Parallel Computation Summary

Moatasem Gamal

FCAI BSU

January 13, 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

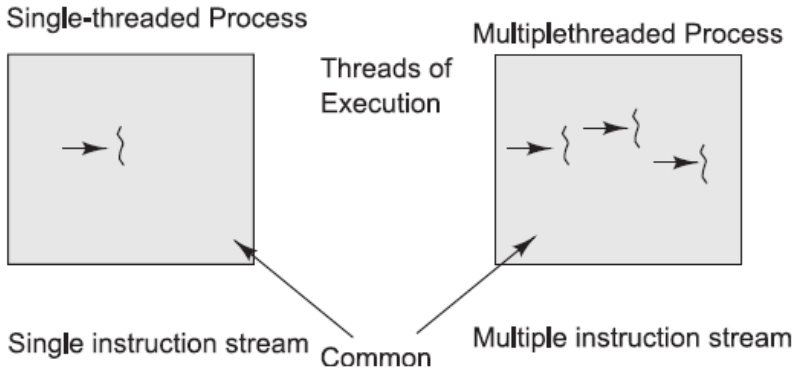
Note: A single processor computer is shared among multiple applications with preemptive multitasking, the processor is switching between the applications at intervals of milliseconds, you feel that all applications run concurrently.

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

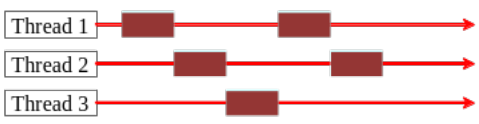
THREADS IN JAVA

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

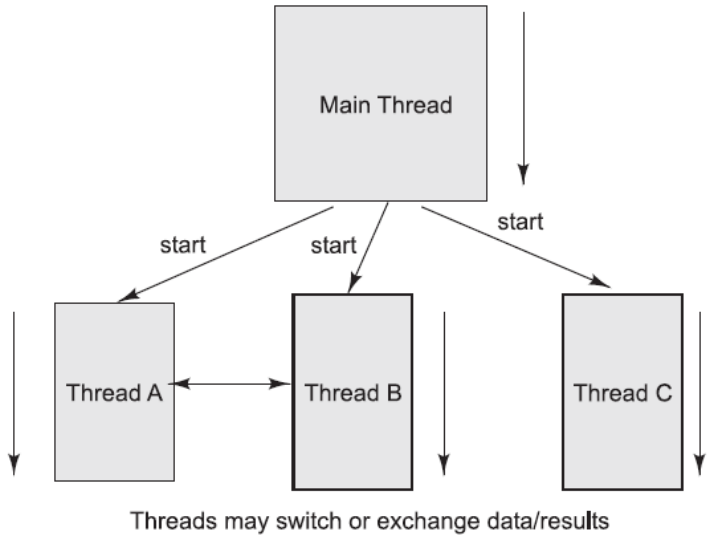
- All threads within a process share the same state and same memory space, and can communicate with each other directly, because they share the same variables.
- A single process might contain multiple threads.
- Java supports thread-based multitasking
- Threads are lightweight processes as the overhead 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

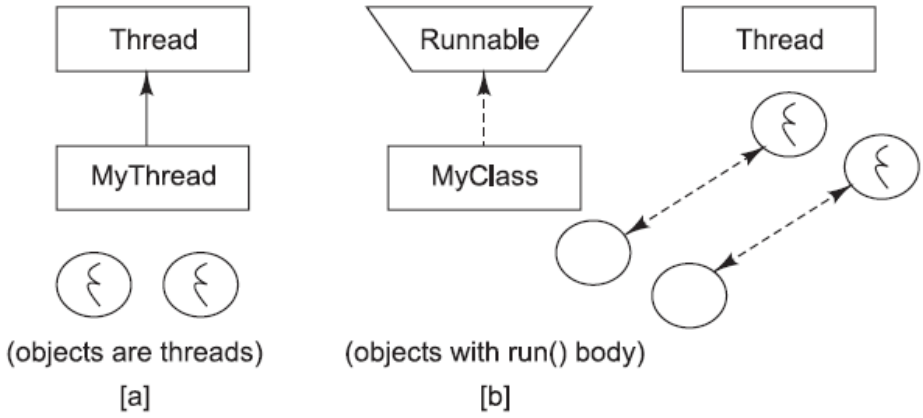


2.1.1 Why do we need threads?

- 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:

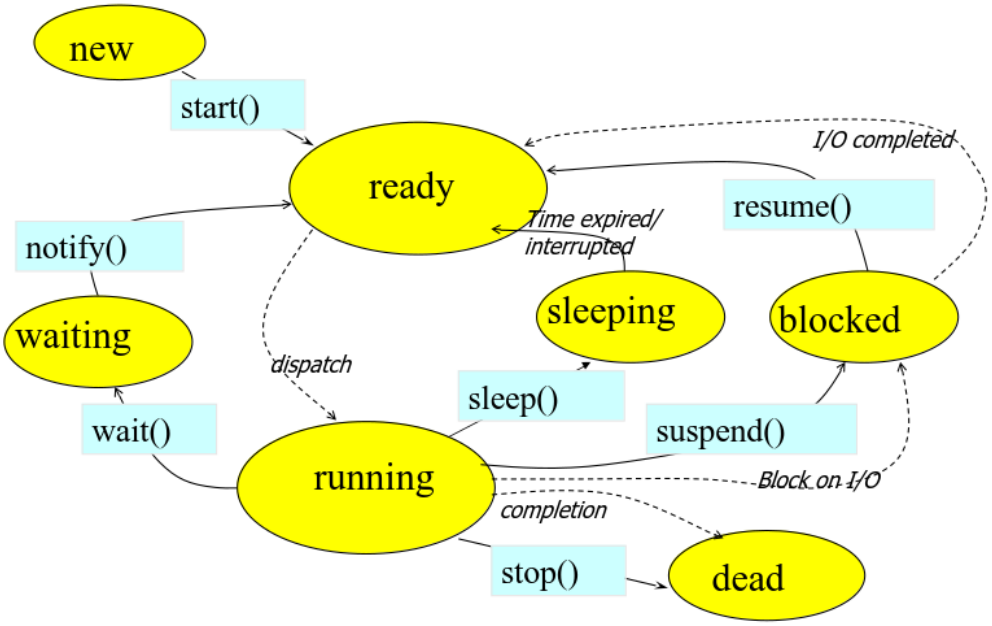
```
thr1.start();
```

Example

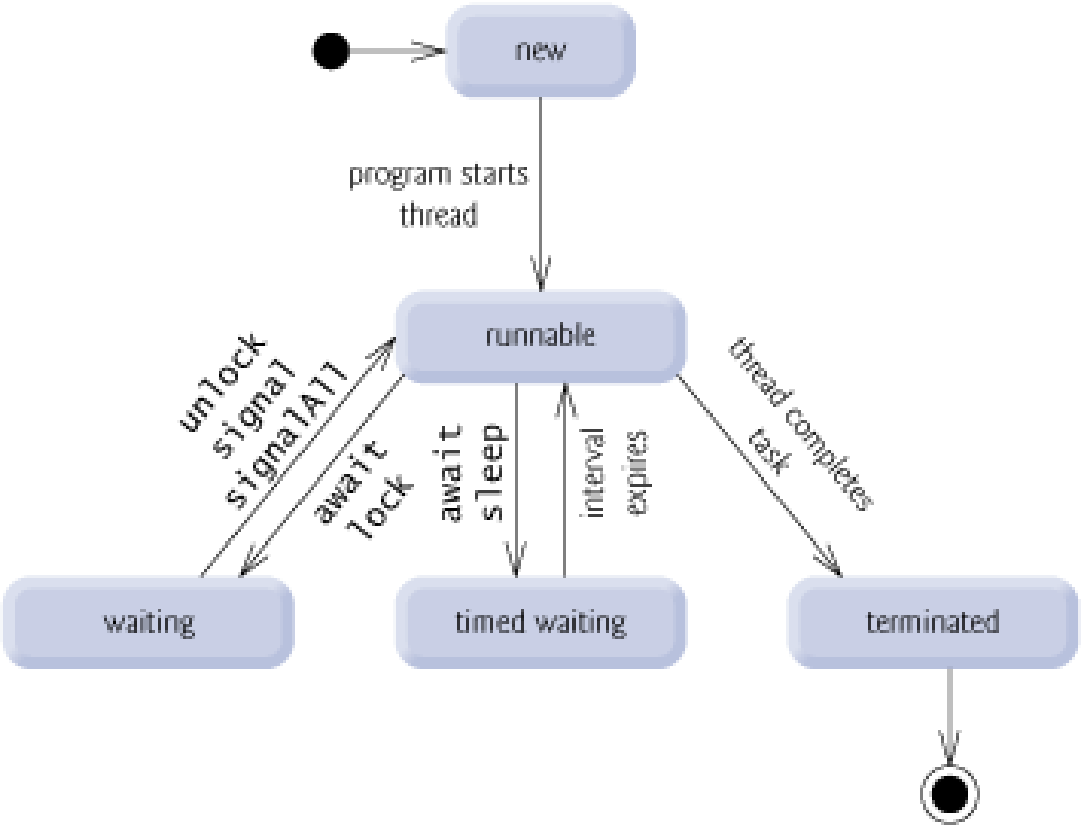
```
/* ThreadEx2.java: A simple program creating and invoking a thread object
by implementing Runnable interface. */
class MyThread implements Runnable {
    public void run() {
        System.out.println("this thread is running ...");
    }
}
class ThreadEx2 {
    public static void main(String [] args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

2.3 Life cycle of threads & Thread states

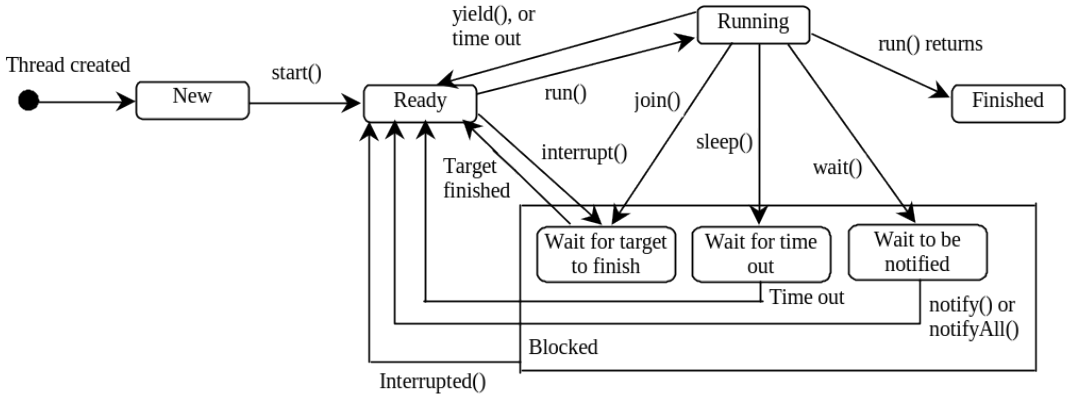
2.3.1 Life Cycle of Thread



2.3.2 Thread States



A thread can be in one of five states: New, Ready, Running, Blocked, or Finished showed in figure(2.3.2).



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.
- 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 void 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 synchronized void 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 void 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 void 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() ...

2.7 Object locking

- The object can be explicitly locked in this way

```
synchronized(myInstance){
    try{
        wait();
    }catch(InterruptedException ex){}

    System.out.println("Iam in this");
    notifyAll();
}
```

- The synchronized keyword locks the object. The wait keyword waits for the lock to be acquired, if the object was already locked by another thread. notifyall() notifies other threads that the lock is about to be released by the current thread.
- Another method notify() is available for use, which wakes up only the next thread which is in queue for the object, notifyall() wakes up all the threads and transfers the lock to another thread having the highest priority.

2.8 Thread Priority

- In Java, each thread is assigned priority, which affects the order in which it is scheduled for running. The threads so far had same default priority (NORM_PRIORITY) and they are served using FCFS policy.
- Java allows users to change priority:

```
ThreadName.setPriority(intNumber)
```

- MIN_PRIORITY = 1
- NORM_PRIORITY = 5
- MAX_PRIORITY = 10

2.9 Shared Resources

- If one thread tries to read the data and other thread tries to update the same data, it leads to inconsistent state.
- This can be prevented by synchronizing access to the data.
- Use "synchronized" method:

```
public synchronized void update()
{
    ....
}
```

2.9.1 the driver: 3 Threads sharing the same object

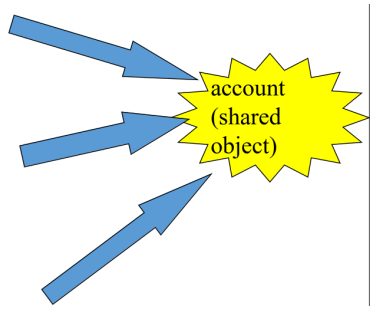
```
class InternetBankingSystem {
    public static void main(String [] args  ) {
        Account accountObject = new Account ();
        Thread t1 = new Thread(new MyThread(accountObject));
            Thread t2 = new Thread(new YourThread(accountObject));
            Thread t3 = new Thread(new HerThread(accountObject));
        t1.start();
        t2.start();
        t3.start();
        // DO some other operation
    } // end main()
}
```

2.9.2 Shared account object between 3 threads

```
class MyThread implements Runnable  {
 Account account;
        public MyThread (Account s) {  account = s;}
        public void run() { account.deposit(); }
} // end class MyThread

class YourThread implements Runnable  {
 Account account;
        public YourThread (Account s) { account = s;}
        public void run() { account.withdraw(); }
} // end class YourThread
```

```
class HerThread implements Runnable  {
 Account account;
        public HerThread (Account s) { account = s; }
        public void run() {account.enquire(); }
} // end class HerThread
```



2.9.3 Monitor (shared object access): serializes operation on shared objects

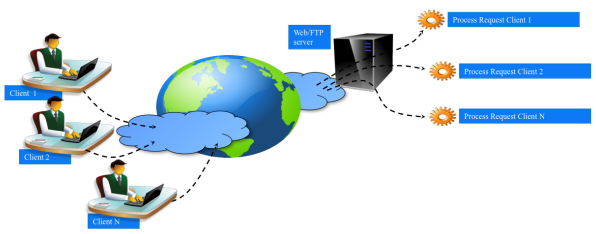
```
class Account {   // the 'monitor'
    int balance;

 // if 'synchronized' is removed, the outcome is unpredictable
  public synchronized void deposit( ) {
     // METHOD BODY : balance += deposit_amount;
   }

    public synchronized void withdraw( ) {
      // METHOD BODY: balance -= deposit_amount;
    }
    public synchronized void enquire( ) {
      // METHOD BODY: display balance.
    }
 }
```

2.10 Threaded Applications

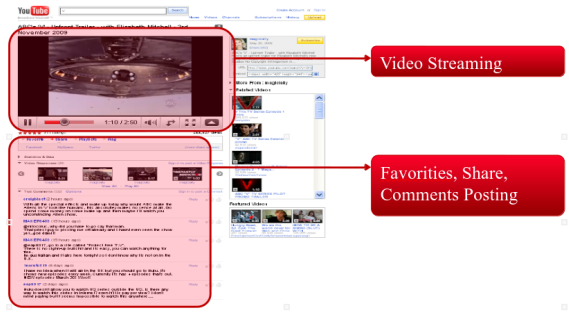
2.10.1 Multithreaded Web Server: For Serving Multiple Clients Concurrently



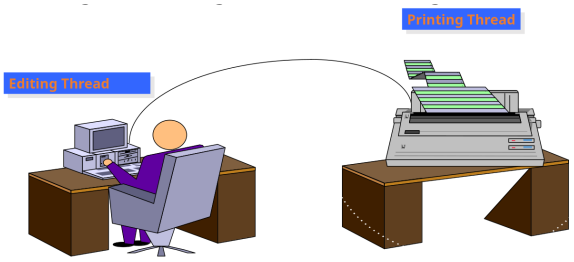
Architecture for Multithread Servers

- Multithreading enables servers to maximize their throughput, measured as the number of requests processed per second.
- Threads may need to treat requests with varying priorities:
  - A corporate server could prioritize request processing according to class of customers.
- Architectures:
  - Worker pool
  - Thread-per-request
  - Thread-per-connection
  - Thread-per-object

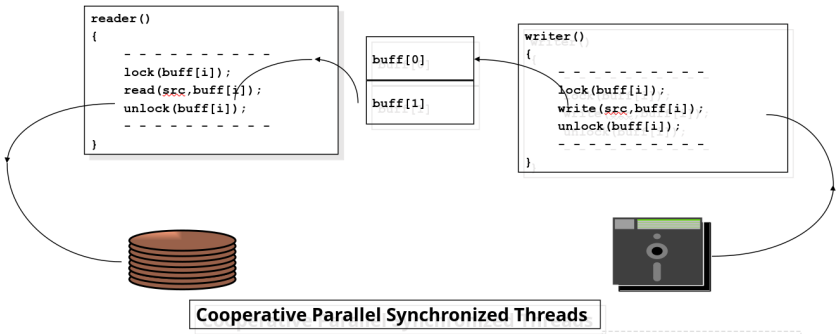
2.10.2 Internet Browser + Youtube



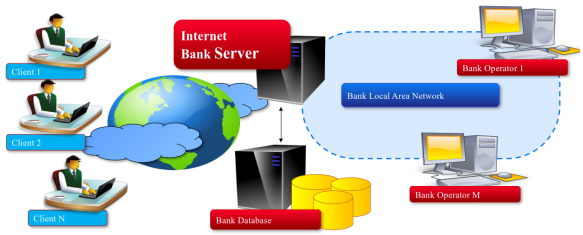
2.10.3 Editing and Printing documents in background



2.10.4 Multithreaded/Parallel File Copy



2.10.5 Online Bank: Serving Many Customers and Operations



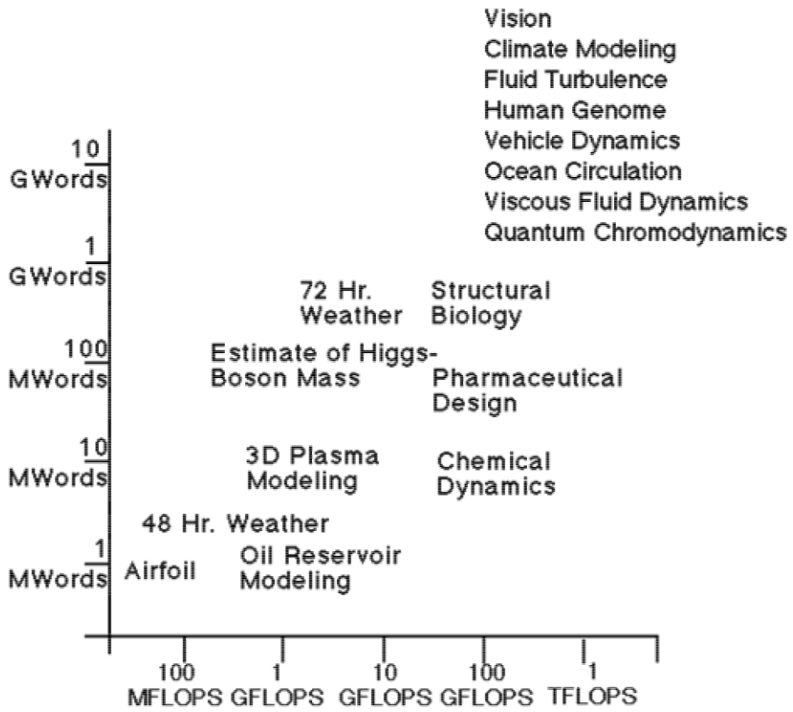
Chapter 3

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.

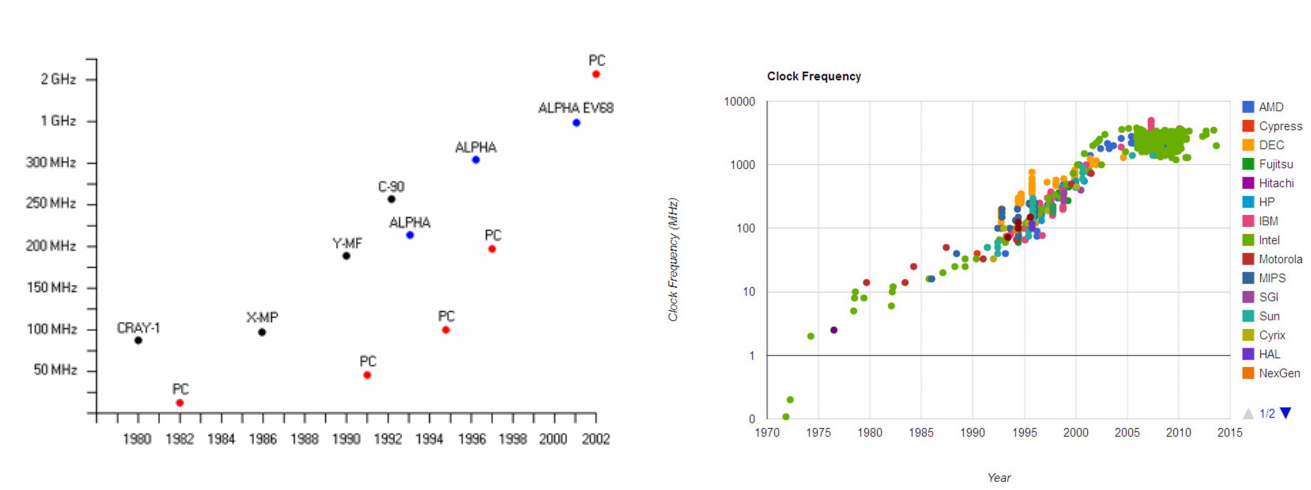


- The performance capabilities of supercomputers are expressed using a standard rate for indicating the number of floating-point arithmetic calculations systems can perform on a per-second basis. The rate, **floating-point operations per second**, is abbreviated as **FLOPS**.

- The per-second rate "FLOPS" is commonly misinterpreted as the plural form of "FLOP" (short for "floating-point operation") [2]

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 50% faster. The rest of the speed increase was due to increased use of parallel techniques:

- More processors (8 → 16)
- Longer vector pipes (64 → 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.

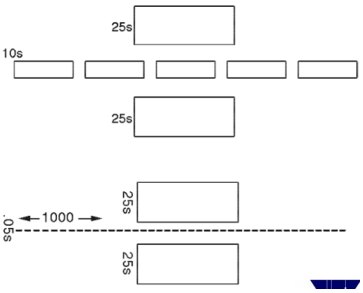
3.3 Amdahl's Law

How many processors can we really use?

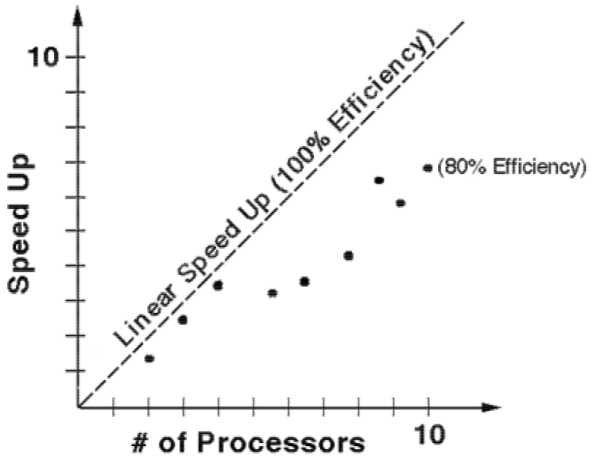
Let's say we have a legacy code such that is it only feasible to convert half of the heavily used routines to parallel:

- 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.

25s	Serial
50s	Parallel
25s	Serial



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 $\text{Efficiency}_\text{n}$ is:

$$\text{Efficiency}_\text{n} = \frac{Speedup_n}{n} \tag{3.5}$$

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

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

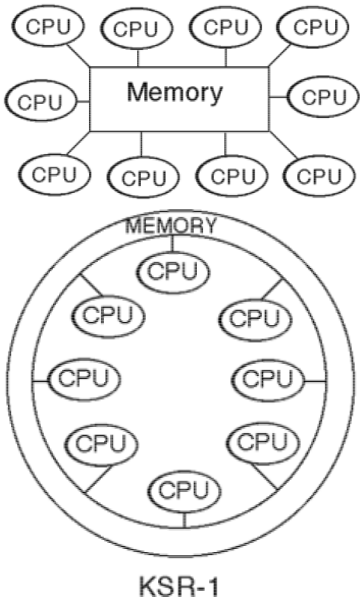
Example 2

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.

3.4.2 Distributed Memory

- Number of processors only limited by physical size (tens of meters).
- Memory only limited by the number of processors time the maximum memory per processor (very large). However, physical packaging usually dictates no local disk per node and hence no virtual memory.
- Since local and remote data have much different access times, data distribution is very important. We must minimize communication.

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)

Common Distributed Memory Machines

- CM-2
- CM-5
- T3E
- Workstation Cluster
- SP3
- TCS

3.4.3 Common parallel computer architectures [1]

- **SIMD** and **MIMD** are types of computer architectures that are used to improve the performance of certain types of computational tasks.
- The basis of this classification is the number of data and instruction streams.
- **SIMD**, short for Single Instruction Multiple Data, computer architecture can execute a single instruction on multiple data streams.
- On the other hand, the **MIMD** (Multiple Instruction Multiple Data) computer architectures can execute several instructions on multiple data streams.
- While the CM-2 is SIMD (one instruction unit for multiple processors), all the new machines are MIMD (multiple instructions for multiple processors) and based on commodity processors.
  - SP-2
  - CM-5
  - T3E
  - Workstations
  - TCS
  - POWER2
  - SPARC
  - Alpha
  - Your Pick
  - Alpha
- Therefore, the single most defining characteristic of any of these machines is probably the network.

3.5 Networking for distributed machines

Even with the "perfect" network we have here, performance is determined by two more quantities that, **together with the topologies** we'll look at, pretty much define the network: **latency** and **bandwidth**.

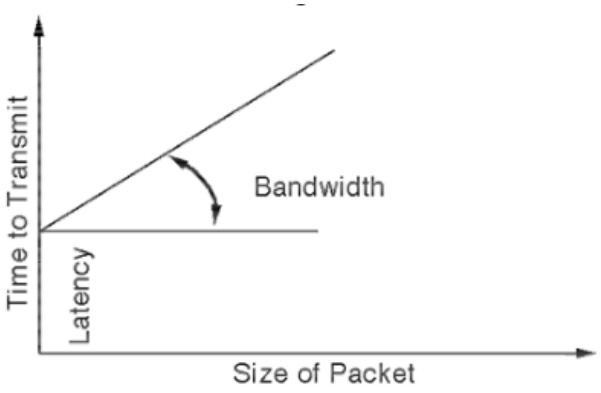
- latency
- bandwidth
- topologies

3.5.1 Latency

- Latency can nicely be defined as the time required to send a message with 0 bytes of data.
- This number often reflects either:
  - the overhead of packing your data into packets,
  - or the delays in making intervening hops across the network between two nodes that aren't next to each other.

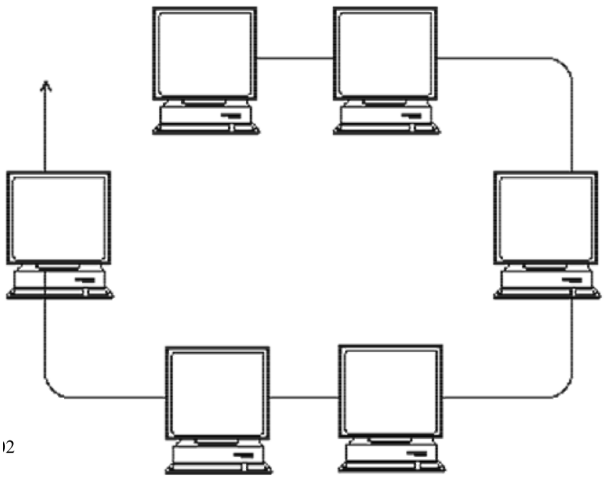
3.5.2 Bandwidth

- Bandwidth is the rate at which very large packets of information can be sent.
- If there was no latency, this is the rate at which all data would be transferred.
- It often reflects the physical capability of the wires and electronics connecting nodes.

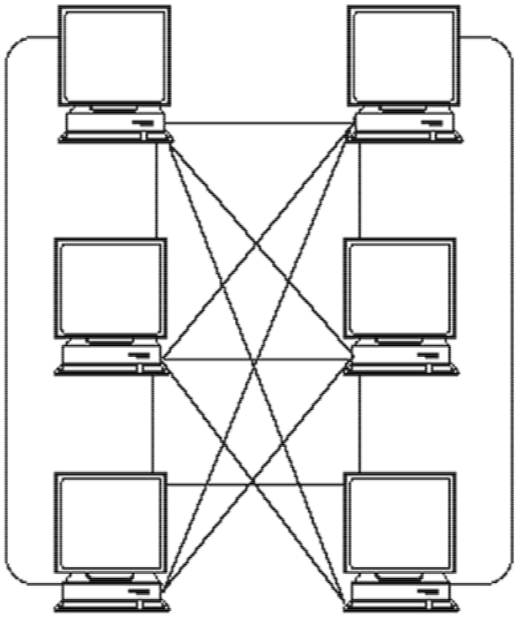


3.5.3 Topologies

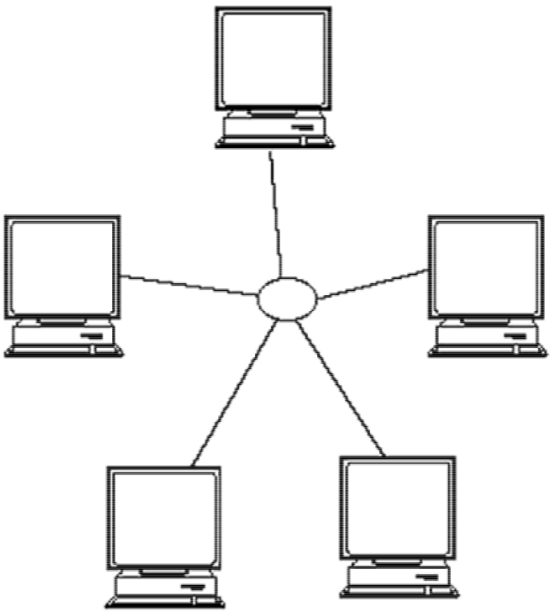
Token-Ring/Ethernet with Workstations



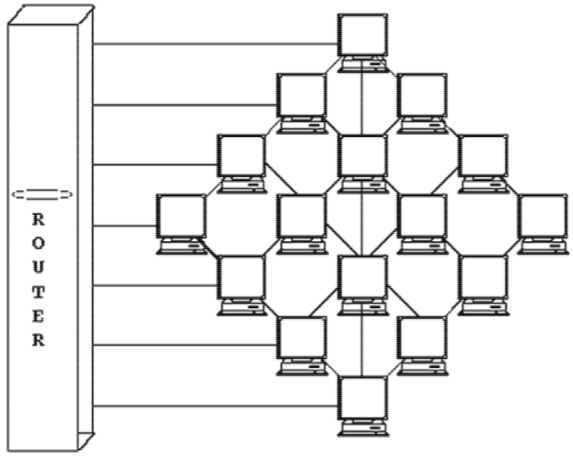
Complete Connectivity



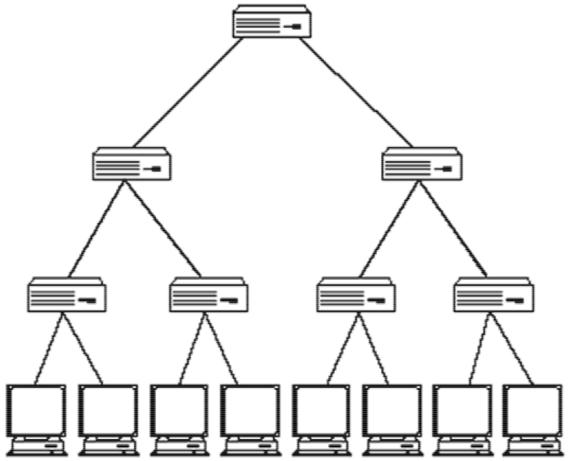
Super Cluster / SP2



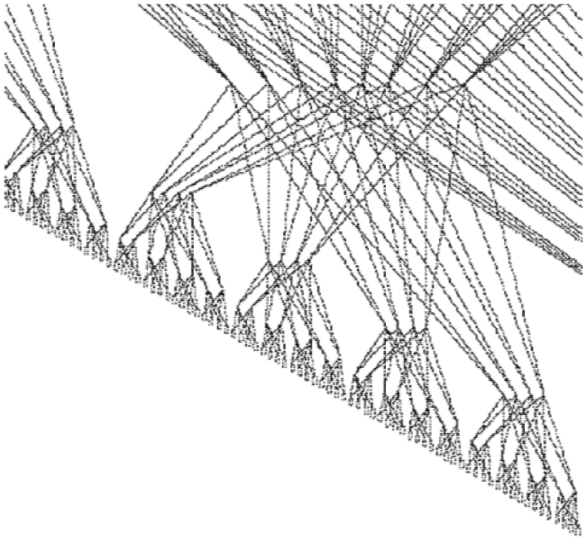
CM-2



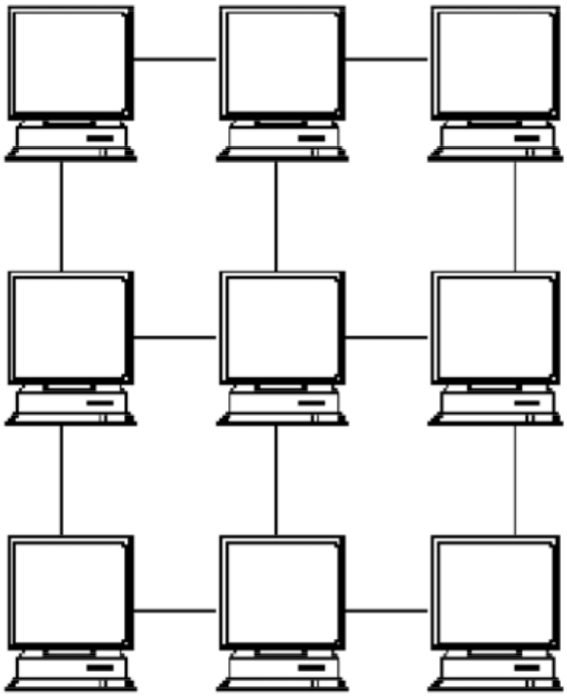
Binary Tree



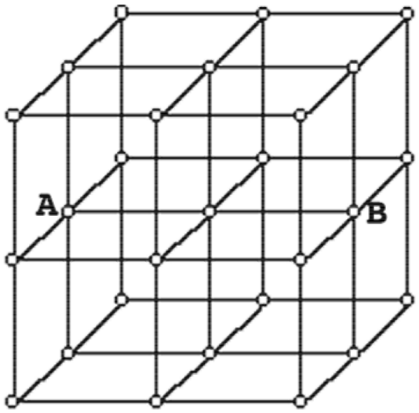
CM-5 Fat Tree



INTEL Paragon (2-D Mesh)

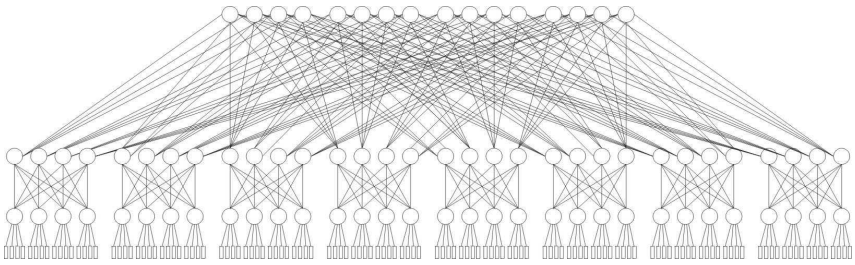


3-D Torus



- T3E has Global Addressing hardware, and this helps to simulate shared memory.
- Torus means that "ends" are connected. This means A is really connected to B and the cube has no real boundary.

TCS Fat Tree

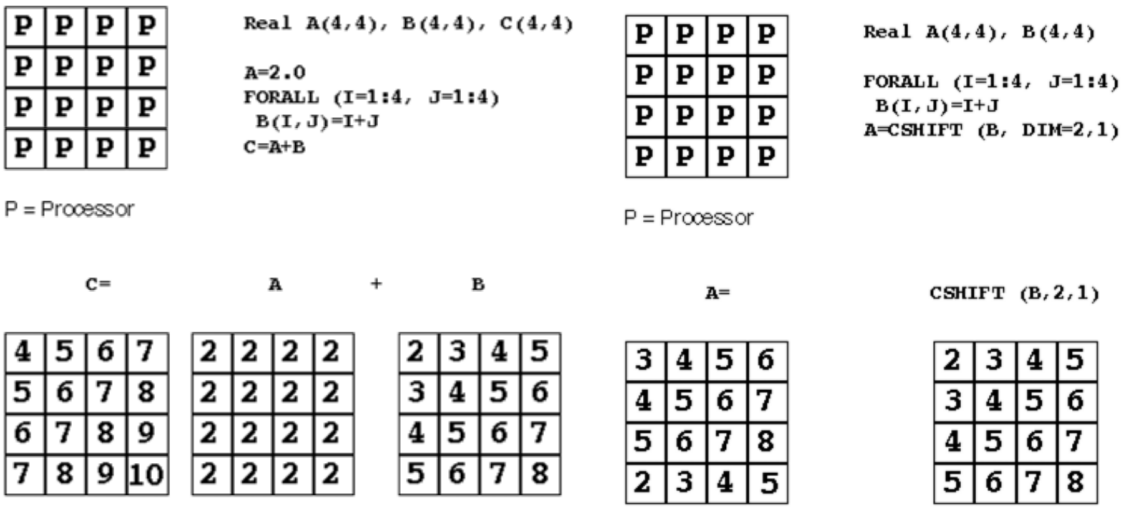


3.6 Data Parallel vs Work Sharing

–	**Data Parallel**	**Work Sharing**
–	Only one executable	Splits up tasks (as opposed to arrays in date parallel) such as loops amongst separate processors
computation	Do computation on arrays of data using array operators	Do computation on loops that are automatically distributed
communication	Do communications using array shift or rearrangement operators.	Do communication as a side effect of data loop distribution. Not important on shared memory machines.
Good for	Good for problems with static load balancing that are array-oriented SIMD machines	Good for shared memory implementations.
Strengths	1. Scales transparently to different size machines 2. Easy debugging, as there is only one copy of the code executing in a highly synchronized fashion	1. Directive based, so it can be added to existing serial codes
Weaknesses	1. Much wasted synchronization 2. Difficult to balance load	1. Limited flexibility 2. Efficiency dependent upon structure of existing serial code 3. May be very poor with distributed memory.
Variants	• FORTRAN 90 • CM FORTRAN • HPF • C* • CRAFT	• CRAFT • Multitasking

When to use	When to use Data Parallel:	When to use Work Sharing:
	• Very array-oriented programs – FEA – Fluid Dynamics – Neural Nets – Weather Modeling • Very synchronized operations – Image processing – Math analysis	• Very large / complex / old existing codes: Gaussian 90 • Already multitasked codes: Charmm • Portability (Directive Based) • (Not Recommended)

3.6.1 Data Parallel - Data Movement in FORTRAN 90



3.6.2 Work Sharing - CRAYs

If you have used CRAYs before, this of this as “advanced multitasking”
Explain: if you have used CRAYs before, you can think of work sharing as ”advanced multitasking.” This is because work sharing allows you to split up tasks and distribute them among multiple processors, which can significantly improve the performance of your code.

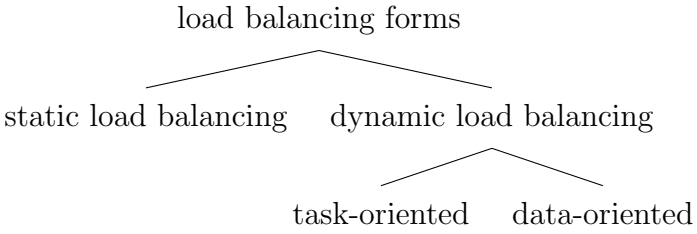
3.7 Load Balancing

- An important consideration which can be controlled by communication is load balancing:
  Consider the case where a dataset is distributed evenly over 4 sites.
  Each site will run a piece of code which uses the data as input and attempts to find a convergence.
  It is possible that the data contained at sites 0, 2, and 3 may converge much faster

than the data at site 1.
If this is the case, the three sites which finished first will remain **idle** while site 1 finishes.
When attempting to balance the amount of work being done at each site, one must take into account:

  - the speed of the processing site,
  - the communication "expense" of starting and coordinating separate pieces of work,
  - and the amount of work required by various pieces of data.

- There are two forms of load balancing: static and dynamic.



3.7.1 Static Load Balancing

- In static load balancing, the programmer must make a decision and assign a **fixed amount** of work to each processing site a priori.
- Static load balancing can be used in either the
  - Master-Slave (Host-Node) programming model
  - or the "Hostless" programming model.
- Static Load Balancing yields **Good performance** when:
  - homogeneous cluster
  - each processing site has an equal amount of work
- **Poor performance** when:
  - heterogeneous cluster where some processors are much faster (unless this is taken into account in the program design)
  - work distribution is uneven

3.7.2 Dynamic Load Balancing

- Dynamic load balancing can be further divided into the categories:
  - **task-oriented (commonly used form)**
    when one processing site finishes its task, it is assigned another task (this is the most commonly used form).

  - **data-oriented (more complicated)**
    when one processing site finishes its task before other sites, the site with the most work gives the idle site some of its data to process (this is much more complicated because it requires an extensive amount of bookkeeping).

- Dynamic load balancing can be used only in the Master-Slave programming model.
- ideal for:
  - codes where tasks are large enough to keep each processing site busy
  - codes where work is uneven
  - heterogeneous clusters

Chapter 4

Parallelism, Threads, and Concurrency

4.1 Our Goals

- Appreciate the (increasing) importance of parallel programming
- Understand fundamental concepts:
  - Parallelism, threads, multi-threading, concurrency, locks, etc.
- See some basics of this is done in Java
- See some common uses:
  - Divide and conquer, e.g. mergesort
  - Worker threads in Swing

4.2 Notes!

- An area of rapid change!
  - 1990s: parallel computers were $$$$
  - Now: 4 core machines are commodity
- Variations between languages
- Old dogs and new tricks? Not so good...
  - Educators, Java books, web pages
- Evolving frameworks, models, etc.
  - E.g. Java's getting Fork/Join in Java 1.7 (summer 11)
  - MAP/REDUCE

4.3 (Multi)Process vs (Multi)Thread

- Assume a computer has one CPU
- Can only execute one statement at a time
  - Thus one program at a time
- Process: an operating-system level "unit of execution"
- Multi-processing
  - Op. Sys. "time-slices" between processes
  - Computer appears to do more than one program (or background process) at a time

4.4 Tasks and Threads

- Thread: "a thread of execution"
  - "Smaller", "lighter" than a process
  - smallest unit of processing that can be scheduled by an operating system
  - Has its own run-time call stack, copies of the CPU's registers, its own program counter, etc.
  - Process has its own memory address space, but threads share one address space
- A single program can be multi-threaded
  - Time-slicing done just like in multiprocessing
  - Repeat: the threads share the same memory

4.5 Task

- A task is an abstraction of a series of steps
  - Might be done in a separate thread
- In Java, there are a number of classes / interfaces that basically correspond to this
  - Example (details soon): Runnable
    - work done by method run()

4.6 Java: Statements → Tasks

- Consecutive lines of code:

```
Foo tmp = f1;
f1 = f2;
f2 = tmp;
```

- A method:

```
swap (f1, f2);
```

- A "task" object:

```
Swap Task task1= new SwapTask(f1,2);
task1.run;
```

4.7 Huh? Why a task object?

- Actions, functions vs. objects. What's the difference?
- Objects:
  - Are persistent. Can be stored.
  - Can be created and then used later.
  - Can be attached to other things. Put in Collections.
  - Contain state.
- Functions:
  - Called, return (not permanent)

4.8 Java Library Classes

- For task-like things:
  - Runnable, Callable
  - SwingWorker, RecursiveAction, etc.
- Thread class
- Managing tasks and threads
  - Executor, ExecutorService
  - ForkJoinPool
- In Swing
  - The Event-Dispatch Thread
  - SwingUtilities.invokeLater()

4.9 Java's Nested Classes

- You can declare a class inside another
  - `http://download.oracle.com/javase/tutorial/java/java00/nested.html`
- If declared static, can use just like any class
- If not static
  - Can only define objects of that type from within non-static code of the enclosing class
  - Object of inner-class type can access all fields of the object that created it. (Useful!)
  - Often used for "helper" classes, e.g. a node object used in a list or tree.
- See demo done in Eclipse: TaskDemo.java

4.10 Possible Needs for Task Objects

- Can you think of any?
- Storing tasks for execution later
  - Re-execution
- Undo and Redo
- Threads

4.11 Undo Operations

- A task object should:
  - Be able to execute and undo a function
  - Therefore will need to be able to save enough state to "go back"
- When application executes a task:
  - Create a task object and make it execute
  - Store that object on a undo stack
- Undo
  - Get last task object stored on stack, make it undo

4.12 slide15: Calculator App Example

- We had methods to do arithmetic operations:

```
public void addToMemory(double inputVal) {
    memory = memory + inputVal;
}
```

- Instead:

```
public void addToMemory(double inputVal) {
    AddTask task = new AddTask(inputVal);
    task.run();
    undoStack.add (task);
}
```

4.13 slide16: Stack, Undo Stack

- A Stack is an important ADT
  - A linear sequence of data
  - Can only add to the end, remove item at the end
  - LIFO organization: "last in, first out"
  - Operations: push(x), pop(), sometimes top()
- Stacks important for storing delayed things to return turn
  - Run-time stack (with activation records)
  - An undo stack (and a separate redo stack)

4.14 slide17: Nested class for Adding

```
private class AddTask implements UndoableRunnable {
    private double param;
    public AddTask(double inputVal) {
        this.param = inputVal;
    }
    public void run() { // memory is field in CalcApp
        memory = memory + this.param;
    }
    public boolean undo() {
        memory = memory - this.param;
        return true;
    }
}
```

4.15 slide18: Undo operation

- In the Calc app:

```
public boolean undo() {
    boolean result = false;
    int last = undoStack.size()-1;
    if ( last >= 0){
        UndoableRunnable task = undoStack.get(last);
        result = task.undo();
        undoStack.remove(last);
    }
    return result;
}
```

4.16 slide19: Java Thread Classes and Methods

- Java has some "primitives" for creating and using threads
  - Most sources teach these, but in practice they're hard to use well
  - Now, better frameworks and libraries make using them directly less important.
- But let's take a quick look

4.17 slide20: Java's Thread Class

- Class Thread: it's method run() does its business when that thread is run
- But you never call run(). Instead, you call start() which lets Java start it and call run ()
- To use Thread class directly (not recommended now):
  - define a subclass of Thread and override run () - not recommended!
  - Create a task as a Runnable, link it with a Thread, and then call start() on the Thread.
    - The Thread will run the Runnable's run () method.

4.18 slide21: Creating a Task and Thread

- Again, the first of the two "old" ways
- Get a thread object, then call start() on that object
  - Makes it available to be run
  - When it's time to run it, Thread's run() is called
- So, create a thread using inheritance

  - Write class that extends Thread, e.g. MyThread
  - Define your own run()
  - Create a MyThread object and call start() on it

- We won't do this! Not good design!

4.19 slide22: Runnables and Thread

- Use the "task abstraction" and create a class that implements Runnable interface
  - Define the run () method to do the work you want
- Now, two ways to make your task run in a separate thread
  - First way:
    * Create a Thread object and pass a Runnable to the constructor
    * As before, call start() on the Thread object
  - Second way: hand your Runnable to a "thread manager" object
    * Several options here! These are the new good ways. More soon.

4.20 slide23: Join (not the most descriptive word)

- The Threadclass defines various primitive methods you could not implement on your own
  - For example: start, which calls run in a new thread
- The join () method is one such method, essential for coordination in this kind of computation
  - Caller blocks until/unless the receiver is done executing (meaning its run returns)
  - E.g. in method foo() running in "main" thread, we call: myThread.start(); myThread.join();
  - Then this code waits ("blocks") until myThread's run() completes
- This style of parallel programming is often called "fork/join"
  - Warning: we'll soon see a library called "fork/join" which simplifies things. In that, you never call join()

4.21 slide30: New Java ForkJoin Framework

- Designed to support a common need
  - Recursive divide and conquer code
  - Look for small problems, solve without parallelism

  - For larger problems
    * Define a task for each subproblem
    * Library provides
      · a Thread manager, called a ForkJoinPool
      · Methods to send your subtask objects to the pool to be run, and your call waits until their done
      · The pool handles the multithreading well

4.22 slide35: Same Ideas as Thread But...

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a ForkJoinPool)

Don't subclass Thread
Don't override run
Don't call start
Don't just call join
or

Do subclass RecursiveTask< V >
Do override compute
Do call invoke, invokeAll, fork
Do call join which returns answer

Do call invokeAll on multiple tasks

Bibliography

[1] Kiran Kumar Panigrahi. Difference between simd and mimd. https://www.tutorialspoint.com/difference-between-simd-and-mimd, 2022.

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