL THEOREM OF MULTI-CORE PROCESSOR**

“MULTI-CORE PROCESSOR takes advantage of a fundamental relationship between power and frequency”.

By incorporating multiple cores each core is able to run at a lower frequency, dividing among them the power normally given to a single core.

Increasing clock frequency by 20 percent to a single core delivers a 13 percent performance gain, but requires 73 percent greater power. Conversely, decreasing clock frequency by 20 percent reduces power usage by 49 percent, but results in just a 13 percent performance loss.

Here we add a second core on the under clocked example in figure 1.1. This result in a dual core processor that a 20 percent reduced clock frequency effectively delivers 73 percent more performance while using approx. the same power as a single core processor at maximum frequency.

* 1. **Problem Statement**

In the early days of personal computing, personal computers, or PCs, were standalone devices with simple, single user operating systems. Only one program would run at a time. User interaction occurred via simple text based interfaces. Programs followed the standard model of straight line instruction execution proposed by the Neumann architecture. Over time, however the exponential growth in the computing performance quickly led to more sophisticated computing platform. There’s growing demand for more convenient from factors for the home, office, data centre, and on the go. Several technique exist to enhance the computing performance discussed later but couldn’t give the result which market demands. So we have moved to the multi-core architecture.

* 1. **Goal and Objective of Work**

Aim of the thesis work is to study the performances analysis of yhe multi-core computing. Analysing the effect of increasing the number of cores in a single integrated chip with the help of parallel extension of.NET 4.0.

We have tried to provide performance improvement results which will help businesses to justify their investment in either changing their existing or rewriting completely.

**Chapter 2**

**Parallel Computing**

**2.1 Background**

Parallel computing is the Computer Science discipline that deals with the system architecture and software issues related to the concurrent execution of applications. It has been an area of active research interest and application for decades, mainly the focus of high performance computing, but is now emerging as the prevalent computing paradigm due to the semiconductor industry’s shift to multicore processors.

**2.2 Brief History of Parallel Computing**

The interest in parallel computing dates back to the late 1950’s with advancements surfacing in the form of supercomputers throughout the 60’s and 70’s. [15] These were **shared memory** multiprocessors**,** with multiple processors working side by side on shared data. In the mid 1980’s, a new kind of parallel computing was launched when the Caltech Concurrent Computing project built a supercomputer for scientific application from 64 Intel 8086/8087 processors. This system showed that extreme performance could be achieved with mass market,

Off the shelf microprocessors. These massively parallel processors came to dominate the top end of computing, with the ASCI Red supercomputer computer in 1997 breaking the barrier of one trillion floating point operations per second. Since then, MPPs have continued to grow in size and power.

Starting in the late 80’s clusters came to compute and eventually displace M PPs for many application .A cluster is a type of parallel computer built for a large number of off-the shelf computers connected by an off-the-shelf network. Today, cluster are the workhorse of scientific computing and are the dominant architecture in the data centers that power the modern information age.

Today, parallel computing becoming mainstream based on multi-core processors. Most desktop and laptop systems now ship with dual-core microprocesssors, with quad-core processors readily available. Chip manufactures have begun to increase overall processing performance by adding additional CPU cores. The reason is that increasing performance through parallel processing can be far more energy-efficient that increasing microprocessor clock frequencies. In a world which is increasingly mobile and energy conscious, this has become essential. Fortunately, the continued transistor scaling predicated by Moore’s Law will allow for a transition from a few core to many

**2.3 Parallel software**

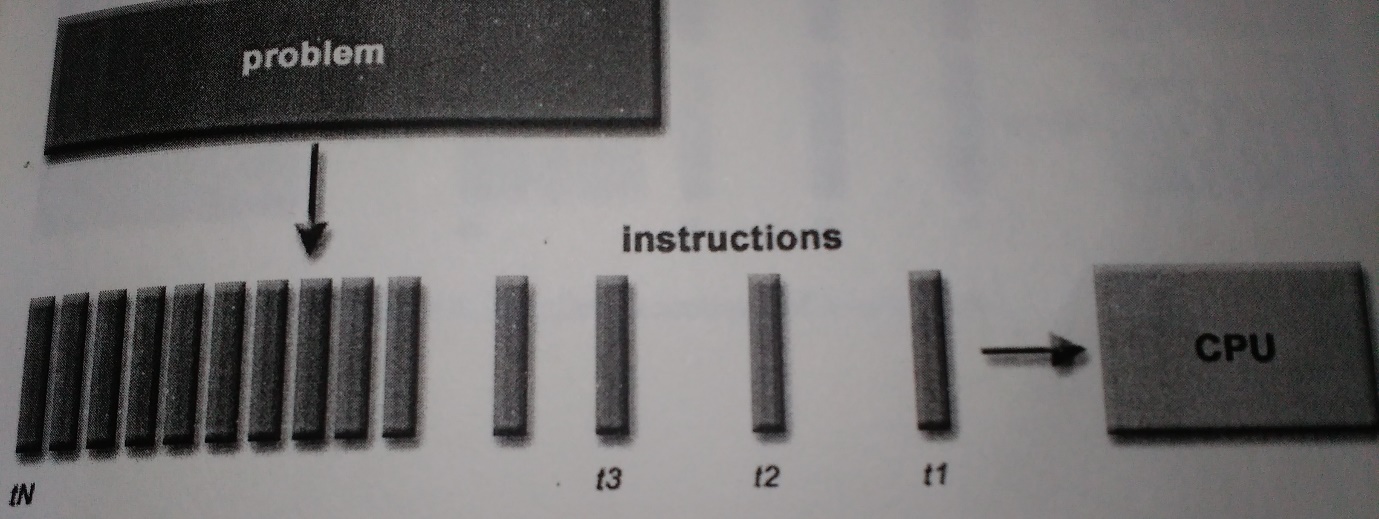
The software world has been very active part of the evolution of parallel computing. Parallel programs have been harder to writer than sequential ones. A program that is divided into multiple concurrent task is more difficult to write due to necessary synchronization and communication that needs to take place between those tasks, some standards have emerged. For MPPs and clusters, a number of application programing interface converged to a single standard called MPI by the mid 1990’s. For shared memory multiprocessor computing. A similar process unfolded with convergence around two standards by the mid to late 1990s. pthread and OpenMp. In addition to these, a multitude of competing parallel programming models and languages have emerged over the years. Some of these models and languages may provide a better solution to the parallel programming problem than the above “standard”, all of which are modifications to conventional, non-parallel language like C.

**2.4 What is Parallel Programming?**

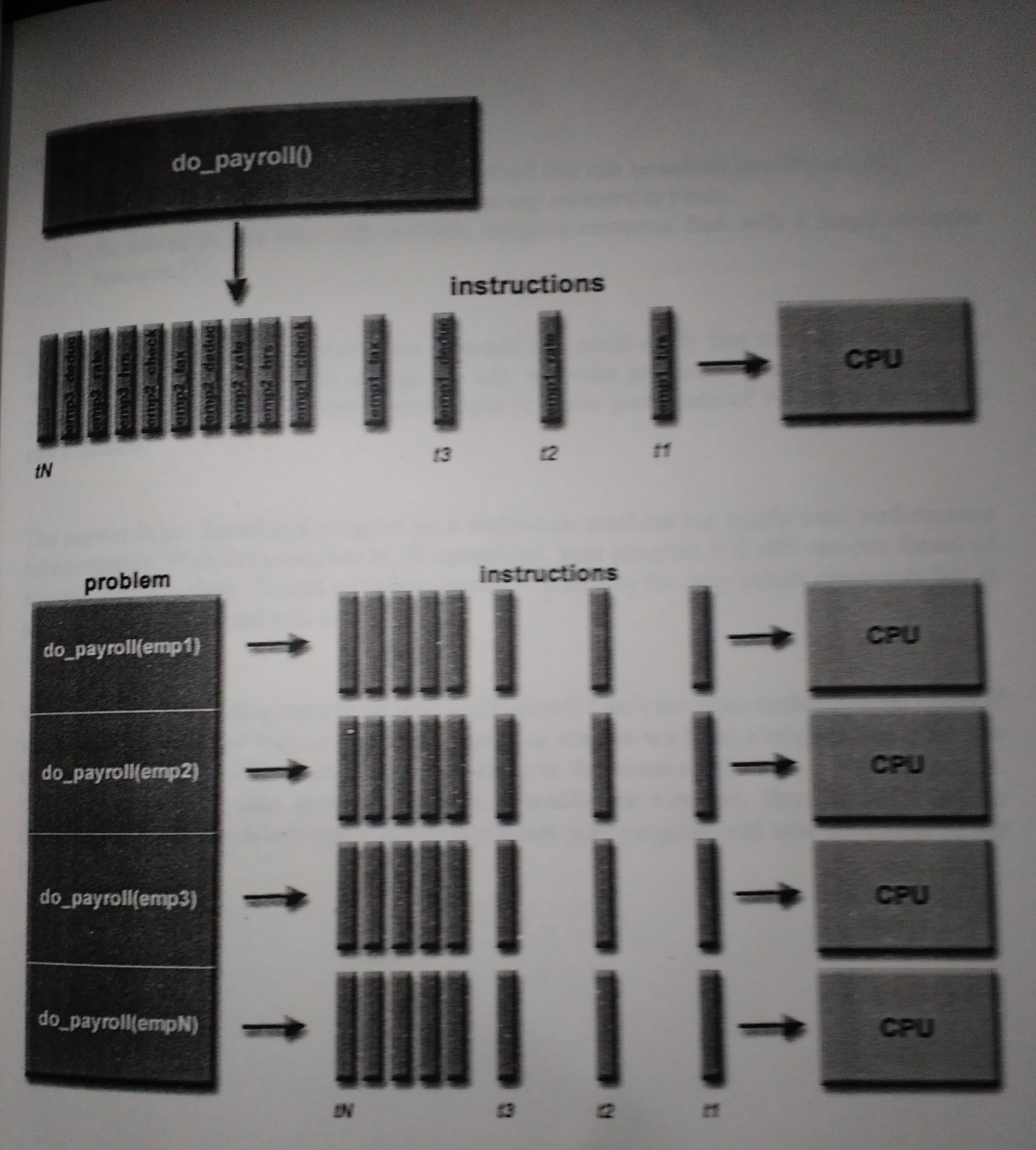
Traditionally, software has been written for serial computation.

1. To be run on a single computer having a single central processing unit.
2. A problem is broken into a discrete series of instructions.
3. Instruction are executed one after another.

Only one instruction may exe cute at any moment in time.



**Figure 3. Sequential execution of instruction.**

****

**Figure 4- Parallel Execution of instructions**

The computational problem should be able to

* Be broken apart into discrete of work that can be solved simultaneously.
* Execute multiple program instructions at any moment in time.
* Be solved in less time with multiple compute resources than with a single compute resources.

Multi-core machines are dominant these days and one could easily find machines with multi-core by default. Do we have to change the way we write programs? Wouldn’t running my problem on a multi-core machine automatically improve performance? After all aren’t there more thread?

The answer is no! Running a program on a multi-core machine has nearly zero performance enhancements. With the exception of IO operation, your program will still use one thread of a single core times only one thread will be active.

What does multithreading have to do with multi-core? Can’t we write multithreaded codes if we have a single core? Yes, we can. But again, as long as we have a single core, only one thread at a time will work since all thread belong to that same single core…logical enough. When your multithreaded program runs on a multi-core machine, threads belonging to different cores will be able to run together and hence your program will take advantage of the hardware power.

**2.5 Multithreading vs. parallelism**

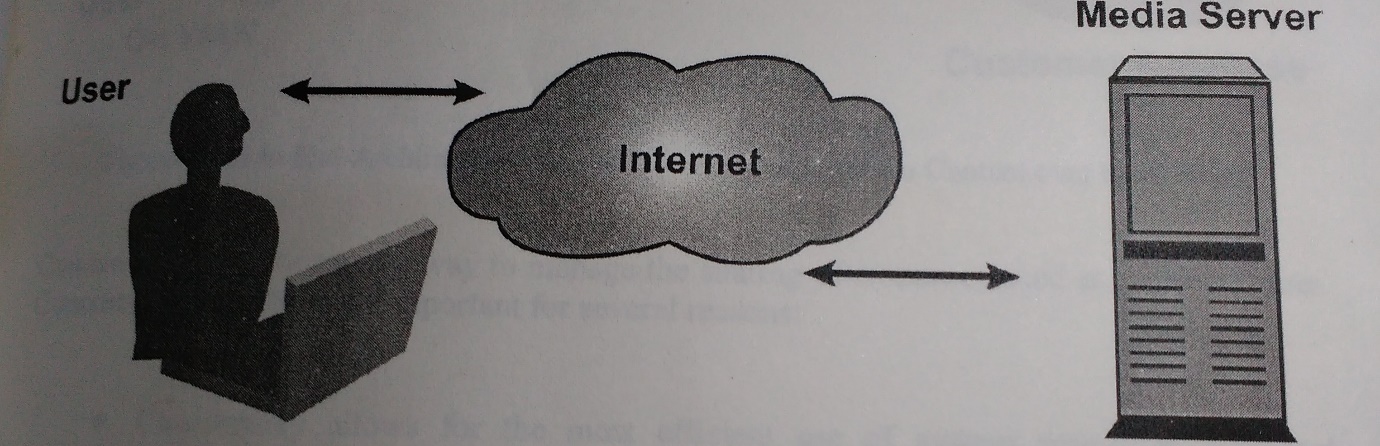
It is useful to point out difference between traditional multithreading and parallel programming. In the past most computer had a single CPU and multithreading was used to take advantage of idle time, such as when a program blocks for user input. Using this approach, one thread can be executed while another is waiting. On a single-CPU system, multithreading is use to allow one or two tasks to share the CPU. Although this type of multithreading will remain useful, it was not designed for situations in which two or more CPU s are available.

When multiple CPUs are present, a second type of multithreading capability is needed because it is possible to execute portions of a program simultaneously, with each part executing on its own CPU. This can be used to significantly speed up the execution of some type of operations, such as sorting, transforming, or searching a large array. [10]

**2.6 Motivation for concurrency in software**

Most end users have a simplistic view of complex computer systems. Consider the following scenario

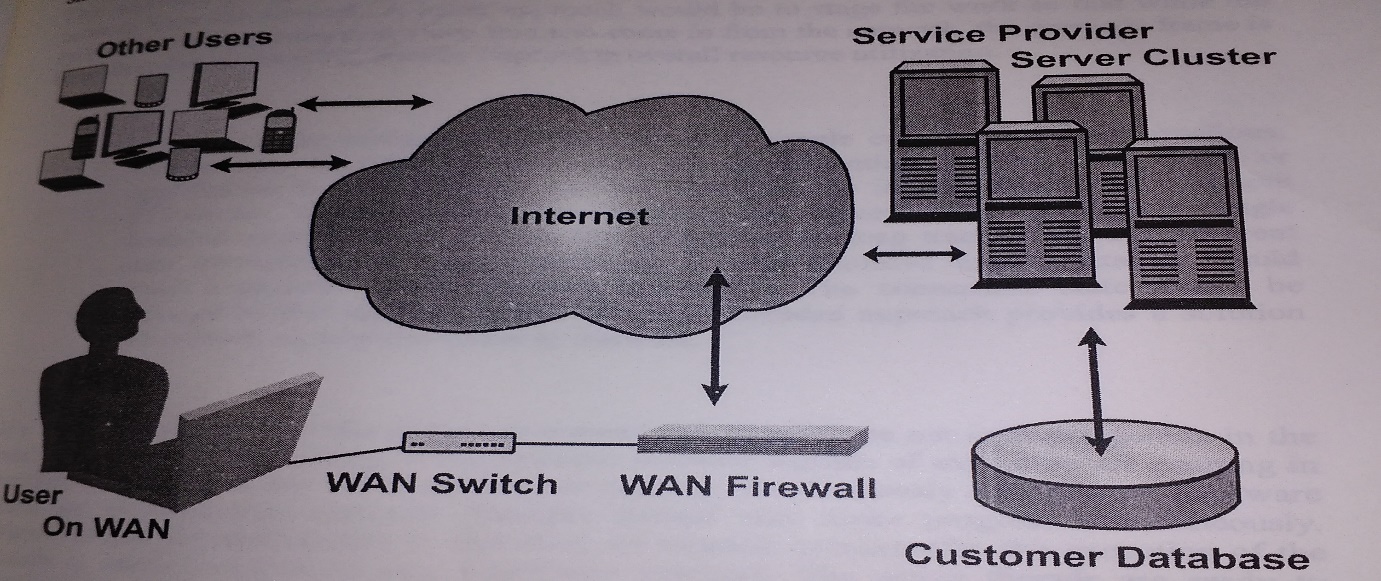
A travelling businessman has to just come back to his hotel after a long day of presentation. Too exhausted to go out, he decided to order room service and stay in his room to watch his favourite baseball team play. Given that he’s on the road, and doesn’t have access to the game on his TV, he decides to check out the broadcast via the Internet. His view of the system might be similar to the one shown in figure.



**Figure 5- End user view of streaming multimedia content via internet**

The user’s expectations are based on conventional broadcast delivery system which provide continuous, uninterrupted delivery of content. The user does not differentiate between streaming the content over the internet and delivering the data via a broadcast network. To the use, watching a baseball game on laptop seems like simple, straightforward task. The reality is that the implementation of such system is far more difficult. From the client side,

The PC must be able to download the streaming video data, compress/decode it, and draw it on the video display, it must handle any streaming audio that accompanies the video stream and send it to soundcard. Meanwhile, given the general purpose nature of computer, the operating system might be configured to run a virus scan or some other task periodically .One the server side, the provider must be able to receive the original broadcast, encode/compress it in near real time and then send it over the network to potentially hundreds of thousands of client. A system designer who is looking to build a computer system capable of stream a Web broadcast might look at the system as it’s shown in figure.



**Figure 6- End to end architecture view of streaming multimedia content over the internet**

Concurrency in software is a way to manage the sharing of resources used at the same time.

Concurrency in software is important for several reasons. Concurrency allows for the most efficient use of system resources .Efficient resource is a key to maximizing performance of computing system. Unnecessarily creating dependencies on different component in the system drastically lowers overall performance. In the aforementioned screening media example, one might naively take this serial, approach the client side.

* Wait for data to arrive on the network.
* Uncompressed the data
* Decode the data
* Send the decoded data to video/audio hardaware

This approach is highly inefficient. The system is completely idle while waiting for data to come in from the network. A better approach would be to stage the work so that while the system is waiting for the next video frame to come in from the network,the previous frame is being decoded by the CPU, thereby improving overall resource utilization.

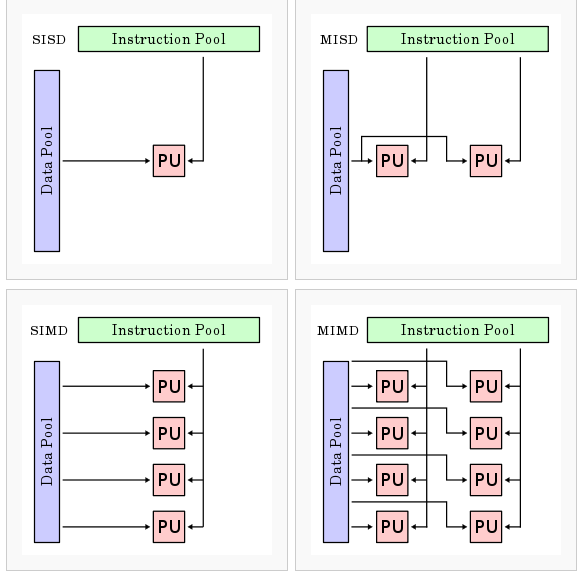
* Many software problems lend themselves to simple concurrent implementations. Concurrent provides an abstraction for implementing software algorithms or applications that are naturally parallel. Consider the implementation of a single FTP server. Multiple client may connect and request different file. A single threaded solution would require the application to keep track of all the different state information for each connection. A more intuitive implementation would create a separate thread for each connection. The connection state would be managed by this separate entity. This multithreaded approach provides a solution that is much simpler and easier to maintain.

It’s worth nothing here that the term concurrent and parallel are not interchangeable in the world of parallel programming. When multiple software threads of execution are running in parallel, it means that active threads are running simultaneously on different hardware resource, or processing elements. Multiple threads may make progress simultaneously. When multiple software threads of execution are running concurrently, the execution of thread is interleaved onto a single hardware resource. The active threads are ready to execute, but only one thread may make progress at a given point in time. In order to have parallelism, you must have concurrency exploiting multiple hardware resources.

**2.7 Parallel computing platforms**

There are two primary forms of data exchange between parallel tasks - accessing a shared data space and exchanging messages. Platforms that provide a shared data space are called shared-address-space machines or multiprocessors. Platforms that support messaging are also called message passing platforms or multicomputer.

Flynn's taxonomy is a classification of computer architectures, proposed by Michael J. Flynn in 1966. The classification system has stuck, and has been used as a tool in design of modern processors and their functionalities. Since the rise of multiprocessing central processing units (CPUs), a multiprogramming context has evolved as an extension of the classification system.



**Fig7-Multi core processor overview architecture**

**Single instruction stream, single data stream (SISD)**

A sequential computer which exploits no parallelism in either the instruction or data streams. Single control unit (CU) fetches single instruction stream (IS) from memory. The CU then generates appropriate control signals to direct single processing element (PE) to operate on single data stream (DS) i.e., one operation at a time.

Examples of SISD architecture are the traditional uniprocessor machines like older personal computers (PCs; by 2010, many PCs had multiple cores) and mainframe computers.

**Single instruction stream, multiple data streams (SIMD)**

A computer which exploits multiple data streams against a single instruction stream to perform operations which may be naturally parallelized. For example, an array processor or graphics processing unit (GPU)

**Multiple instruction streams, single data stream (MISD)**

Multiple instructions operate on one data stream. Uncommon architecture which is generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer.

**Multiple instruction streams, multiple data streams (MIMD)**

Multiple autonomous processors simultaneously executing different instructions on different data. MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space.

**2.8 Parallel Computing in Microprocessor**

The problem is that the memory interfaces stayed the same as parallel execution (threads or cores) were added, thus memory is even more of a bottleneck than before! The typical parallel processing model, spatial decomposition, splits the work between the threads or cores, each of which simultaneously access memory to perform their computation, which exacerbates the problem. (It can also cause interference in shared caches, but others are working on that problem.) How bad is it? Most people turn Hyper Threading off on their Pentium 4 because it provides no benefits for most people. We have examples of memory-intensive parallel programs that scale very well on clusters, but actually run slower on dual processor machines, like the PowerMac G5, dual Opteron’s, and Athlon64 X2 machines.

Our Synchronized Pipelined Parallelism Method (SPPM) restructures programs into producer/consumer chains. The threads of these chains then communicate through shared levels of the memory hierarchy (i.e., cache or front-side buses), which are normally much faster than memory buses.

We can get significant performance improvement, even on on a Pentium 4 with Hyper Threading. The SPPM versions of our programs can run 20-50% faster than the sequential version while maintaining the same number of L2 cache misses! The spatial decomposition versions tend to be much slower and incur many more cache misses. SPPM can even get parallel performance from programs that are not typically parallel, like ARC4 encryption. So SPPM can make your code take advantage of modern parallel microprocessors!

We currently have a detailed journal paper under review, but some preliminary results can be found in this reference:

“The Synchronized Pipelined Parallelism Model (SPPM)” presented at the Parallel and Distributed Computer Systems Symposium in Cambridge, MA in November 2004. (Best paper award)

It turns out that AMD processors are particularly awful at cache coherence performance. So bad that it is faster to fetch data from the memory than from the other core’s cache! On Opteron’s, it is faster to fetch from the remote processor’s memory than from the remote processor’s cache, so it clearly isn’t the Hyper Transport. This made SPPM work badly on these systems, because the consumer was slower than the producer, thus the producer had to wait. Therefore, we invented a new approach to keep the communications between producer and consumer in the cache while reducing the remote cache transfers. This idea, called Polymorphic Threads, works extremely well and is documented in:

In order to understand this surprising behaviour, we developed methods to measure cache to cache communications behaviour. Because this benchmarking tool will be of general interest to the community, we are releasing it into the Open Source community. Please see our page on Source Forge. So if this works, what is left to do? Lots, actually. We have shown that the idea works well, but it is currently hand-coded. We need to make it easier to use, possibly with the use of a synchronized loop class. We also need to develop detailed analytical models so we can predict how a program/machine pair will perform before effort is spent converting the code to SPPM. Ideally, SPPM could be integrated into a compiler, but that is a longer term goal.

**Chapter 3**

**Programming using MATLAB**

**3.1 Introduction**

Millions of engineers and scientists worldwide use MATLAB® to analyze and design the systems and products transforming our world. MATLAB is in automobile active safety systems, interplanetary spacecraft, and health monitoring devices, smart power grids, and LTE cellular networks. It is used for machine learning, signal processing, image processing, computer vision, communications, computational finance, control design, robotics, and much more**.**

**3.2 Parallel Computing Toolbox**

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters. High-level constructs—parallel for-loops, special array types, and parallelized numerical algorithms—let you parallelize MATLAB applications without CUDA or MPI programming. You can use the toolbox with Simulink to run multiple simulations of a model in parallel. The toolbox lets you use the full processing power of multicore desktops by executing applications on workers (MATLAB computational engines) that run locally. Without changing the code, you can run the same applications on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server. You can run parallel applications interactively or in batch.

**Key Features**

* Parallel for-loops (parfor) for running task-parallel algorithms on multiple processors
* Support for CUDA-enabled NVIDIA GPUs
* Full use of multicore processors on the desktop via workers that run locally
* Computer cluster and grid support (with MATLAB Distributed Computing Server)
* Interactive and batch execution of parallel applications
* Distributed arrays and single program multiple data (spmd) construct for large dataset handling and data-parallel algorithms
* Using MATLAB Distributed Computing Server to scale up Parallel Computing Toolbox applications for execution on a cluster.
* Parallel computing with MATLAB. We can use Parallel Computing Toolbox to run applications on a multicore desktop with local workers available in the toolbox, take advantage of GPUs, and scale up to a cluster (with MATLAB Distributed Computing Server).

**3.3 Parallel Programming Applications**

Parallel Computing Toolbox provides several high-level programming constructs that let you convert your applications to take advantage of computers equipped with multicore processors and GPUs. Constructs such as parallel for-loops (parfor) and special array types for distributed processing and for GPU computing simplify parallel code development by abstracting away the complexity of managing computations and data between your MATLAB session and the computing resource you are using.

We can run the same application on a variety of computing resources without reprogramming it. The parallel constructs function in the same way, regardless of the resource on which your application runs—a multicore desktop (using the toolbox) or on a larger resource such as a computer cluster (using toolbox with MATLAB Distributed Computing Server).

**3.4 SPMD**

It executes code in parallel on workers of parallel pool

Syntax

spmd, statements, end

spmd (n), statements, end

spmd (m,n), statements, end

Description

The general form of a spmd (single program, multiple data) statement is:

spmd

statements

end

spmd, statements, end defines a spmd statement on a single line. MATLAB® executes the spmd body denoted by statements on several MATLAB workers simultaneously. The spmd statement can be used only if you have Parallel Computing Toolbox. To execute the statements in parallel, you must first open a pool of MATLAB workers using parpool or have your parallel pretences allow the automatic start of a pool.

Inside the body of the spmd statement, each MATLAB worker has a unique value of labindex, while numlabs denotes the total number of workers executing the block in parallel. Within the body of the spmd statement, communication functions for communicating jobs (such as labSend and labReceive) can transfer data between the workers.

Values returning from the body of a spmd statement are converted to Composite objects on the MATLAB client. A Composite object contains references to the values stored on the remote MATLAB workers, and those values can be retrieved using cell-array indexing. The actual data on the workers remains available on the workers for subsequent spmd execution, so long as the Composite exists on the client and the parallel pool remains open.

By default, MATLAB uses as many workers as it finds available in the pool. When there are no MATLAB workers available, MATLAB executes the block body locally and creates Composite objects as necessary.

spmd(n), statements, end uses n to specify the exact number of MATLAB workers to evaluate statements, provided that n workers are available from the parallel pool. If there are not enough workers available, an error is thrown. If n is zero, MATLAB executes the block body locally and creates Composite objects, the same as if there is no pool available. Spmd (m,n), statements, end uses a minimum of m and a maximum of n workers to evaluate statements. If there are not enough workers available, an error is thrown. M can be zero, which allows the block to run locally if no workers are available.

**Examples**

Perform a simple calculation in parallel, and plot the results:

parpool(3)

spmd

% build magic squares in parallel

q = magic(labindex + 2);

end

for ii=1: length (q)

% plot each magic square

figure, images c (q{ii});

end

delete (gcp)

**3.5 Codistributed**

It creates codistributed array from replicated local data

**Syntax**

C = codistributed(X)

C = codistributed(X,codist)

C = codistributed(X, lab,codist)

C = codistributed (C1, codist)

**Description**

C = codistributed (X) distributes a replicated array X using the default codistributor, creating a codistributed array C as a result. X must be a replicated array, that is, it must have the same value on all workers. Size (C) is the same as size(X). C = codistributed(X, codist) distributes a replicated array X using the distribution scheme defined by codistributor codist. X must be a replicated array, namely it must have the same value on all workers. Size(C) is the same as size(X). For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc. C = Codistributed (X, lab,codist) distributes a local array X that resides on the worker identified by lab, using the codistributor codist. Local array X must be defined on all workers, but only the value from lab is used to construct C. size(C) is the same as size(X). C = Codistributed (C1, codist) accepts an array C1 that is already codistributed, and redistributes it into C according to the distribution scheme defined by the codistributor codist. This is the same as calling C = redistribute (C1, codist). If the existing distribution scheme for C1 is the same as that specified in codist, then the result C is the same as the input C1.

**Examples**

Create a 1000-by-1000 codistributed array C1 using the default distribution scheme.

spmd

N = 1000;

X = magic(N); % Replicated on every worker

C1 = codistributed(X); % Partitioned among the workers

end

Create a 1000-by-1000 codistributed array C2, distributed by rows (over its first dimension).

spmd

N = 1000;

X = magic (N);

C2 = codistributed(X, codistributor1d (1));

End

**Chapter 4**

**Multi Core Processor**

**4.1 What is Multi Core Processor**

A multi-core processor is a single computing component with two or more independent actual processing units (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions (such as add, move data, and branch), but the multiple cores can run multiple instructions at the same time, increasing overall speed for programs amenable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package.

Engineers developed the first processors (CPUs) with only one core. In the mid-1980s Rockwell International manufactured versions of the 6502 with two 6502 cores on one chip as the R65C00, R65C21, and R65C29, sharing the chip's pins on alternate clock-phases. Intel, AMD and others developed other multi-core processors in the early 2000s.

Multi-core processors may have:

* Two cores (dual-core CPUs, for example, AMD Phenom II X2 and Intel Core Duo)
* Three cores (tri-core CPUs, for example, AMD Phenom II X3)
* Four cores (quad-core CPUs, for example, AMD Phenom II X4, Intel's i5 and i7 processors)
* Six cores (hexa-core CPUs, for example, AMD Phenom II X6 and Intel Core i7 Extreme Edition 980X)
* Eight cores (octa-core CPUs, for example, Intel Core i7 5960X Extreme Edition and AMD FX-8350)
* Ten cores (deca-core CPUs, for example, Intel Xeon E7-2850)
* Or more - up to 18 on the Intel Xeon E5 2699v3 or even up to 32 on a future Zen based AMD Opteron APU.

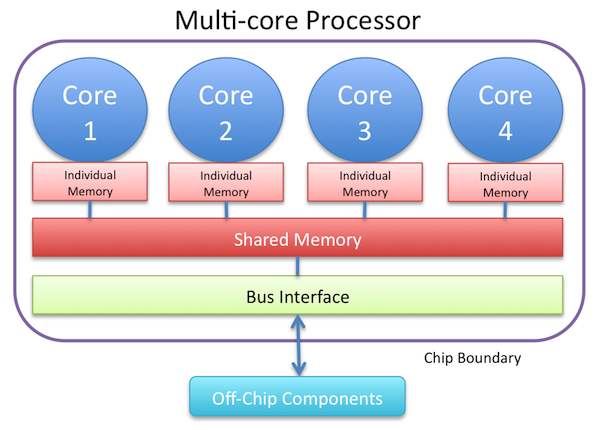
A multi-core processor implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical (e.g. big. LITTLE). Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

Multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU).

The improvement in performance gained by the use of a multi-core processor depends very much on the software algorithms used and their implementation. In particular, possible gains are limited by the fraction of the software that can run in parallel simultaneously on multiple cores; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding use of much slower main-system memory. Most applications, however, are not accelerated so much unless programmers invest a prohibitive amount of effort in re-factoring the whole problem. The parallelization of software is a significant ongoing topic of research.

The trend in processor development has been towards an ever increasing number of cores, as processors with hundreds or even thousands of cores become theoretically possible.In addition, multi-core chips mixed with simultaneous multithreading, memory-on-chip, and special-purpose "heterogeneous" (or asymmetric) cores promise further performance and efficiency gains, especially in processing multimedia, recognition and networking applications. For example, a big. LITTLE core includes a high-performance core (called 'big') and a low-power core (called 'LITTLE'). There is also a trend towards improving energy-efficiency by focusing on performance-per-watt with advanced fine-grain or ultra fine-grain power management and dynamic voltage and frequency scaling (i.e. laptop computers and portable media players).

Chips designed from the outset for a large number of cores (rather than having evolved from single core designs) are sometimes referred to as manycore designs, emphasising qualitative differences.



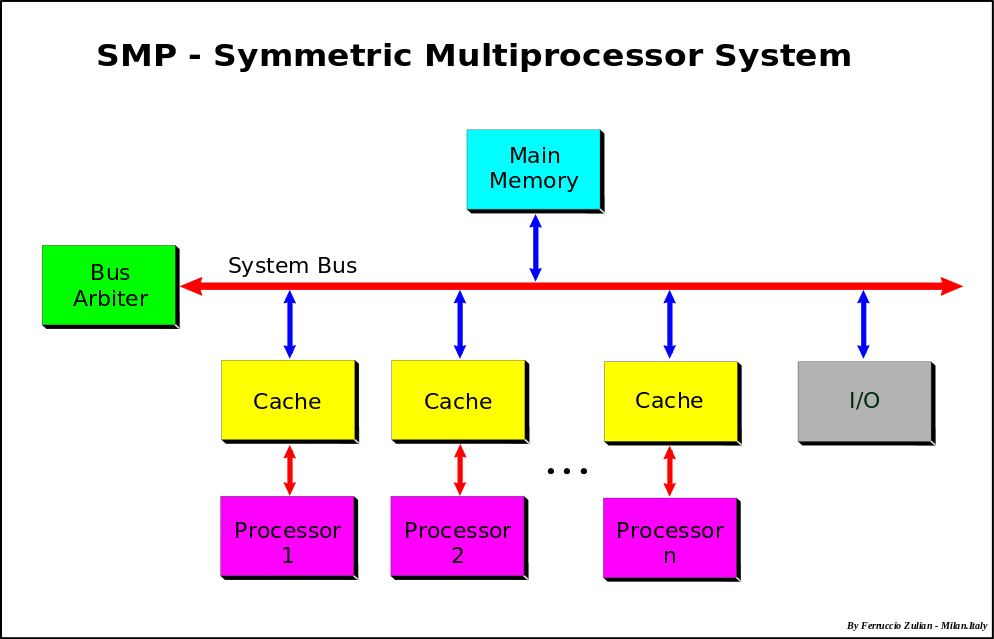
**Fig 8-Logical view of multicore**

**4.2 Symmetric Multi-Processor**

A symmetric multiprocessor system (SMP) is a multiprocessor system with centralized shared memory called main memory (MM) operating under a single operating system with two or more homogeneous processors—i.e., it is not a heterogeneous computing system. More precisely, an SMP is a tightly coupled multiprocessor system with a pool of homogeneous processors running independently, each processor executing different programs and working on different data, with the capability to share resources (memory, I/O device, interrupt system, etc.), and connected using a system bus or a crossbar. Each processor usually has an associated private high-speed memory known as cache memory (or cache) to speed-up the MM data access and to reduce the system bus traffic.

Sometimes the term "symmetric multiprocessor" is confused with the term symmetric multiprocessing. While multiprocessing is a type of processing in which two or more processors work together to process more than one program simultaneously, the term "multiprocessor" refers to the hardware architecture that allows multiprocessing.

The term "symmetric multiprocessor" is used in the majority of the technical papers.



**Fig 9- Symmetric Multiprocessor System**

**4.3 Multithreading on Single core vs Multi-Core Platform**

The multithreading paradigm has become more popular as efforts to further exploit instruction-level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge from the more specialized field of transaction processing; even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multitasking among multiple threads or programs. Thus, techniques that improve the throughput of all tasks result in overall performance gains. Two major techniques for throughput computing are multithreading and multiprocessing.

**4.4 Advantages**

If a thread gets a lot of cache misses, the other threads can continue taking advantage of the unused computing resources, which may lead to faster overall execution as these resources would have been idle if only a single thread were executed. Also, if a thread cannot use all the computing resources of the CPU (because instructions depend on each other's result), running another thread may prevent those resources from becoming idle. If several threads work on the same set of data, they can actually share their cache, leading to better cache usage or synchronization on its values.

A **multi-core processor** implements multiprocessing in a single physical package. Designers may couple cores in a multi-core device tightly or loosely. For example, cores may or may not share caches, and they may implement message passing or shared-memory inter-core communication methods. Common network topologies to interconnect cores include bus, ring, two-dimensional mesh, and crossbar. Homogeneous multi-core systems include only identical cores, heterogeneous multi-core systems have cores that are not identical (e.g. big. LITTLE). Just as with single-processor systems, cores in multi-core systems may implement architectures such as VLIW, superscalar, vector, or multithreading.

**Chapter 5**

**Data, Result and Analysis**

**5.1 Sequential Program Code of Sorting**

function result = Merge\_sort2(input, p, r)

global A;

A = input;

if p < r

q = floor((p+r)/2);

Merge\_sort2(A, p, q);

Merge\_sort2(A, q+1, r);

Merge2(p, q, r);

end

result = A;

end

function Merge2(p, q, r)

global A;

n1 = q - p + 1;

n2 = r - q;

L = [];

R = [];

for i = 1 : n1

L(i) = A(p+i-1);

end

for j = 1 : n2

R(j) = A(q+j);

end

L(n1+1) = inf;

R(n2+1) = inf;

i = 1;

j = 1;

for k = p : r

if L(i) <= R(j)

A(k) = L(i);

i = i + 1;

else

A(k) = R(j);

j = j + 1;

end

end

end

**5.2 Parallel Sort**

clear;

a=randi([1,100000],1,100);

% initialising matlabpool with 4 workers

matlabpool open 4

% initialising spmd tool to distribute jobs over

% all 4 labs for parallel processing

tic;

spmd

% distributing inputs to each lab

p=codistributed(a);

P=getLocalPart(p);

len=length(P);

B=zeros(size(P));

% calling radixSort() method to sort elements

% and storing result from individual lab

%into distributed array B

B=radixSort(P);

end

% taking results from each lab

input1=B{1}; %lab 1

input2=B{2}; %lab 2

input3=B{3}; %lab 3

input4=B{4}; %lab 4

tot=toc;

matlabpool close

%creating a cell with all sorted arrays from each lab

cell={input1,input2,input3,input4};

%merging results from each lab

matlabpool(2)

tic;

spmd

dcell=codistributed(cell);

lcell=getLocalPart(dcell);

i=1;

j=1;

k=1;

while i<=length(lcell{1}) & j<=length(lcell{2})

if(lcell{1}(i)<=lcell{2}(j))

result(k)=lcell{1}(i);

i=i+1;

else

result(k)=lcell{2}(j);

j=j+1;

end

k=k+1;

end

while(i<=length(lcell{1}))

result(k)=lcell{1}(i);

i=i+1;

k=k+1;

end

while(j<=length(lcell{2}))

result(k)=lcell{2}(j);

j=j+1;

k=k+1;

end

%disp(result);

end

%merging(result{1},result{2});

result1=result{1};

result2=result{2};

tot=tot+toc;

matlabpool close

tic;

i=1;

j=1;

k=1;

while i<=length(result1) & j<=length(result2)

if(result1(i)<=result2(j))

fresult(k)=result1(i);

i=i+1;

else

fresult(k)=result2(j);

j=j+1;

end

k=k+1;

end

while(i<=length(result1))

fresult(k)=result1(i);

i=i+1;

k=k+1;

end

while(j<=length(result2))

fresult(k)=result2(j);

j=j+1;

k=k+1;

end

tot=tot+toc;

disp(tot);

%displaying resultant sorted array

%disp(fresult);

% method to implement radix sort

function array = radixSort(array)

maxx = max(array);

base = 1;

while maxx/base > 0

array = counting\_sort(array,base);

base = base \* 10;

end

% method implenting counting

function W = counting\_sort(array,base)

X = zeros(1,11);

W = zeros(1,numel(array));

% Store count of occurrences in X()

for j = 1:numel(array)

X(rem(floor(array(j)/base),10)+1) = X(rem(floor(array(j)/base),10)+1) + 1;

end

% Change X(i) so that X(i) now contains actual

% position of this digit in output array W()

for i = 2:11

X(i) = X(i) + X(i-1);

end

% Building output array W()

for j = numel(array):-1:1

W(X(rem(floor(array(j)/base),10)+1)) = array(j);

X(rem(floor(array(j)/base),10)+1) = X(rem(floor(array(j)/base),10)+1) - 1;

end

end

end

**5.3 Test data time in seconds**

|  |  |  |
| --- | --- | --- |
| No. of Input | Sequential Sort | Parallel Sort |
| 100 | 0.0088 | 0.9509 |
| 500 | 0.1066 | 0.9793 |
| 5000 | 0.9002 | 1.6671 |
| 10000 | 1.9880 | 2.4198 |
| 100000 | 28.9187 | 5.3444 |
| 160000 | 134.5690 | 15.3758 |
| 200000 | 211.5329 | 17.5682 |

Table 5.3.1-Comparative study for sequential sort vs parallel sort on 2 core processor

|  |  |  |
| --- | --- | --- |
| No. of Input | Sequential Sort | Parallel Sort |
| 100 | 0.0088 | 1.803 |
| 500 | 0.1066 | 2.4803 |
| 5000 | 0.9002 | 3.0527 |
| 10000 | 1.9880 | 3.8078 |
| 100000 | 28.9187 | 20.0426 |
| 160000 | 134.5690 | 56.2867 |
| 200000 | 211.5329 | 94.2005 |

Table 5.3.2-Comparative study for sequential sort vs parallel sort on 4 core processor

**5.4 Graph based on Test Data**

**5.5 Results**

* Parallel programming provided great performance improvement (30% to 80%). Time taken in few cases was less than one-fourth. Percentage will increase many times as number of inputs increase.
* Test timing were affected if work load on CPU increased like multiple application being opened, automatic updates being run, wireless router lookup for available networks or any other background process run by any system applications.
* Based on the results and analysis we can conclude
  + Multi core processors are the future of computing.
  + Multi core processors represent an important new trend in computer architecture.

Decreased power consumption and heat generation.

Minimised wire lengths and inter connect latencies.

* To utilize their full potential, applications will need to move from a single to multi-processor model.
* The software industry needs to get back into the state where existing applications run faster on new hardware.

**5.6 Challenges of Multi Core Computing**

* Need for Parallel Computing and parallel programming
* Requires mechanism for efficient inter-process coordination.
* Power and temperature management
* Memory/ Cache coherence
* Ability of multi core processor to increase application performance depends on the use of multiple threads within applications.
* If a single core is closed to being memory bandwidth limited, going to dual core might only give 30 to 70 % improvement.
* If memory bandwidth is not a problem, a 90% improvement can be expected.
* Most current video games will run faster on 3 GHz single core processor than on a 2 GHz dual core processor.

**5.7 Benefits of Multi Core Computing**

* Massive performance improvement if application design uses parallel programming architecture or multithreading. Design might be a constraint for current business application.
* Signals between different CPUs travel shorter distances
* These high quality signals allow more data to be sent in a given time period
* A dual-core processor uses slightly less power than two coupled single core processor.
* Better user experience and system responsiveness while working on multiple applications at the same time on the computer.

**5.8 Future scope**

As multicore processors bring parallel computing to mainstream customers, the key challenge in computing today is to transition the software industry to parallel programming. The long history of parallel software has not revealed any “silver bullets” and indicates that there will not be likely be any single technology that will make parallel software ubiquitous. With the increasingly ubiquitous and pervasive nature of computers in the modern society, the class of problems and applications computing science has to address is continuously expanding.

The importance played by parallelism in each of these two major development trends confirms the fundamental role parallel processing continues to occupy in the theory of computing. The idea of massive parallelism permeates virtually all unconventional models of computation proposed to date and this is shown here through examples such as DNA computing, quantum computing or reaction–diffusion computers. Even a model that is mainly of theoretical interest, like the accelerating machine, can be thought of as deriving its power from doubling the number of processing units (operating in parallel) at each step.

The scope of computing science has expanded enormously from its modest boundaries formulated at the inception of the field and many of the unconventional problems we encounter today in this area are inherently parallel.We illustrate this by presenting five examples of tasks in quantum information processing that can only be carried out successfully through a parallel approach. It is one more testimony to the fact that parallelism is universally applicable and that the future of computing cannot be conceived without parallel processing.

**CONCLUSION**

We have clearly seen here that time for sequential sort first was less than parallel sort when input was less but as input size increases the time for parallel sort gradually decreases. Hence we can say that parallel programming is the future of modern computing when it comes to deal with huge amount of datasets.

**References and Bibliography**

* 1. <http://in.mathworks.com/products/matlab/>
  2. <http://in.mathworks.com/products/parallel-computing/>
  3. G. Păun, G. Rozenberg, A. Salomaa, DNA Computing – New Computing Paradigms, Springer (1998).
  4. L. Adleman, Molecular computation of solutions to combinatorial problems, Science 266 (1994) 1021–1024.CrossRef
  5. R. J. Lipton, DNA solution of hard computational problems, Science 268 (5210) (1995) 542– 545.CrossRef
  6. W.-L. Chang, M. Guo, J. Wu, Solving the independent-set problem in a DNA-based supercomputer model, Parallel Processing Letters 15 (4) (2005) 469–479.CrossRefMathSciNet
  7. G. Păun, Computing with membranes, Journal of Computer and System Sciences 61 (1) (2000) 108–143.MATHCrossRefMathSciNet
  8. M. Pérez-Jiménez, A. Riscos-Núñez, A linear solution for the knapsack problem using active membranes, in: Membrane Computing. Lecture Notes in Computer Science, Vol. 2933, Springer (2004) pp. 250–268.
  9. G. Păun, P systems with active membranes: Attacking NP-complete problems, Journal of Automata, Languages, Combinatorics 6 (1) (2001) 5–90.
  10. C. Zandron, C. Ferretti, G. Mauri, Solving NP-complete problems using P systems with active membranes, in: I. Antoniou, C. Calude, M. Dinneen (Eds.), Unconventional Models of Computation, Springer, London (2000) pp. 289–301, dISCO – Universita di Milano-Bicocca, Italy.
  11. P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, Special issue on Quantum Computation of the SIAM Journal on Computing 26 (5) (1997) 1484–1509.MATHMathSciNet
  12. L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, I. L. Chuang, Experimental realization of Shor’s quantum factoring algorithm using nuclear magnetic resonance, Nature 414 (2001) 829–938.CrossRef