Parallel Computation Summary

Moatasem Gamal

FCAI BSU

January 11, 2024

Contents

Chapter 1

Multithreaded Programming

1.1 Introduction

- Modern operating systems hold more than one activity (program) in memory and the processor can switch among all to execute them.
- **Multitasking/MultiProcesses:** is the simultaneous occurrence of several activities (program) on a computer.
- Actually to have true multitasking, the applications run on a machine with multiple processors.
- Multitasking results in effective and simultaneous utilization of various system resources such as processors, disks, and printers.

1.2 Parallel Programming vs Sequential Programming

- In parallel programming, multiple tasks are executed simultaneously, allowing for better performance and maximum utilization of system resources such as processors, disks, and printers.
- Sequential Programming means that process are executed sequentially, one after another. When running a sequential Java program, commands are executed linearly, where each process must complete before the next one starts.

1.3 The operating system multitasking

The operating system supports multitasking in a **cooperative** or **preemptive** manner.

1.3.1 Cooperative manner

In cooperative multitasking each application is responsible for relinquishing control to the processor to enable it to execute the other application, as in earlier versions of operating systems.

1.3.2 Preemptive manner

In the preemptive type multitasking, the processor is responsible for executing each application in a certain amount of time called a time slice, as in modern operating systems.

1.3.3 Cooperative VS Preemptive

table (??) shows the main differences between the two manners

Manner	responsibility	used in
Cooperative	**application** → relinquishing control to the **processor** to enable it to execute the other application	earlier versions of operating systems
Preemptive	**processor** → executing each application in a certain amount of time called a **time slice**	modern operating systems

Table 1.1: Cooperative VS Preemptive

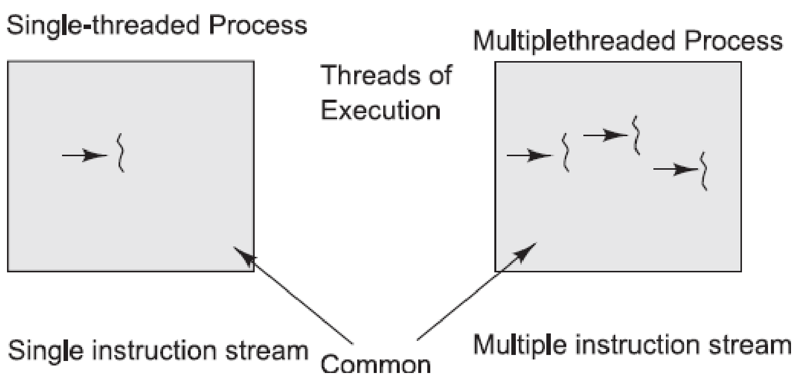
Note: A single processor computer is shared among multiple applications with preemptive multitasking, the processor is switching between the applications at intervals of milliseconds, you feel that all applications run concurrently.

1.4 Concepts: Process and Thread

	Process	Thread
Definition	• A process is a program in execution • A process is sometime referred as task • A process is a collection of one or more threads and associated system resources. • A process may be divided into a number of independent units known as threads • A process may have a number of threads in it.	• Threads are light-weight processes within a process • A thread is a dispatchable unit of work • A thread may be assumed as a subset of a process. • A thread is a smallest part of the process that can execute concurrently with other parts(threads) of the process

Multitasking	• Multitasking of two or more processes is known as **process-based multitasking** • Process-based multitasking is totally controlled by the **operating system**	• Multitasking of two or more threads is known as **thread-based multitasking** • The concept of multithreading in a programming language refers to thread-based multitasking • thread-based multitasking can be controlled by the **programmer** to some extent in a program
Address Space	A process has its own address space	A thread uses the process's address space and share it with the other threads of that process
Communication	A process can communicate with other process by using inter-process communication	• A thread can communicate with other thread (of the same process) directly by using methods like wait(), notify(), notifyAll(). • All threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables
New Creation	the creation of new processes require duplication of the parent process	New threads are easily created
Control	A process does not have control over the sibling process, it has control over its child processes only	Threads have control over the other threads of the same process
Construct	Processes are an architectural construct	Thread is a coding construct that does not affect the architecture of an application

1.4.1 Process containing single and multiple threads



1.4.2 The advantages of thread-based multitasking as compared to process-based multitasking :

- Threads share the same address space.
- Context-switching between threads is normally inexpensive.
- Communication between threads is normally inexpensive.
- Java supports thread-based multitasking.

1.5 Context Switching

- The concept of context switching is integral to threading.
- A hardware timer is used by the processor to determine the end of the time-slice for each thread.
- The timer signals at the end of the timeslice and in turn the processor saves all information required for the current thread onto a **stack**. Then the processor moves this information from the stack into a predefined data structure called a **context structure**.
- When the processor wants to switch back to a previously executing thread, it transfers all the information from the context structure associated with the thread to the stack.

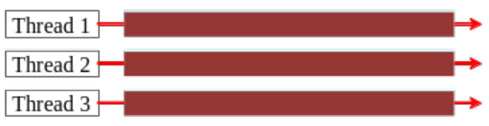
Chapter 2

THREADS IN JAVA

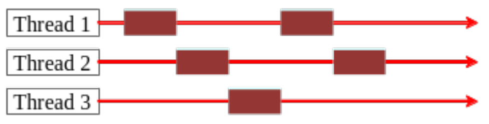
We focus on learning how to write an application containing multiple tasks that can be executed concurrently. In Java, this is realized by using multithreading techniques.

2.1 Threads

- All threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables.
- A single process might contain multiple threads.
- Java supports thread-based multitasking
- Threads are lightweight processes as the overhead of switching between threads is less
- They can be easily spawned
- The Java Virtual Machine (JVM) spawns a thread when your program is run called the Main Thread
- Multiple Threads on single CPU or multiple CPUs:

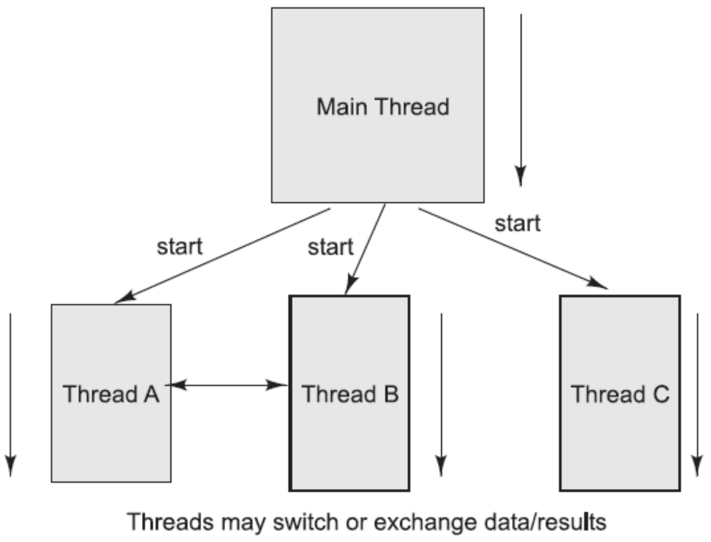


Multiple Threads on multiple CPUs



Multiple threads sharing a single CPU

- Program with master and children threads

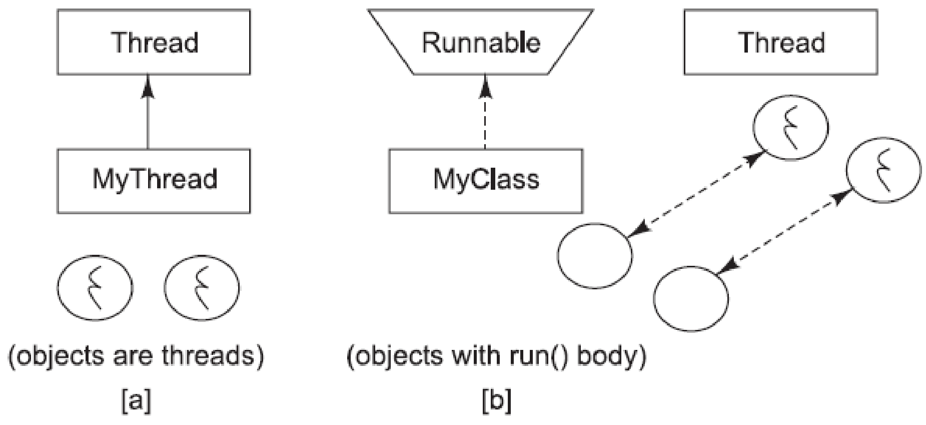


2.1.1 Why do we need threads?

- To enhance parallel processing
- To increase response to the user
- To utilize the idle time of the CPU
- Prioritize your work depending on priority

2.2 Implementing Threads in Java

- Threads are objects in the Java language. They can be created by using two different mechanisms:
  1. Create a class that **extends** the standard **Thread** class.
  2. Create a class that **implements** the standard **Runnable** interface



- Thread can be defined by:

  - Extending the java.lang.Thread class, or
  - Implementing the java.lang.Runnable interface.

- The run() method should be overridden and should contain the code that will be executed by the new thread. This method must be public with a void return type and should not take any arguments.
- run() method is the starting point for thread execution

2.2.1 Extending the Thread Class

1. Create a class by extending the Thread class and override the run() method:

```
class MyThread extends Thread {
    public void run() {
            // thread body of execution
    }
}
```

2. Create a thread object:

```
MyThread thr1 = new MyThread();
```

3. Start Execution of created thread:

```
thr1.start();
```

Example

```
/* ThreadEx1.java: A simple program creating and invoking a thread object
by extending the standard Thread class. */
class MyThread extends Thread {
    public void run() {
        System.out.println(" this thread is running ... ");
    }
}
class ThreadEx1 {
    public static void main(String [] args ) {
        MyThread t = new MyThread();
        t.start();
    }
}
```

2.2.2 Implementing the Runnable Interface

It is more preferred to implement the Runnable Interface so that we can extend properties from other classes

1. Create a class that implements the interface Runnable and override run() method:

```
class MyThread implements Runnable {
    ...
    public void run() {
        // thread body of execution
    }
}
```

2. Creating Object:

```
        MyThread myObject = new MyThread();
```

3. Creating Thread Object:

```
        Thread thr1 = new Thread(myObject);
```

4. Start Execution:

```
        thr1.start();
```

Example

```
/* ThreadEx2.java: A simple program creating and invoking a thread object
implementing Runnable interface. */
class MyThread implements Runnable {
    public void run() {
        System.out.println("␣this␣thread␣is␣running␣...␣");
    }
}
class ThreadEx2 {
    public static void main(String [] args ) {
        Thread t = new Thread(new MyThread());
        t.start();
    }
}
```

2.3 Life cycle of threads & Thread states

2.3.1 Life Cycle of Thread

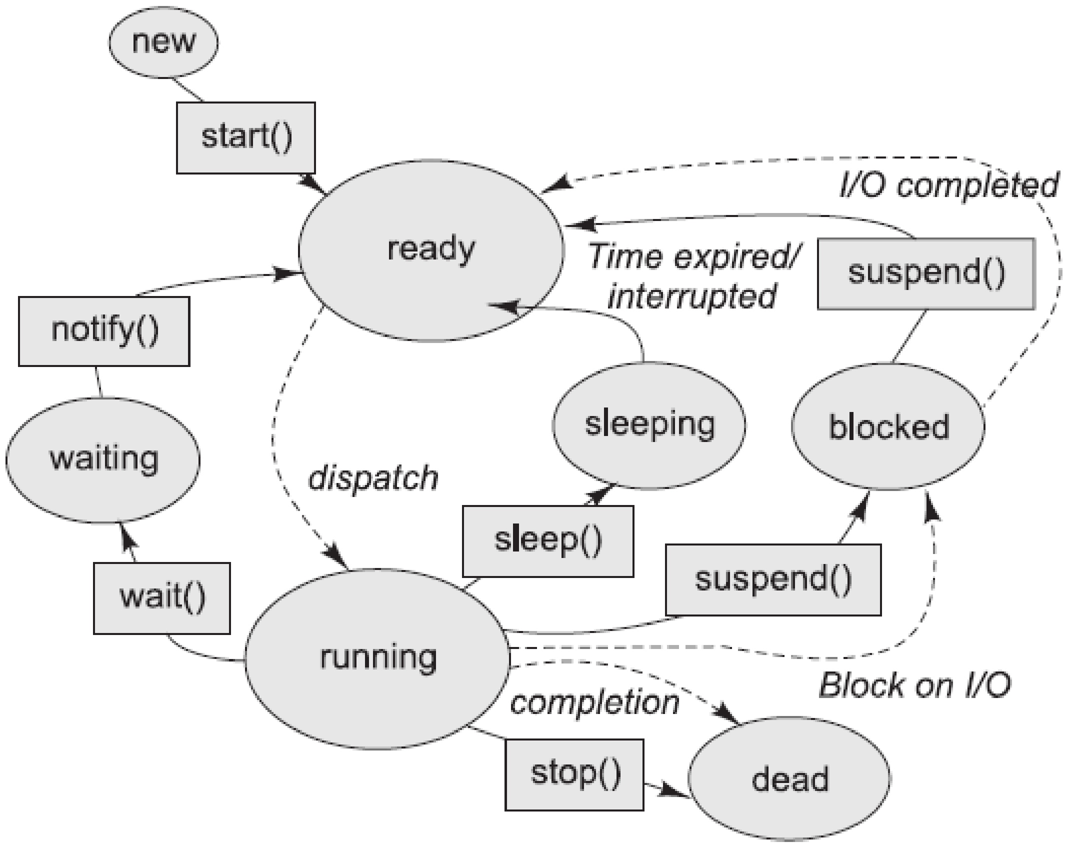
2.3.2 Thread States

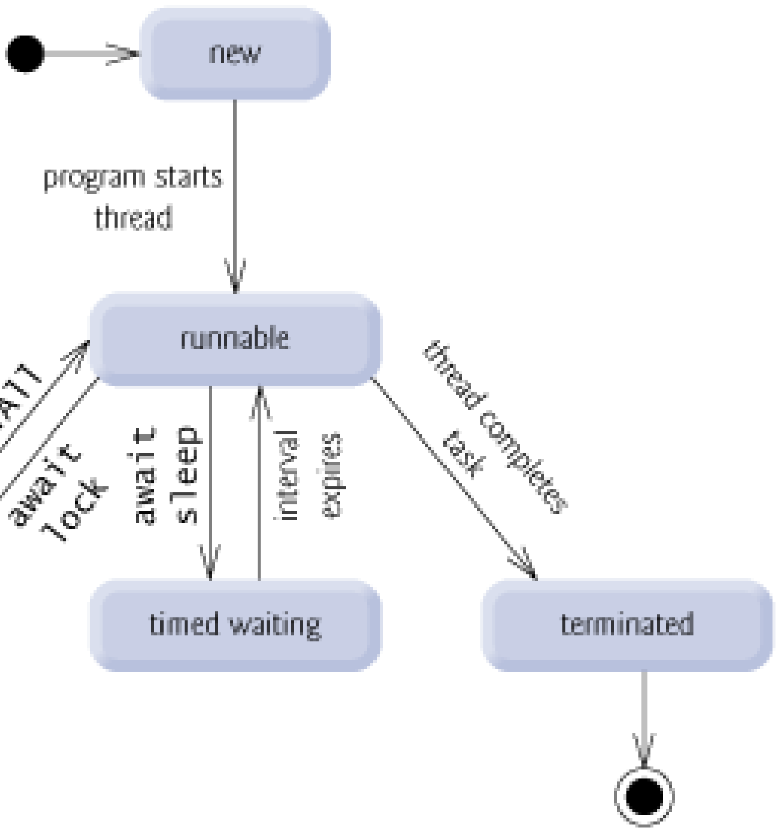
A thread can be in one of five states: New, Ready, Running, Blocked, or Finished showed in figure(??).

2.3.3 Thread termination

A thread becomes Not Runnable when one of these events occurs:

- Its sleep method is invoked.





- The thread calls the wait method to wait for a specific condition to be satisfied.

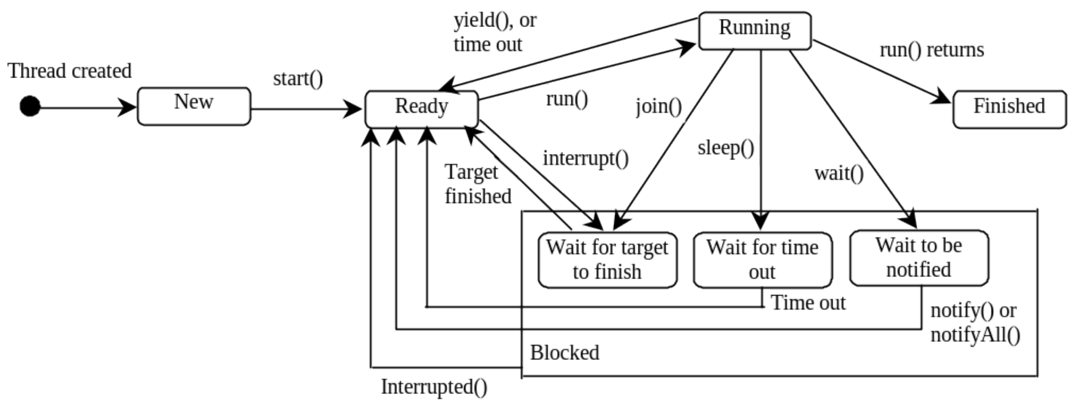


Figure 2.1: Thread states

- The thread is blocking on I/O.

2.4 Good example

- Consider a simple web server
- The web server listens for request and serves it
- If the web server was not multithreaded, the requests processing would be in a queue, thus increasing the response time and also might hang the server if there was a bad request.
- By implementing in a multithreaded environment, the web server can serve multiple request simultaneously thus improving response time

2.5 Running threads

```
class mythread implements Runnable{
    public void run(){
        System.out.println("Thread␣Started");
    }
}

class mainclass {
    public static void main(String args[]){
        Thread  t = new Thread(new mythread()); // This is the way to insta
thread implementing runnable interface
        t.start(); // starts the thread by running the run method
    }
}
```

- Calling t.run() does not start a thread, it is just a simple method call.
- Creating an object does not create a thread, calling start() method creates the thread.

2.6 Synchronization

- Synchronization is prevent data corruption
- Synchronization allows only one thread to perform an operation on a object at a time.
- If multiple threads require an access to an object, synchronization helps in maintaining consistency.

2.6.1 Example on Synchronization

```
public class Counter{
    private int count = 0;
    public int getCount(){
        return count;
    }

    public setCount(int count){
        this.count = count;
    }
}
```

- In this example, the counter tells how many an access has been made.
- If a thread is accessing setCount and updating count and another thread is accessing getCount at the same time, there will be inconsistency in the value of count.

Fixing the example

```
public class Counter{
        private static int count = 0;
        public synchronized int getCount(){
                return count;
        }

        public synchoronized setCount(int count){
        this.count = count;
        }
}
```

- By adding the synchronized keyword we make sure that when one thread is in the setCount method the other threads are all in waiting state.

2.6.2 What about static methods?

```
public class Counter{
        private int count = 0;
        public static synchronized int getCount(){
            return count;
        }

        public static synchronized setCount(int count){
            this.count = count;
        }
}
```

- In this example the methods are static and hence are associated with the class and not the instance.

- Hence the lock is placed on the class object that is, Counter.class object and not on the object itself. Any other non static synchronized methods are still available for access by other threads.

2.6.3 Common Synchronization mistake

```
public class Counter{
        private int count = 0;
        public static synchronized int getCount(){
                return count;
        }

        public synchronized setCount(int count){
                this.count = count;
        }
}
```

- The common mistake here is one method is static synchronized and another method is non static synchronized.

- This makes a difference as locks are placed on two different objects. The class object and the instance and hence two different threads can access the methods simultaneously.

2.6.4 Synchronization vs Static Synchronization

Feature	**Synchronization**	**Static Synchronization**
Scope	Object-level	Class-level
Lock Acquisition	Acquires lock on the object instance	Acquires lock on the class itself
Impact on Multiple Objects	Each object's synchronized methods can be accessed by different threads simultaneously.	Only one thread can access any static synchronized method of the class at a time.
Usage	Used to protect shared resources at the object level	Used to protect shared resources at the class level or for operations that involve the class itself
Declaration	Applied to instance methods using the synchronized keyword	Applied to static methods using the static synchronized keywords
Example	public synchronized void deposit() ...	public static synchronized void getInstance() ...

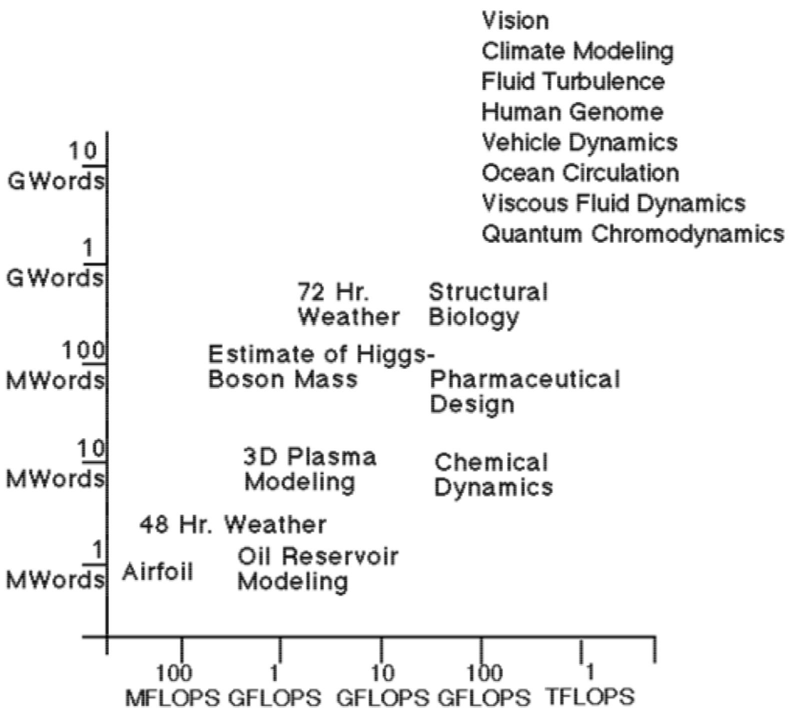
Chapter 3

Parallel Computing: Overview (PSC)

by John Urbanicurbanic@psc.edu

3.1 Why we need parallel computing?

for **New Applications**
The graph shows that applications that require processing large amounts of data are the ones that benefit most from parallel computing.

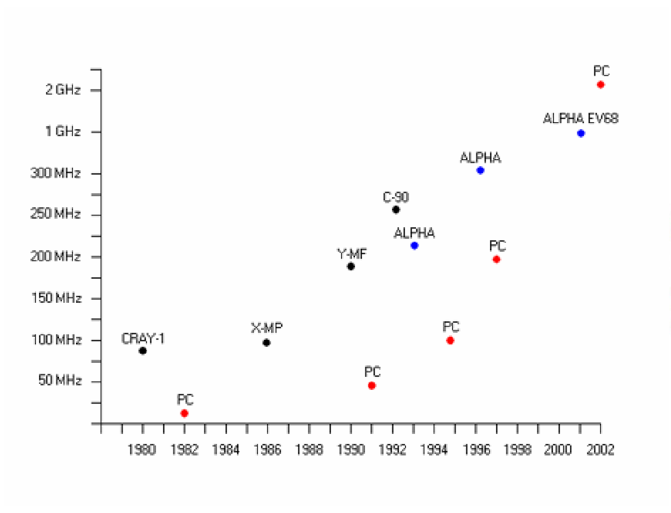


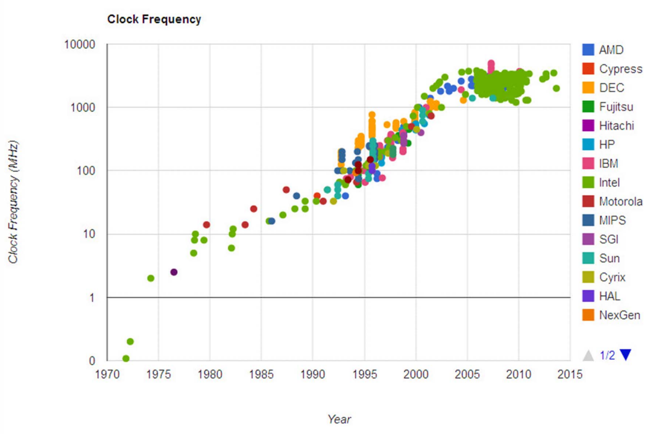
- The performance capabilities of supercomputers are expressed using a standard rate for indicating the number of floating-point arithmetic calculations systems can perform on a per-second basis. The rate, **floating-point operations per second**, is abbreviated as **FLOPS**.

- The per-second rate ”FLOPS” is commonly misinterpreted as the plural form of ”FLOP” (short for ”floating-point operation”) [1]

3.2 Clock Speed

3.2.1 clock speed over previous years





3.2.2 Y-MP vs C90 supercomputers

When the PSC [1] went from a 2.7 GFlop Y-MP to a 16 GFlop C90, the clock only got 50faster. The rest of the speed increase was due to increased use of parallel techniques:

- More processors (8 $\rightarrow$ 16)
- Longer vector pipes (64 $\rightarrow$ 128)
- Parallel functional units (2)

–	Y-MP	C90
processors	8	16
vector pipes	64	128
Parallel functional units	2	2

So, we want as many processors working together as possible. How do we do this? There are two distinct elements:

- Hardware: vendor does this
- Software: you, at least today

[1]PSC Pittsburgh Supercomputing Center, The Pittsburgh Supercomputing Center (PSC) is a high performance computing and networking center founded in 1986 and one of the original five NSF Supercomputing Centers.

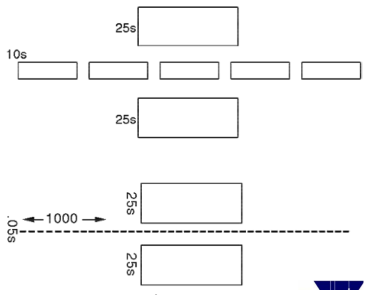
3.3 Amdahl's Law

How many processors can we really use?

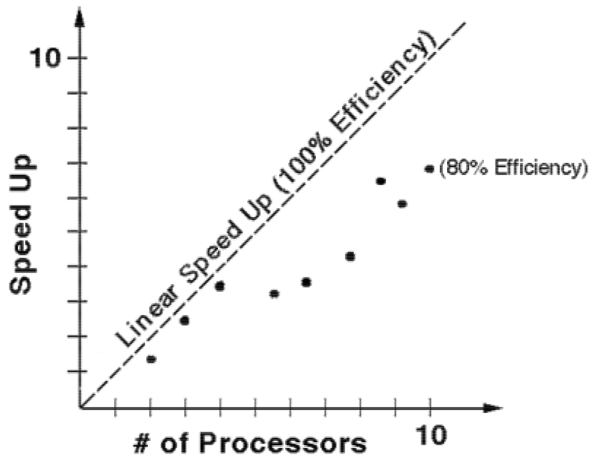
25s	Serial
50s	Parallel
25s	Serial

Let's say we have a legacy code such that is it only feasible to convert half of the heavily used routines to parallel:

- If we run this on a parallel machine with five processors: Our code now takes about 60s. We have sped it up by about 40%.
- Let's say we use a thousand processors: We have now sped our code by about a factor of two.



This seems pretty depressing, and it does point out one limitation of converting old codes one subroutine at a time. However, most new codes, and almost all parallel algorithms, can be written almost entirely in parallel (usually, the "start up" or initial input I/O code is the exception), resulting in significant practical speed ups. This can be quantified by how well a code scales which is often measured as efficiency.



3.3.1 Amdahl's Law equation

Time

if single-processor finishes one program in one unit of time, how much time will multiple-processors require to finish that task?

$$Time_{for\ 1\ processor} = 1 \tag{3.1}$$

$$Time_{for\ 2\ processor} = \frac{1}{2} \tag{3.2}$$

$$Time_{for\ n\ processors} = \frac{1}{n} \tag{3.3}$$

Speedup

If you are using n processors, your $Speedup_n$ is:

$$Speedup_n = \frac{T_1}{T_n} \tag{3.4}$$

And your Speedup $\mathrm{Efficiency_n}$ is:

$$\mathrm{Efficiency_n} = \frac{Speedup_n}{n} \tag{3.5}$$

which could be as high as 1., but probably never will be.

Amdahl's law

If you put in n processors, you should get n times Speedup (and 100% Speedup Efficiency), right? Wrong! There are always some fraction of the total operation that is inherently sequential and cannot be parallelized no matter what you do. This includes reading data, setting up calculations, control logic, storing results, etc.

If you think of all the operations that a program needs to do as being divided between a fraction that is parallelizable and a fraction that isn't (i.e., is stuck at being sequential), then Amdahl's Law says:

$$Speedup_n = \frac{T_1}{T_n} = \frac{1}{\frac{F_{parallel}}{n} + F_{sequential}} = \frac{1}{\frac{F_{parallel}}{n} + (1 - F_{parallel})} \tag{3.6}$$

Maximum Possible SpeedUp

$$max\ Speedup = \frac{1}{1 - F_{parallel}} \tag{3.7}$$

Example 1

5% of a parallel programs's execution time is spent within inherently sequential code. Calculate The maximum speedup achievable by this program, regardless of how many PEs are used
Solution

$$Speedup_\infty = \frac{1}{\frac{0.95}{\infty} + 0.05} = 20 \tag{3.8}$$

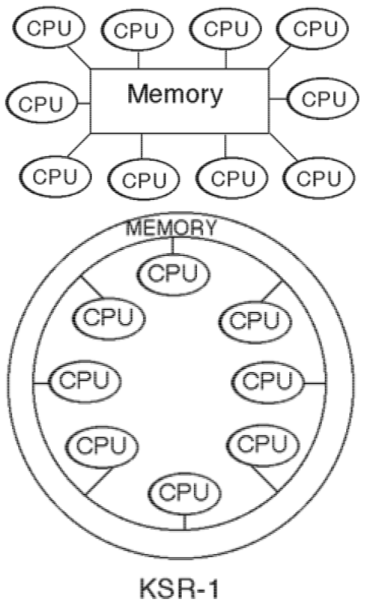
Example 2

95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?
Solution

$$Speedup_8 = \frac{1}{\frac{0.95}{8} + 0.05} \approx 5.9 \tag{3.9}$$

3.4 Shared Memory and Distributed Memory

3.4.1 Shared Memory



Easiest to program. There are no real data distribution or communication issues. Why doesn't everyone use this scheme?

- Limited numbers of processors (tens) - Only so many processors can share the same bus before conflicts dominate.
- Limited memory size - Memory shares bus as well. Accessing one part of memory will interfere with access to other parts.

Table 3.1: Comparison of Shared and Distributed Memory Architectures

Feature	**Shared Memory**	**Distributed Memory**
Ease of Programming	Easier	More difficult
Data Distribution	Not required	Crucial
Communication Issues	Minimal	Significant
Number of Processors	Limited (tens)	Limited by physical size (tens of meters)
Memory Size	Limited	Very large
Local Disk per Node	Often available	Usually not available
Virtual Memory	Supported	Not typically supported
Data Access Times	Uniform	Dependent on data location (local vs. remote)

3.5

3.5.1 Common Distributed Memory Machines

3.6 Latency and Bandwidth

3.7 Data Parallel

3.8 Work Sharing

3.9 Load Balancing

Bibliography

[1] University Information Technology Services. Understand measures of supercomputer performance and storage system capacity. https://kb.iu.edu/d/apeq, 2023.