الجامعة التكنولوجية

قسم علوم الحاسوب

Parallel programming paradigms

3ed Class, SW branch, Second Semester.

(2025-2024)

By: Teaba Wala aldeen Khairi.

University of technology, Computer science department.

E-Mail: teaba.w.khairi@uotechnology.edu.iq

Requirements for the course

- Google doc
- Google slide & using Google meet

To pass the course the student must do:

-1 Home works ..…………

-2 Reports..…………………

-3 Group project…………………

-5 Final exam.………………

-6 Mid exam..………………

Class 2021-2020 first course 3rd / Parallel programming paradigm

Content of the lectures

CH1 FOUNDATIONS OF PARALLEL PROGRAMMING

- Motivation for parallel programming
- Need to increasing performance
- Building parallel system
- Need to write parallel programs.
- How to write parallel programs.
- Concurrence, parallel , distributed.

CH 2 PARALLEL HARDWARE AND PARALLEL SOFTWARE

- Process, multitasking and threading
- Instruction level parallelism
- SIMD
- MIMD
- Parallel program design
- Writing and running parallel program

CH3 DISTRIBUTED-MEMORY PROGRAMING WITH MPI

- Basic MPI programming
- Compilation and execution
- MPI programs
- MPI_Init and MPI_Finalize
- Communicator , MPI_Comm_size and MPI_Comm_rank
- MPI_Send
- MPI_Recv
- Performance evaluation of MPI programs
    - taking trimming
    - results
    - speedup and efficiency
    - scalability
- A parallel sorting algorithm
    - Some simple serial sorting algorithm
    - Parallel odd-even transposition sort
    - Safety in MPI programs
    - Final details of parallel odd-even sort

CH4 SHARED MEMORY PROGRAMMING WITH PTHREADS

- Process , thread, and Pthreads
- Hello, world
    - Execution
    - Preliminaries
    - Starting the thread
    - Running the thread
    - Stopping the thread

    - Error checking
    - Other approaches to thread start up
- Read-write locks
    - Linked list function
    - A multi-thread Linked list
    - Pthread read-write locks
    - performance of the varioce implementation
    - implementing read-write locks

CH5 PAR ALLEL PROGRAMMING DEVELOPMENT

- Tree search
    - Recursive depth first search
    - Non recursive depth first search
    - Data structure for the serial implementation
    - Performance of the serial implementation
    - Parallelizing tree search
    - A static Parallelization of tree search using pthread
    - A dynamic Parallelization of tree search using pthread
    - Evaluation the pthread tree search programs
    - Performance of the implementation
    - Implementation of tree search and static partitioning
    - Implementation of tree search and dynamic partitioning

REFERENCES

[1] Peter S. Pacheco, “An introduction to parallel programming”, Morgan Kaufmann, 2011.

[2] C Lin, L Snyder. Principles of Parallel Programming. USA: Addison-Wesley Publishing Company, 2008.

[3] A Grama, A Gupra, G Karypis, V Kumar. Introduction to Parallel Computing (2nd ed.). Addison Wesley, 2003.

[4] M. J. Quinn, “Parallel programming in C with MPI and OpenMP”, Tata McGraw Hill, 2003.

[5] T Mattson, B Sanders, B Massingill. Patterns for Parallel Programming. Addison-Wesley Professional, 2004.

[6] Gergel V.P. (2007) Theory and Practice of Parallel Programming. Moscow, Intuit. (In Russian).

Requirements

Requirements for the course the student must have acknowledge of using :

- Google docs
- Google slide
- https://onlinegdb.com
- Code Blocks with threads.

Lecture1 Fundamental

Foundation of Parallel Programming

In computers, parallel processing is the processing of program instructions by dividing them among multiple processors with the objective of running a program in less time.

In the earliest computers, only one program ran at a time. A computation-intensive program that took one hour to run and a tape copying program that took one hour to run would take a total of two hours to run. An early form of parallel processing allowed the interleaved execution of both programs together. The computer would start an I/O operation, and while it was waiting for the operation to complete, it would execute the processor-intensive program. The total execution time for the two jobs would be a little over one hour.

Parallel processing is a method of simultaneously breaking up and running program tasks on multiple microprocessors, thereby reducing processing time.

Parallel processing may be accomplished via a computer with two or more processors or via a computer network. Parallel processing is also called *parallel computing*.

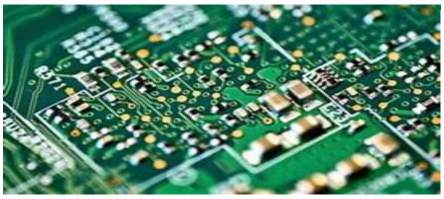
Parallel processing is particularly *useful* *when running programs that perform complex computations*, and it provides a viable option to the quest for cheaper computing alternatives.

Most computers have just one CPU, but some models have several. There are even computers with thousands of CPUs. With single-CPU computers, it is possible to perform parallel processing by connecting the computers in a network. However, this type of parallel processing requires very sophisticated software called distributed processing software.

Motivation Parallelism

Development of parallel software has traditionally been thought of as *time and effort intensive*. When viewed in the context of the brisk rate of development of microprocessors, one is tempted to question the need for devoting significant effort towards exploiting parallelism as a means of accelerating applications. It takes two years to develop a parallel application, during which time the underlying hardware and/or software platform has become obsolete, the *development effort* is clearly wasted. However, there are some unmistakable trends in hardware design, which indicate that uniprocessor (or implicitly parallel) architectures may not be able to sustain the rate of *realizable* performance increments in the future.

Smaller transistors = faster processors.
Faster processors = increased power consumption.
Increased power consumption = increased heat.
Increased heat = unreliable processors.



Why we Need to Increasing Performance

Computing a solution can take anywhere from hours to days. Parallel computing techniques *can help reduce the time it takes to reach a solution.* To derive the full benefits of parallelization, it is important to choose an approach that is appropriate for the optimization problem. As our computational power increases solving problems will be much easier.

Why we're Building Parallel Systems

Much of the tremendous increase in single processor performance has been driven by the ever-increasing density of transistors the electronic switches on integrated circuits. As the size of transistors decreases, their speed can be increased, and the overall speed of the integrated circuit can be increased. As the speed of transistors increases, their power consumption also increases. Most of this power is dissipated as heat, and when an integrated circuit gets too hot, it becomes unreliable. In the first decade of the twenty-first century, air-cooled integrated circuits are reaching the limits of their ability to dissipate heat.

Therefore, it is becoming impossible to continue to increase the speed of integrated circuits.

Q: How can we exploit the continuing increase in transistor density?

The answer is *parallelism.*

Rather than building ever-faster, more complex, monolithic processors, the industry has decided to put multiple, relatively simple, complete processors on a single chip. Such integrated circuits are called **multicore** processors, and **core** has become synonymous with central processing unit, or CPU. In this setting a conventional processor with one CPU is often called a **single-core** system.

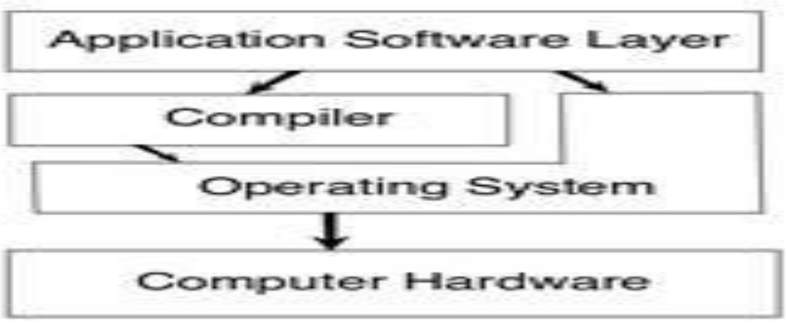
Q: Why Parallel?

- Because it's faster.
- Because it's cheaper!
- Because it's natural!

Why we Need to Write Parallel Programs

Most programs that have been written for conventional, **single-core** systems cannot exploit the presence of multiple cores. We can run multiple instances of a program on a multicore system, *but this is often of little help*. For example, being able to run multiple instances of our favorite game program isn't really what we want the program to run faster with more realistic graphics. *In order to do this, we need to either rewrite our serial programs so that they're parallel, so that they can make use of multiple cores, or write translation programs*, that is, programs that will automatically convert serial programs into parallel programs. (مهم فهم)

Parallel computing requires combining an understanding of hardware, software, and parallelism to develop an application. It is more than just message passing or threading. Current hardware and software give many different options to bring parallelism to your application. Some of these options can even be combined to yield even greater efficiency and speedup.



An efficient parallel implementation of a serial program may not be obtained by finding efficient parallelizations of each of its steps. Rather, the best parallelization may be obtained by stepping back and devising an entirely new algorithm. (مهم فهم)

As an example, suppose that we need to compute *n* values and add them together. We know that this can be done with the following serial code:

```
sum = 0;
for (i = 0; i < n; i++)
 { x = Compute next value(. . .);
sum += x; }
```

Now suppose we also have *p* cores and *p* is much smaller than *n*. Then each core can form a partial sum of approximately *n*=*p* values:

```
my sum = 0;   my first i = . . . ;    my last i = . . . ;
for (my i = my first i; my i < my last i; my i++) {
my x = Compute next value(. . .);
my sum += my x;}
```

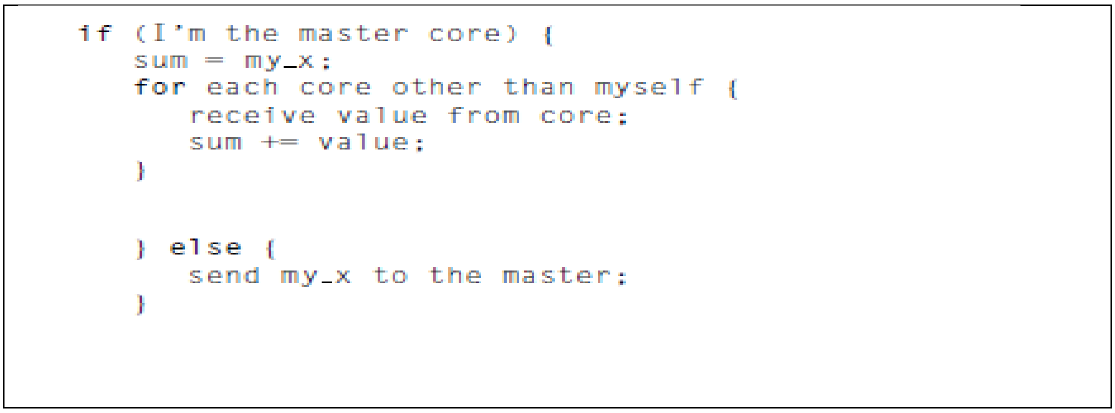
Here the prefix my indicates that each core is using its own, private variables, and each core can execute this block of code independently of the other cores. After each core completes execution of this code, its variable my sum will store the sum of the values computed by its calls to compute next value. For example, if there are eight cores, *n* = 24, and the 24 calls to Compute next value return the values:

1, 4, 3 , 9, 2, 8 , 5, 1, 1 , 6, 2, 7 , 2, 5, 0 , 4, 1, 8 , 6, 5 ,1 , 2, 3, 9

Then the values stored in my sum might be:

Core	**0 1 2 3 4 5 6 7**
my sum	**8 19 7 15 7 13 12 14**

Here we're assuming the cores are identified by nonnegative integers in the range0, 1, … ,*p*-1, where *p* is the number of cores. When the cores are done computing their values of my sum, they can form a global sum by sending their results to a designated "master" core, which can add their results:



```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }


   } else {
      send my_x to the master;
   }
```

In our example, if the **master core** is core **0**, it would add the values 8+19+7+ 15+7+13+12+14 = 95. But you can probably see a better way to do this especially if the number of cores is large. Instead of making the master core do all the work of computing the final sum, we can pair the cores so that while core 0 adds in the result of core 1, core 2 can add in the result of core 3, core 4 can add in the result of core 5 and so on. Then we can repeat the process with only the even-ranked cores: 0 adds in the result of 2, 4 adds in the result of 6, and so on. Now cores divisible by 4 repeat the process, and so on. See **Figure 1.1**. The circles contain the current value of each core's sum, and the lines with arrows indicate that one core is sending its sum to another core. The plus signs indicate that a core is receiving a sum from another core and adding the received sum into its own sum. For both “global” sums, the master core (core 0) does more work than any other core, and the length of time it takes the program to complete the final sum should be the length of time it takes for the master to complete. However, with eight cores, the master will carry out seven receives and adds using the first method, while with the second method it will only carry out three.

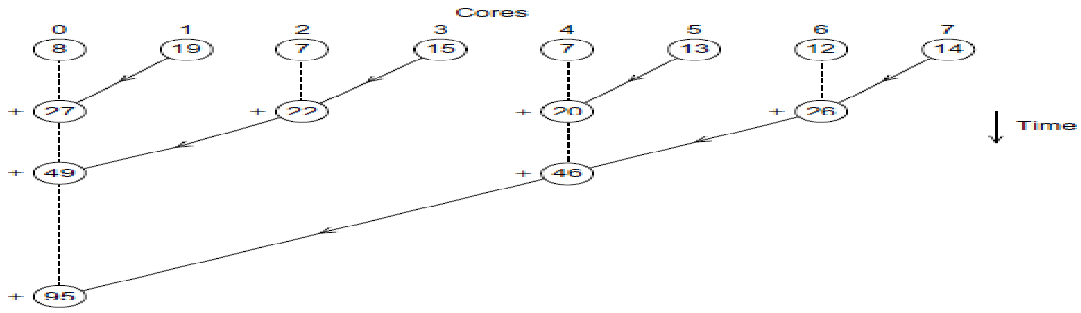


Figure (1.1) Multiple cores forming a global sum.

H.W: Does Parallelism considered useful all the time? مهم

1.5 How do we Write Parallel Programs?

There are a number of possible answers to this question, but most of them depend on the basic idea of ***partitioning*** the work to be done among the cores. There are two widely used approaches: **task-parallelism** and **data-parallelism**. In task-parallelism, we partition the various tasks carried out in solving the problem among the cores. In data-parallelism, we partition the data used in solving the problem among the cores, and each core carries out more or less similar operations on its part of the data.

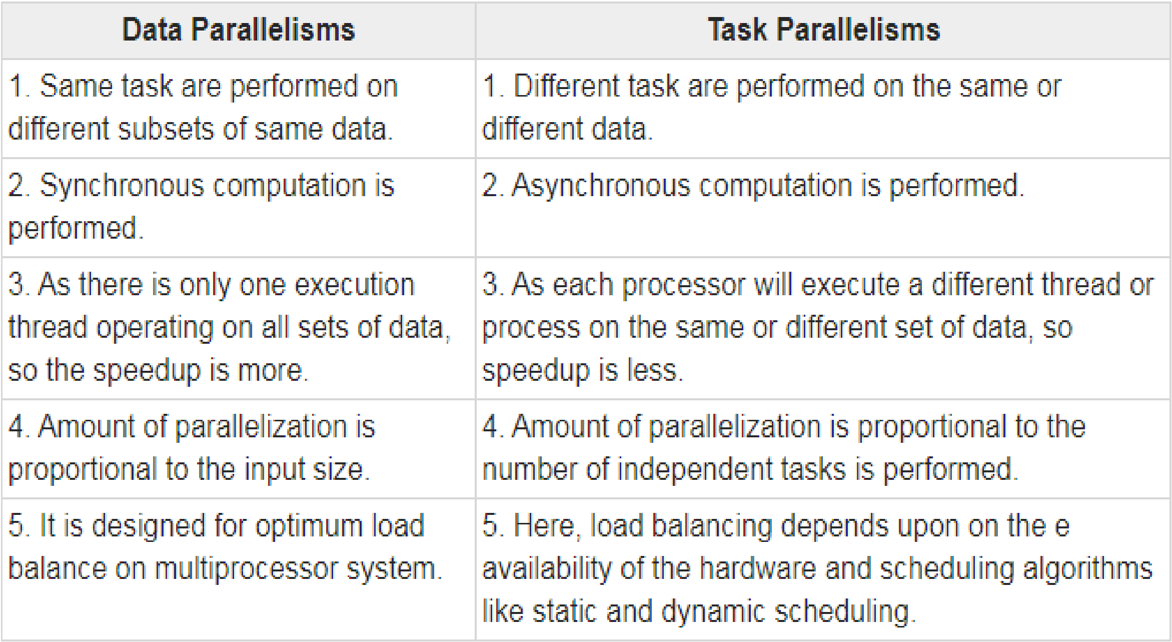
- **Data Parallelism**

Data Parallelism means concurrent execution of the same task on each multiple computing core. Let’s take an example, summing the contents of an array of size N. For a single-core system, one thread would simply sum the elements [0] . . . [N − 1]. For a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] and while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1]. So the Two threads would be running in parallel on separate computing cores.

- **Task Parallelism**

Task Parallelism means concurrent execution of the different task on multiple computing cores. Consider again our example above, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. Again The threads are operating in parallel on separate computing cores, but each is performing a unique operation.

The key differences between Data Parallelisms and Task Parallelisms are



Data Parallelisms	Task Parallelisms
1. Same task are performed on different subsets of same data.	1. Different task are performed on the same or different data.
2. Synchronous computation is performed.	2. Asynchronous computation is performed.
3. As there is only one execution thread operating on all sets of data, so the speedup is more.	3. As each processor will execute a different thread or process on the same or different set of data, so speedup is less.
4. Amount of parallelization is proportional to the input size.	4. Amount of parallelization is proportional to the number of independent tasks is performed.
5. It is designed for optimum load balance on multiprocessor system.	5. Here, load balancing depends upon on the e availability of the hardware and scheduling algorithms like static and dynamic scheduling.

1.6 Concurrent, Parallel, Distributed

In concurrent computing, a program is one in which multiple tasks can be *in progress* at any instant. In parallel computing, a program is one in which multiple tasks *cooperate closely* to solve a problem. In distributed computing, a program may need to cooperate with other programs to solve a problem. So parallel and distributed programs are concurrent, but a program such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core, since multiple tasks can be *in progress* at any instant. There isn’t a clear-cut distinction between parallel and distributed programs, but a parallel program usually runs multiple tasks simultaneously on cores that are physically close to each other and that either share the same memory or are connected by a very high-speed network. On the other hand, distributed programs tend to be more “loosely coupled.”

The tasks may be executed by multiple computers that are separated by large distances, and the tasks themselves are often executed by programs that were created independently.

Lecture2 Background

2.1 Introduction and Some Background

Parallel hardware and software have grown out of conventional **serial** hardware and software: hardware and software that runs (more or less) a single job at a time. A serial systems **The von Neumann architecture** The "classical" **von Neumann architecture** consists of :

- **Central Processing Unit (CPU)**

  The Central Processing Unit (CPU) is the electronic circuit responsible for executing the instructions of a computer program. It is sometimes referred to as the microprocessor or processor .The CPU contains the ALU, CU and a variety of registers.

  - **Arithmetic and Logic Unit (ALU)**

  The ALU allows arithmetic (add, subtract etc) and logic (AND,OR, NOT etc) operations to be carried out.

  - **Control Unit (CU)**

  The control unit controls the operation of the computer's ALU, memory and input/output devices, telling them how to respond to the program instructions it has just read and interpreted from the memory unit. The control unit also provides the timing and control signals required by other computer components. The control unit is responsible for deciding which instructions in a program should be executed. The control unit has a special register called the program counter. It stores the address of the next instruction to be executed.

  - **Registers**

    Data in the CPU and information about the state of an executing program are stored in special, very fast storage called registers.

- **Memory Unit**

  The memory unit consists of RAM, sometimes referred to as primary or main memory. Unlike a hard drive (secondary memory), this memory is fast and also directly accessible by the CPU.

  RAM is split into partitions. Each partition consists of an address and its contents (both in binary for.)The address will uniquely identify every location in the memory. Loading data from permanent memory (hard drive), into the faster and directly accessible temporary memory (RAM), allows the CPU to operate much quicker.

- **Buses**

  Buses are the means by which data is transmitted from one part of a computer to another, connecting all major internal components to the CPU and memory.

A von Neumann machine executes a single instruction at a time, and each instruction operates on only a few pieces of data. **See Figure 2.1.** When data or instructions are transferred from memory to the CPU, we sometimes say the data or instructions are **fetched** or **read** from memory. When data are transferred from the CPU to memory, we sometimes say the data are **written to memory** or **stored**.

The separation of memory and CPU is often called the **von Neumann Bottleneck** , since the interconnect determines the rate at which instructions and data can be accessed. The potentially vast quantity of data and instructions needed to run a program is effectively isolated from the CPU. In 2010 CPUs are capable of executing instructions more than one hundred times faster than they can fetch items from main memory.

H.W : What is von Neumann main problem?

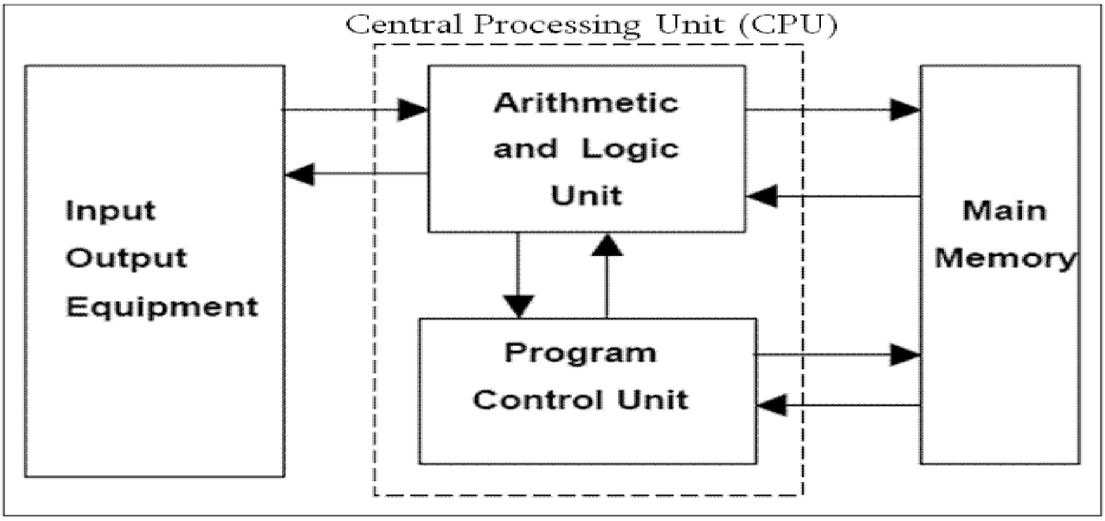


Figure : General structure of Von Neumann Architecture

Figure(2.1) The von Neumann architecture.

Processes, Multitasking, and Threads

Operating system, or OS, is a major piece of software whose purpose is to manage hardware and software resources on a computer. It determines which programs can run and when they can run. It also controls the allocation of memory to running programs and access to peripheral devices such as hard disks and network interface cards.

When a user runs a program, the operating system creates a **process** an instance of a computer program that is being executed. A process consists of several entities.

- The executable machine language program.
- A block of memory, which will include the executable code, a **call stack** that keeps track of active functions, a **heap**, and some other memory locations.
- Descriptors of resources that the operating system has allocated to the process for example, file descriptors.
- Security information for example, information specifying which hardware and software resources the process can access.
- Information about the state of the process, such as whether the process is ready to run or is waiting on some resource, the content of the registers, and information about the process' memory.

A process : is an instance of a program running in a computer. It is close in meaning to task , a term used in some operating systems. In some operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program). Like a task, a process is a running program with which a particular set of data is associated so that the process can be kept track of. An application that is being shared by multiple users will generally have one process at some stage of execution for each user.

Multitasking: Multitasking is when a CPU is provided to execute multiple tasks at a time. Multitasking involves often CPU switching between the tasks, so that users can collaborate with each program together. Unlike multithreading, In multitasking, the processes share separate memory and resources. As multitasking involves CPU switching between the tasks rapidly, So the little time is needed in order to switch from the one user to next.

Multithreading: Multithreading is a system in which many threads are created from a process through which the computer power is increased. In multithreading, CPU is provided in order to execute many threads from a process at a time, and in multithreading, process creation is performed according to cost. Unlike multitasking, multithreading provides the same memory and resources to the processes for execution.

(مهم) Most modern operating systems are **multitasking**. This means that the operating system provides support for the apparent simultaneous execution of multiple programs. This is possible even on a system with a single core, since each process runs for a small interval of time (typically a few milliseconds), often called a **time slice**. After one running program has executed for a time slice, the operating system can run a different program. A multitasking OS may change the running process many times a minute, even though changing the running process can take a long time. In a multitasking OS if a process needs to wait for a resource for example, it needs to read data from external storage it will **block**. This means that it will stop executing and the operating system can run another process. However, many programs can continue to do useful work even though the part of the program that is currently executing must wait on a resource. For example, an airline reservation system that is blocked waiting for a seat map for one user could provide a list of available flights to another user.

Threading provides a mechanism for programmers to divide their programs into more or less independent tasks with the property that when one thread is blocked another thread can be run. in most systems it's possible to switch between threads much faster than it's possible to switch between processes . This is because threads are "lighter weight" than processes. Threads are contained within processes, so they can use the same executable, and they usually share the same memory and the same I/O devices.

We can say that two threads belonging to one process can share most of the process' resources.

The two most important exceptions are :(مهم)

- that they'll need a record of their own program counters

- and they’ll need their own call stacks

So that they can execute independently of each other If a process is the “master” thread of execution and threads are started and stopped by the process, then we can envision the process and its subsidiary threads as lines: when a thread is started, it **forks** off the process; when a thread terminates, it **joins** the process, **See Figure(2.2).**

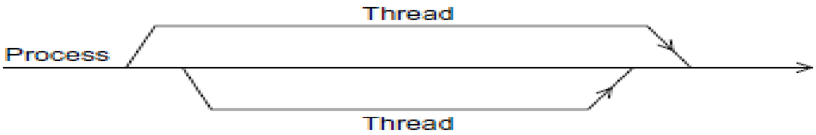


Figure (2.2). A Process and two threads.

After that, **the “join” part begins**, in which results of all subtasks are recursively joined into a single result, the program simply waits until every subtask is executed.

The main difference between process and thread are list in the table :

Basic for comparison	process	thread
Basic	Program in execution	Light weight process or part of it
Memory sharing	Completely isolated and do not share memory	Shares memory with each other
Resource conception	More	Less
efficiency	Less efficient	Enhanced efficiency
Time required for creation	More	Less
Context switching time	Takes more time	Consumes less time
Uncertain termination	Results in loss of process	A thread can be reclaimed
Time required for termination	More	Less

See figure 1 for more detail about the structure outline of the process and thread.

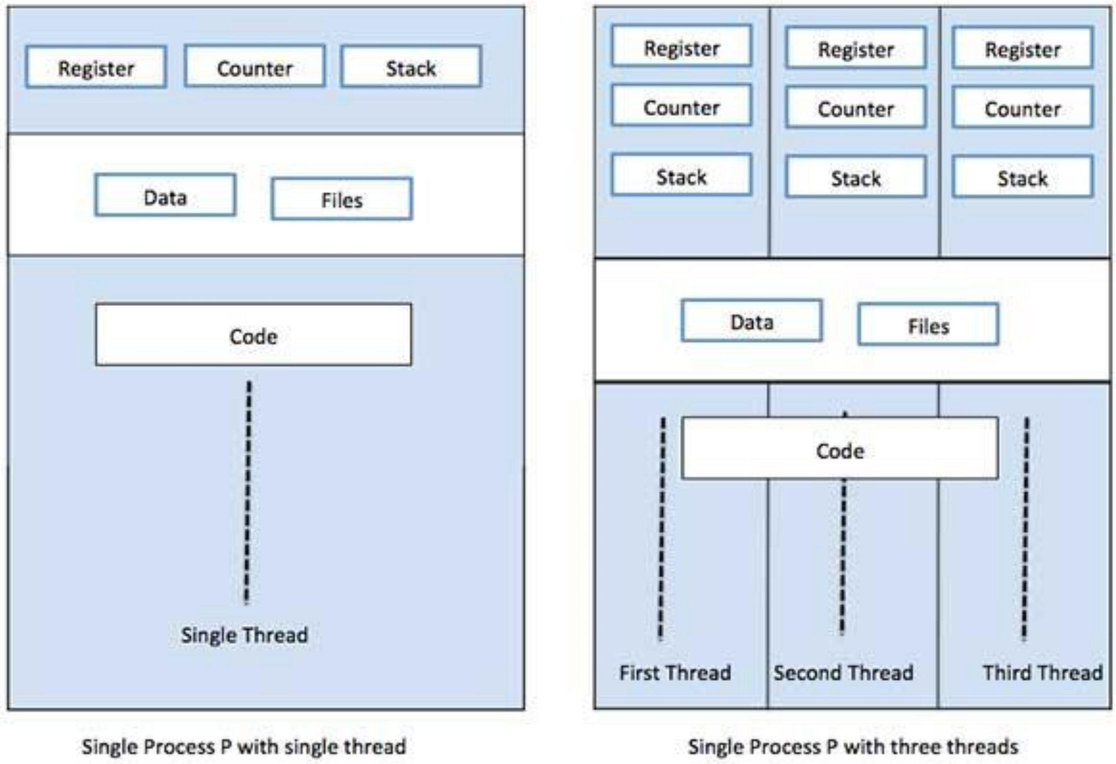


Figure 1 Process and thread structure.

Instruction-level Parallelism (ILP)

Instruction Level Parallelism (ILP) is used to refer to the *architecture* in which multiple operations can be performed parallelly in a particular process, with its own set of *resources* (address space, registers, identifiers, state, program counters). It refers to the compiler design techniques and processors designed to execute operations, like memory load and store, integer addition, float multiplication, in parallel to improve the performance of the processors.

Instruction-level parallelism (ILP) is a measure of how many of the instructions in a computer program can be executed simultaneously. ILP must not be confused with concurrency:

- ILP is the parallel execution of a sequence of instructions belonging to a specific thread of execution of a process (a running program with its set of resources: address space, a set of registers, its identifiers, its state, program counter (aka instruction pointer), and more).

- Concurrency involves the assignment of threads of one or different processes to a CPU's core in a strict alternation, or in true parallelism if there are enough CPU cores, ideally one core for each runnable thread.

There are two approaches to instruction level parallelism: Hardware and Software.

Hardware level works upon dynamic parallelism.
Software level works on static parallelism.

Dynamic parallelism means the processor decides at run time which instructions to execute in parallel, whereas static parallelism means the compiler decides which instructions to execute in parallel. The Pentium processor works on the dynamic sequence of parallel execution, but the Itanium processor works on the static level parallelism.

Architecture:

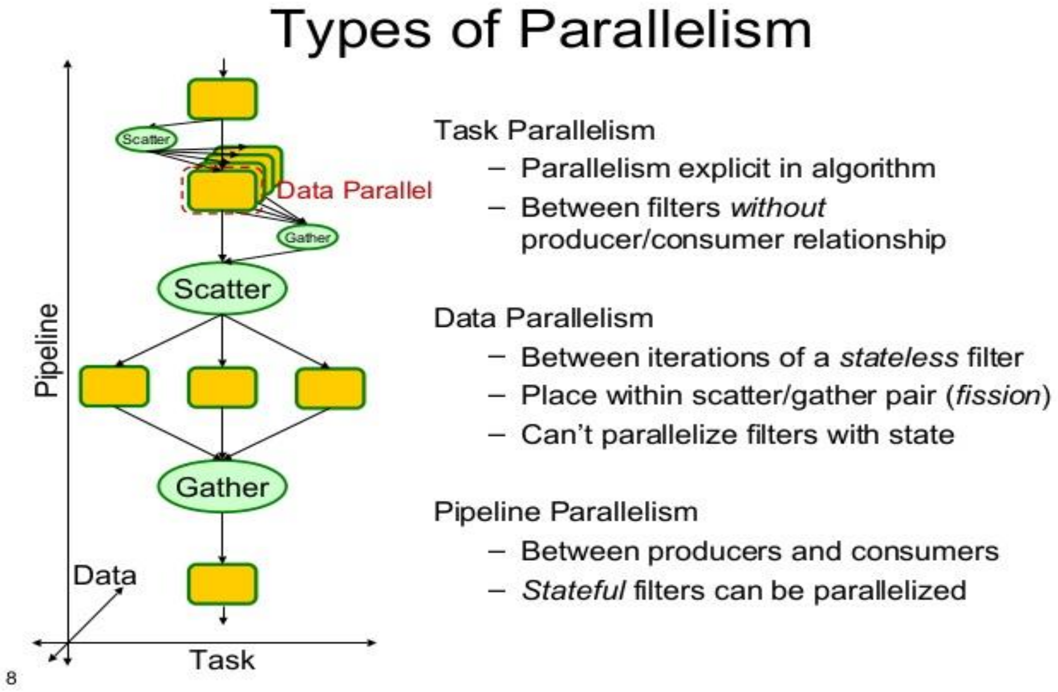
Instruction Level Parallelism is achieved when multiple operations are performed in single cycle, that is done by either executing them simultaneously or by utilizing gaps between two successive operations that is created due to the latencies.

Now, the decision of when to execute an operation depends largely on the compiler rather than hardware. However, extent of compiler's control depends on type of ILP architecture where information regarding parallelism given by compiler to hardware via program varies. The classification of ILP architectures can be done in the following ways :

- *Sequential Architecture:*
  program is not expected to explicitly convey any information regarding parallelism to hardware, like superscalar architecture.
- *Dependence Architectures:*
  program explicitly mentions information regarding dependencies between operations like dataflow architecture.
- *Independence Architecture:*

program gives information regarding which operations are independent of each other so that they can be executed .

In order to apply ILP, compiler and hardware must determine data dependencies, independent operations, and scheduling of these independent operations, assignment of functional unit, and register to store data.



Types of Parallelism

Task Parallelism
- Parallelism explicit in algorithm
- Between filters *without* producer/consumer relationship

Data Parallelism
- Between iterations of a *stateless* filter
- Place within scatter/gather pair (*fission*)
- Can't parallelize filters with state

Pipeline Parallelism
- Between producers and consumers
- *Stateful* filters can be parallelized

Figure (2.3) Types of parallelism.

Task Parallelism : This form of parallelism covers the execution of computer programs across multiple processors on same or multiple machines. It focuses on executing different operations in parallel to fully utilize the available computing resources in form of processors and memory.

One example of task parallelism would be an application creating threads for doing parallel processing where each thread is responsible for performing a different operation. Here is pseudo code illustrating task parallelism :

```
FOR each CPU in parallel computing environment
   Retrieve next task from task queue
   Create a thread and provide it with the retrieved task
   Start the created thread
END FOR
```

Some of Big Data frameworks that utilize task parallelism are Apache Storm and Apache YARN (it supports more of hybrid parallelism providing both task and data parallelism).

Data Parallelism : This form of parallelism focuses on distribution of data sets across the multiple computation programs. In this form, same operations are performed on different parallel computing processors on the distributed data sub set.

One example of data parallelism would be to divide the input data into sub sets and pass it to the threads performing same task on different CPUs. Here is the pseudo example illustrating data parallelism using a data array called :

```
lower_limit = 0
upper_limit = 0
FOR each CPU in parallel computing environment
   lower_limit = upper_limit + 1
   upper_limit = upper_limit + round(d.length/ no_of_cpus(
   Create a thread and provide it with lower_limit and upper_limit data array indexes
   Start the created thread
END FOR
```

Some of Big Data frameworks that utilize data parallelism are Apache Spark, Apache MapReduce and Apache YARN (it supports more of hybrid parallelism providing both task and data parallelism).

Parallel Hardware

Multiple issue and pipelining can clearly be considered to be parallel hardware, since functional units are replicated. Since this form of parallelism isn’t usually visible to the programmer, we’re treating both of them as extensions to the basic von Neumann model, and for our purposes, parallel hardware will be limited to hardware that’s visible to the programmer. In other words we’ll consider the hardware to be parallel if :

1- We can readily modify source code to exploit it.
2- We must modify source code to exploit it.

Flynn's taxonomy in general, digital computers may be classified into four categories, according to the multiplicity of instruction and data streams. This scheme for classifying computer organizations was introduced by Michael J. Flynn. The essential computing process is the execution of a sequence of instructions on a set of data. The term ***stream*** is used here to denote a sequence of items (instructions or data) as executed or operated upon by a single processor.

Instructions or data are defined with respect to a referenced machine. An ***instruction stream*** is a sequence of instructions as executed by the machine; a ***data stream*** is a sequence of data including input, partial, or temporary results, called for the instruction stream. Computer organizations are characterized by the multiplicity of the hardware provided to service the instruction and data streams. Listed below are Flynn’s four machine organizations:

- Single instruction stream single data stream (SISD)
- Single instruction stream multiple data stream (SIMD)
- Multiple instruction stream single data stream (MISD)
- Multiple instruction stream multiple data stream (MIMD)

2.4.1 SIMD systems

- In parallel computing, **Flynn’s taxonomy** is frequently used to classify computer architectures.
- It classifies a system according to the number of instruction streams and the number of data streams it can simultaneously manage.
- A classical von Neumann system is therefore a **single instruction stream, single data stream**, or **SISD** system, since it executes a single instruction at a time and it can fetch or store one item of data at a time
- **Single instruction, multiple data**, or **SIMD**, systems are parallel systems.
- As the name suggests, SIMD systems operate on multiple data streams by applying the same instruction to multiple data items, so an abstract SIMD system can be thought of as having a single control unit and multiple ALUs. An instruction is broadcast from the control unit to the ALUs, and each ALU either applies the instruction to the current data item, or it is idle.

As an example, suppose we want to carry out a “vector addition.” That is, suppose we have two arrays x and y, each with *n* elements, and we want to add the elements of y to the elements of x.

Suppose further that our SIMD system has *n* ALUs.

1- Then we could load x[i] and y[i] into the *i*th ALU, have the *i*th ALU add y[i] to x[i], and store the result in x[i].

2- If the system has *m* ALUs and *m* < *n*, we can simply execute the additions in blocks of *m* elements at a time. For example, if *m* D 4 and *n* D 15, we can first add elements 0 to 3, then elements 4 to 7, then elements 8 to 11, and finally elements 12 to 14. Note that in the last group of elements in our example elements 12 to 14 we're only operating on three elements of x and y, so one of the four ALUs will be idle.

Note in a "classical" SIMD system, the ALUs must operate synchronously, that is, each ALU must wait for the next instruction to be broadcast before proceeding.

- The ALUs have no instruction storage, so an ALU can't delay execution of an instruction by storing it for later execution.
- The first example shows, SIMD systems are ideal for parallelizing simple loops that operate on large arrays of data.
- Parallelism that's obtained by dividing data among the processors and having the processors all apply (more or less) the same instructions to their subsets of the data is called **data-parallelism**.
- SIMD parallelism can be very efficient on large data parallel problems, but SIMD systems often don't do very well on other types of parallel problems.
- The late 1990s the only widely produced SIMD systems were **vector processors**.
- More recently, graphics processing units, or GPUs, and desktop CPUs are making use of aspects of SIMD computing.

2.4.2 MIMD systems

- **MIMD systems multiple instruction, multiple data**, or MIMD, systems support multiple simultaneous instruction streams operating on multiple data streams.
- MIMD systems typically consist of a collection of fully independent processing units or cores, each of which has its own control unit and its own ALU.

- Unlike SIMD systems, MIMD systems are usually **asynchronous** the processors can operate at their own pace.
- In many MIMD systems there is no global clock, and there may be no relation between the system times on two different processors.

There are two principal types of MIMD systems:

1- Shared-memory systems
2- Distributed-memory system

Writing and Running Parallel Programs

In the past, virtually all parallel program development was done using a text editor such as vi or Emacs, and the program was either compiled and run from the command line or from within the editor. Debuggers were also typically started from the command line. Now there are also integrated development environments (IDEs) available from Microsoft, the Eclipse project, and others. We can use text editors or warped functions.

Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also . Before spending time in an attempt to develop a parallel solution for a problem :

1- determine whether or not the problem is one that can actually be parallelized.
2- Calculate the potential energy for each of several thousand independent conformations of a molecule.
3- When done, find the minimum energy conformation.

This problem is able to be solved in parallel if :

1- Each of the molecular conformations is independently determinable.
2- The calculation of the minimum energy conformation is also a parallelizable problem.

Lecture 3 Shared-Memory

Shared-Memory Programming with Pthreads

From a programmer’s point of view a shared-memory system is one in which all the cores can access all the memory locations (see Figure 4.1). Thus, an obvious approach to the problem of **coordinating** the work of the cores is to specify that certain memory locations are “**shared**.” we might well wonder why all parallel programs don’t use this shared-memory approach.

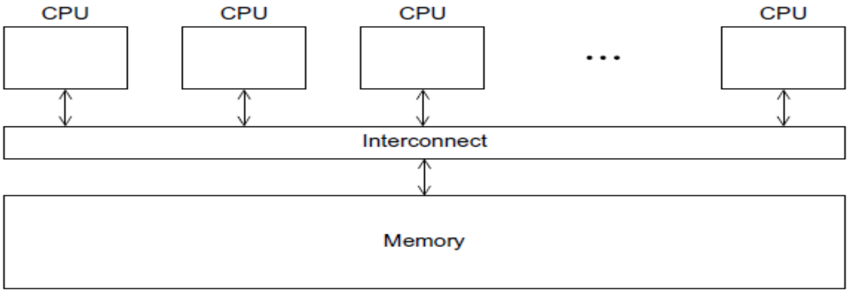


Figure 4.1 shared-memory system.

There are *problems in programming* shared-memory systems, problems that are often different from the problems encountered in distributed-memory programming. For example, we saw that if different cores attempt to update a single shared-memory location, then the contents of the shared location can be unpredictable. The code that updates the shared location is an example of a ***critical section***.

In shared-memory programming, an instance of a program running on a processor is usually called a ***thread*** *(unlike MPI, where it’s called a process). (مهم)*

Processes, Threads, and Pthreads

In shared-memory programming, a thread is somewhat analogous to a process in MPI programming, t’s a “**lighter-weight”.** A process is an instance of a running (or suspended) program. In addition to its executable, it consists of the following:

- A block of memory for the stack.

- A block of memory for the heap.
- Descriptors of resources that the system has allocated for the process for example, file descriptors.
- Security information for example, information about which hardware and software resources the process can access.
- Information about the state of the process, such as whether the process is ready to run or is waiting on a resource, the content of the registers including the program counter, and so on .

In most systems, by default, a process' memory blocks are private, another process can't directly access the memory of a process unless the operating system intervenes. One user's processes shouldn't be allowed access to the memory of another user's processes. However, this isn't what we want when we're running shared-memory programs. At a minimum, we'd like certain variables to be available to multiple processes, so shared-memory "processes" typically allow much easier access to each others' memory.

It's conceivable that they share pretty much everything that's process specific, except their stacks and their program counters this can be relatively easily arranged by starting a single process and then having the process start these "**lighter-weight**" processes. For this reason, they're often called **light-weight processes**. The more commonly used term, **thread**, comes from the concept of "**thread of control**." A thread of control is just a sequence of statements in a program. The term suggests a stream of control in a single process, and in a shared-memory program a single *process* may have multiple *threads* of control.

The expression fork-join in parallelism indicates a way to describe parallel performance of an application where the program stream splits (forks) into two or more threads capable of being executed simultaneously and then assemble (join) together, back into one flow after completing all the parallel work.

Why Multithreading? Threads are popular way to improve application through parallelism. For example, in a browser, multiple tabs can be different threads. MS word uses multiple threads, one thread to format the text, other thread to process inputs, etc.

Threads operate faster than processes due to following reasons:

1) Thread creation is much faster.

2) Context switching between threads is much faster.

3) Threads can be terminated easily

4) Communication between threads is faster.

HELLO, WORLD

Pthreads program which the main function starts up several threads. Each thread prints a message and then quits.

Execution

The program is compiled like an ordinary C program, with the possible exception that we may need to link in the **Pthreads** Library.

A Pthreads "hello, world" program

```
#include <pthread.h>

/* function to be run as a thread always must have the same signature:

   it has one void* parameter and returns void */

void *threadfunction(void *arg)

{  printf("Hello, World!\n"); }

int main(void)

{

  pthread_t thread;

  pthread_create(&thread, NULL, threadfunction, NULL);

  /*creates a new thread with default attributes and NULL passed as the argument to the start
routine*/

  pthread_join(thread, NULL); /*wait until the created thread terminates*/

  return 0;

    }

  printf("%s\n", strerror(createerror), stderr);

  return 1;
```

```
}
```

Preliminaries (وصف البرنامج السابق)

Let’s take a closer look at the source code in Program of hello world above .

- First notice that this *is* just a C program with a main function and one other function.
- In Line 1 we include **pthread.h,** the **Pthreads** header file, which declares the various Pthreads functions, constants, types, and so on.
- We define a ***global*** variable **thread** . In Pthreads programs, global variables are shared by all the threads.
- To create thread we use pthread_create(&thread, NULL, threadfunction, NULL);
- To finish thread work we use pthread_join(thread, NULL);

Local variables and function arguments that is, variables declared in functions are (ordinarily) private to the thread executing the function. If several threads are executing the same function, each thread will have its own private copies of the local variables and function arguments. This makes sense if you recall that each thread has its own stack.

Starting the Threads , Running the threads

In Pthreads the threads are started by the program executable. This introduces a bit of additional complexity, as we need to include code in our program to explicitly start the threads, and we need data structures to store information on the threads. we allocate storage for one **pthread_t** object for each thread. The **pthread_t** data structure is used for storing thread-specific information. It’s declared in **pthread.h**. The **pthread_t** objects are examples of **opaque** objects.

The actual data that they store is system specific, and their data members aren’t directly accessible to user code.

We use the pthread create function to start the threads. The syntax of **pthread_create** is:

pthread_create(&thread name, attr, start function, arg);

- The first argument is a pointer to the appropriate **pthread_t** object.
- We won’t be using the second argument, so we just pass the argument NULL in our function call.
- The third argument is the function that the thread is to run, and the last argument is a pointer to the argument that should be passed to the function **start_routine**.
- The return value for most Pthreads functions indicates if there’s been an error in the function call.

Recall that the type **void*** can be cast to any pointer type in C, so **args_p** can point to a list containing one or more values needed by **thread_function**. Similarly, the return value of **thread_function** can point to a list of one or more values.

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

no	Parameter	Description
1	**thread**	An opaque, unique identifier for the new thread returned by the subroutine.
2	**attr**	An opaque attribute object that may be used to set thread attributes. You can specify a thread attributes object, or NULL for the default values.
3	**start_routine**	The C++ routine that the thread will execute once it is created.
4	**arg**	A single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Start Thread

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h> //Header file for sleep().

#include <pthread.h>

// A normal C function that is executed as a thread

// when its name is specified in pthread_create()

void *myThreadFun(void *vargp)

{

        printf("Hi friends \n");

        return NULL;

}

int main()

{       pthread_t thread_id;

        printf("Before Thread\n");

        pthread_create(&thread_id, NULL, myThreadFun, NULL);

        pthread_join(thread_id, NULL);

        printf("After Thread\n");

        exit(0);

}
```

In main(), we declare a variable called thread_id, which is of type Pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread .pthread_create() takes 4 arguments .

The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join *blocks* the calling thread until the thread with identifier equal to the first argument terminates.

A C program to show multiple threads with global and static variables

```
#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

// Let us create a global variable to change it in threads

int g = 0;

// The function to be executed by all threads

void *myThreadFun(void *vargp)

{

        // Store the value argument passed to this thread

        int *myid = (int *)vargp;

        // Let us create a static variable to observe its changes

        static int s = 0;

        // Change static and global variables

        ++s; ++g;

        // Print the argument, static and global variables

        printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g); }

int main()

{       int i;

        pthread_t tid;

        // Let us create three threads

        for (i = 0; i < 3; i++)

                pthread_create(&tid, NULL, myThreadFun, (void *)&tid);

        pthread_exit(NULL);

        return 0;

}
```

Passing arguments to threads

Passing arguments to threads

```
#include <stdio.h>
#include <pthread.h>
void *thread_func(void *arg)
{
   printf("I am thread #%d\n", *(int *)arg);
   return NULL;
}
int main(int argc, char *argv[])
{
   pthread_t t1, t2;
   int i = 1;      int j = 2;
   pthread_create(&t1, NULL, &thread_func, &i);
   pthread_create(&t2, NULL, &thread_func, &j);
   /* This makes the main thread wait on the death of t1 and t2. */
   pthread_join(t1, NULL);
   pthread_join(t2, NULL);
   printf("In main thread\n");
   return 0; }
```

/* Create 2 threads t1 and t2 with default attributes which will execute function "thread_func()" in their own contexts with specified arguments. */

Stopping the Threads

We call the function **pthread_join** once for each thread. A single call to **pthread_join** will wait for the thread associated with the **pthread_t** object to complete.

pthread_join(thread, NULL);

The second argument can be used to receive any return value computed by the thread. There are following two routines which we can use to join or data threads . The default non-detached thread is allocated storage by the system that needs to be released on termination. To wait for a non-detached thread to terminate and reclaim the allocated storage and get the termination status, we use the following function:

pthread_join (id,status)		
no	**Parameter**	**Description**
1	id	Input that specifies the Thread Id
2	status	Input pointer to a pointer that on successful return will contain the termination status of the specified thread

Error checking

In the interest of keeping the program compact and easy to read, we have resisted the temptation to include many details that would therefore be important in a “real” program. The most likely source of problems in this example (and in many programs) is the user input or lack of it. It would therefore be a very good idea to check that the program was started with command line arguments, and, if it was, to check the actual systems value of the number of threads to see if it’s reasonable. It may also be a good idea to check the error codes returned by the Pthreads functions.

Other approaches to thread startup

The main thread then creates all of the “subsidiary” threads. While the threads are running, the main thread prints a message, and then waits for the other threads to terminate. This approach to threaded programming is very similar to our approach to MPI programming, in which the MPI system starts a collection of processes and waits for them to complete.

There is, however, a very different approach to the design of multithreaded programs. In this approach, subsidiary threads are only started as the need arises. Our main thread can start all the threads it anticipates needing at the beginning of the program . However, when a thread has no work, instead of terminating, it can sit idle until more work is available.

Mutex , READ-WRITE Locks

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a *race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.*

Mutex Locks: Theory:

- A ***mutex lock*** variable has 2 values (states)
    - ***Unlocked***
    - ***Locked***
- A ***mutex lock*** is a synchronization object with 2 operations
- ***Lock***
    - If the mutex lock is in the ***unlocked*** state, the ***lock*** will complete (and the thread continues with the next instruction following the ***lock*** command). The value (state) of the mutex lock is changed to ***locked***
    - If the mutex lock is in the ***locked*** state, the thread that executes the ***lock*** command will ***block*** (it stops execution) until the value (state) of the mutex lock becomes ***unlocked*** (When the state of the mutex lock does

become ***unlocked***, the ***lock*** command will complete and change the state of the mutex lock to ***locked***)

- ***Unlock***
  - If the mutex lock is in the ***locked*** state, the state is changed to ***unlocked***
  - If the mutex lock is in the ***unlocked*** state, this operation has no effect.

The mutex lock can ONLY be unlocked by the thread had previously locked the mutex.

```
pthread_mutex_t x;
```

Initializing a mutex variable:

After defining the mutex lock variable, you must initialized it using the following function:

```
int pthread_mutex_init(pthread_mutex_t    *mutex, pthread_mutexattr_t
*attr)
```

mutex: is the mutex lock that you want to initialize (pass the address) attr: is the set of initial property of the mutex lock.

The most common mutex lock is one where the lock is initially in the unlock. This kind of mutex lock is created using the (default) attribute null

Example: initialize a mutex variable:

```
pthread_mutex_t x;     /* Define a mutex lock "x/* "
pthread_mutex_init(&x, NULL);  /* Initialize "x" */
```

With locks

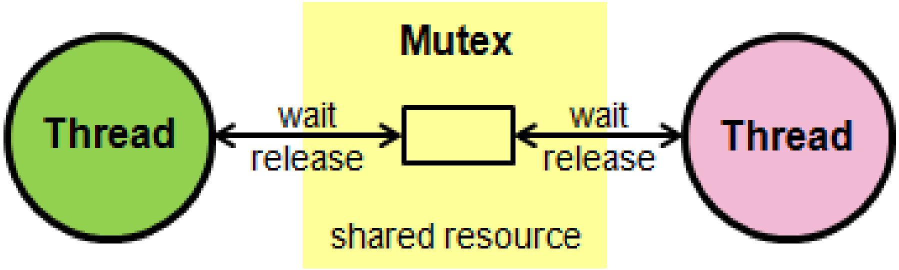
```
pthread_mutex_lock(&x);
```

Unlock

```
pthread_mutex_unlock(&x);
```

Mutex lock for Thread Synchronization

Thread synchronization is defined as a mechanism which ensures that two or more concurrent processes or threads do not simultaneously execute some particular program segment known as a critical section. Processes' access to critical section is controlled by using synchronization techniques. When one thread starts executing the critical section (a serialized segment of the program) the other thread should wait until the first thread finishes. If proper synchronization techniques are not applied, it may cause a race condition where the values of variables may be unpredictable and vary depending on the timings of context switches of the processes or threads.



Thread Synchronization Problems

An example code to study synchronization problems :

Thread Synchronization Problems

```
#include <pthread.h>

#include <stdio.h>

#include <stdlib.h>
```

```
#include <string.h>

#include <unistd.h>

  pthread_t tid[2];

int counter;

void* trythis(void* arg)

{

    unsigned long i = 0;

    counter += 1;

    printf("\n Job %d has started\n", counter)  ;

    for (i = 0; i < (0xFFFFFFFF); i++)        ;

    printf("\n Job %d has finished\n", counter);

    return NULL;

}

 int main(void)

{

    int i = 0;

    int error;

      while (i < 2) {

        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);

        if (error != 0)

            printf("\nThread can't be created : [%s]", strerror(error));

        i++;

    }

      pthread_join(tid[0], NULL);

    pthread_join(tid[1], NULL);

      return 0;}
```

How to compile above program?

In this example, two threads(jobs) are created and in the start function of these threads, a counter is maintained to get the logs about job number which is started and when it is completed.

Output :

```
Job 1 has started

Job 2 has started

Job 2 has finished

Job 2 has finished
```

Problem: From the last two logs, one can see that the log '*Job 2 has finished*' is repeated twice while no log for '*Job 1 has finished*' is seen.

Why it has occurred ?

On observing closely and visualizing the execution of the code, we can see that :

- The log '*Job 2 has started*' is printed just after '*Job 1 has Started*' so it can easily
- be concluded that while thread 1 was processing the scheduler scheduled the thread 2.
- If we take the above assumption true then the value of the '*counter*' variable got incremented again before job 1 got finished.
- So, when Job 1 actually got finished, then the wrong value of counter produced the log '*Job 2 has finished*' followed by the '*Job 2 has finished*' for the actual job 2 or vice versa as it is dependent on scheduler.
- So we see that its not the repetitive log but the wrong value of the 'counter' variable that is the problem.
- The actual problem was the usage of the variable 'counter' by a second thread when the first thread was using or about to use it.

- In other words, we can say that lack of synchronization between the threads while using the shared resource ‘counter’ caused the problems or in one word we can say that this problem happened due to ‘Synchronization problem’ between two threads.

How to solve it ?

The most popular way of achieving thread synchronization is by using **Mutexes**.

Mutex

- A Mutex is a lock that we set before using a shared resource and release after using it.
- When the lock is set, no other thread can access the locked region of code.
- So we see that even if thread 2 is scheduled while thread 1 was not done accessing the shared resource and the code is locked by thread 1 using mutexes then thread 2 cannot even access that region of code.
- So this ensures synchronized access of shared resources in the code.

Working of a mutex

1- Suppose one thread has locked a region of code using mutex and is executing that piece of code.

2- Now if scheduler decides to do a context switch, then all the other threads which are ready to execute the same region are unblocked

3- Only one of all the threads would make it to the execution but if this thread tries to execute the same region of code that is already locked then it will again go to sleep.

4- Context switch will take place again and again but no thread would be able to execute the locked region of code until the mutex lock over it is released.

5- Mutex lock will only be released by the thread who locked it

6- So this ensures that once a thread has locked a piece of code then no other thread can execute the same region until it is unlocked by the thread who locked it.

Hence, this system ensures synchronization among the threads while working on shared resources.

A mutex is initialized and then a lock is achieved by calling the following two functions : The first function initializes a mutex and through second function any critical region in the code can be locked.

1. **int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr) :** Creates a mutex, referenced by mutex, with attributes specified by attr. If attr is NULL, the default mutex attribute (NONRECURSIVE) is used.

   Returned value

   If successful, pthread_mutex_init() returns 0, and the state of the mutex becomes initialized and unlocked.

   If unsuccessful, pthread_mutex_init() returns -1.

2. **int pthread_mutex_lock(pthread_mutex_t *mutex) :** Locks a mutex object, which identifies a mutex. If the mutex is already locked by another thread, the thread waits for the mutex to

become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it. When the mutex has the attribute of recursive, the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then a count is incremented and no waiting thread is posted. The owning thread must call pthread_mutex_unlock() the same number of times to decrement the count to zero.

Returned value

If successful, pthread_mutex_lock() returns 0.

If unsuccessful, pthread_mutex_lock() returns -1.

The mutex can be unlocked and destroyed by calling following two functions : The first function releases the lock and the second function destroys the lock so that it cannot be used anywhere in future.

1. **int pthread_mutex_unlock(pthread_mutex_t *mutex) :** Releases a mutex object. If one or more threads are waiting to lock the mutex, pthread_mutex_unlock() causes one of those threads to return from pthread_mutex_lock() with the mutex object acquired. If no threads are waiting for the mutex, the mutex unlocks with no current owner. When the mutex has the attribute of recursive the use of the lock may be different. When this kind of mutex is locked multiple times by the same thread, then unlock will decrement the count and no waiting thread is posted to continue running with the lock. If the count is decremented to zero, then the mutex is released and if any thread is waiting for it is posted.

Returned value

If successful, pthread_mutex_unlock() returns 0.

If unsuccessful, pthread_mutex_unlock() returns -1

2. **int pthread_mutex_destroy(pthread_mutex_t *mutex) :** Deletes a mutex object, which identifies a mutex. Mutexes are used to protect shared resources. mutex is set to an invalid value, but can be reinitialized using pthread_mutex_init().

   Returned value

   If successful, pthread_mutex_destroy() returns 0.

   If unsuccessful, pthread_mutex_destroy() returns -1.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void* trythis(void* arg)
{
        pthread_mutex_lock(&lock);
        unsigned long i = 0;
        counter += 1;
        printf("\n Job %d has started\n", counter);

        for (i = 0; i < (0xFFFFFFFF); i++)     ;

        printf("\n Job %d has finished\n", counter);

        pthread_mutex_unlock(&lock);

        return NULL; }

int main(void)

{

        int i = 0;          int error;

        if (pthread_mutex_init(&lock, NULL) != 0) {

                printf("\n mutex init has failed\n");

                return 1;           }

        while (i < 2) {

                error = pthread_create(&(tid[i]),

                NULL,

                &trythis, NULL);

                if (error != 0)

        printf("\nThread can't be created :[%s]",

        strerror(error));   i++;  }

        pthread_join(tid[0], NULL);

        pthread_join(tid[1], NULL);

        pthread_mutex_destroy(&lock);

        return 0;   }
```

In the above code:

- A mutex is initialized in the beginning of the main function.
- The same mutex is locked in the ‘trythis()’ function while using the shared resource ‘counter.’
- At the end of the function ‘trythis()’ the same mutex is unlocked.
- At the end of the main function when both the threads are done, the mutex is destroyed.

Output

```
Job 1 started
Job 1 finished
Job 2 started
Job 2 finished
```

So this time the start and finish logs of both the jobs are present. So thread synchronization took place by the use of Mutex.

General notes

- A thread holds a mutex if it has successfully locked that mutex.
- Only one thread at any one time can hold a mutex.
- Mutexes are used to protect critical sections.
- ***Critical Sections*** are where threads are updating shared data.
- The updating thread should be forced to hold a mutex before updating.
- releasing (unlocking) the mutex when done.
- The mutex should have been locked by the thread unlocking it.
- If this is not the case, behaviour depends on what sort of mutex it is. Linux is non-standard here!

- Read `man pthread_mutex_unlock` for gory details, here.
- In the case of error checking this will result in an error.
- If other threads are waiting to lock this mutex, then one of them will subsequently succeed.
- Which one is entirely non-deterministic.

Mutex eaxample

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#define THREADS 5
static int sum = 1;
void *updater(void *ptr){
    sum = sum + 1;
    pthread_exit(NULL);
}
int main(void){
  int i;
  pthread_t threads[THREADS];
  for(i = 0; i < THREADS ; i++)
    pthread_create(&threads[i],NULL,updater,NULL);
  for(i = 0; i < THREADS ; i++)
    pthread_join(threads[i],NULL);
  fprintf(stderr, "sum = %d\n", sum);
  exit(EXIT_SUCCESS);
}
```

Notes

- Presumably it is designed so that after execution sum should be incremented THREADS times.
- Thus it should print out: `sum = 1 + THREADS`
- In reality this does seem to work ***almost*** all the time.
- However it is at the mercy of the scheduler.
- In a unfortunate world the answer could be any number greater than 1 and no bigger than 1 + THREADS.
- Pretty dangerous.

Another example

```
#include <pthread.h>

#include <unistd.h>

#define BUFSIZE 8

static int buffer[BUFSIZE];

static int bufin = 0;

static int bufout = 0;

static pthread_mutex_t

   buffer_lock = PTHREAD_MUTEX_INITIALIZER;

int get_buffersize(){

  return BUFSIZE;

}



void get_item(int *itemp){

   pthread_mutex_lock(&buffer_lock);

   *itemp = buffer[bufout];
   bufout = (bufout + 1) % BUFSIZE;
   pthread_mutex_unlock(&buffer_lock);
   return;
}
void put_item(int item){
   pthread_mutex_lock(&buffer_lock);
   buffer[bufin] = item;
   bufin = (bufin + 1) % BUFSIZE;
   pthread_mutex_unlock(&buffer_lock);
   return;
}
```

Time performance

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>
```

التعاريف الخاصة بلهيدرات

```
void *entry_point(void *value)

{

\\ Time for Thread Execution  Begin \\

    printf("hello from the second thread \n");

    int *num = (int *) value ;

    printf("the value of value is %d", *num");

    return NULL;

\\ Time for Thread Execution End \\

}
```

الدالة التي تنفذ الThread

نلاحظ ان وقت تنفيذ الدالة يتم بحساب وقت التنفيذ بين اقواس الدالة نفسها, بينما حساب وقت انهاء عمل ال Thread هو بوضع الوقت بين الانشاء والانهاء Create & Join .

نستطيع حساب الوقت الكلي بوضع داله الوقت ضمن قوس البداية والنهاية لدالة الMain.

أما وقت التنفيذ الخاص بالبرنامج من اول قراءة الهيدرات ولغاية قوس النهاية Execution Time نحصل عليه من شاشة التنفيذ الخاصة

```
int main(int argc, char **argv)

 {

\\ أضافة دالة الوقت من هنا ولغاية نهاية البرنامج يعطينا Overall Time \\

    pthread_t thread;  \\ Thread Starts\\

    printf("hello from the first thread \n");

    int num = 123; \\ Value to pass through the function \\

\\ Time here is the time for Thread creation to it Ends \\

    pthread_create(&thread, NULL, entry_point  , &num);   \\ Thread Create here \\

    pthread_join(thread, NULL);  \\ Thread Ends here or dead or finish it job \\

\\ Time here is the time for Thread creation to it Ends \\

   return EXIT_SUCCESS;

 \\ Time End here Over all Time \\

 }
```

Notes

pthread_create(&Thread Name, Attribute , Thread Function , &Argument pass to Function);

**\\ Thread Create here يتم انشاء الخيط هنا ضمن هذه الجملة **

pthread_join(Thread Name , Argument to pass from another thread);

**\\ Thread Ends here or dead or finish it job يتم انهاء عمل الخيط هنا باستخدام هذه الجملة **

ملاحظة : في حال تم طلب عمل خيطين بالبرنامج ومن ثم اخذ مخرجات الخيط الاول و تمريرها للخيط الثاني نستعمل دالة الجوين لعمل تمرير.

Linked list functions

The list itself is composed of a collection of list ***nodes***, each of which is a struct with two members: an **int** and a **pointer** to the next node.

A typical list is shown in **Figure 4.4**. A pointer, **head_p**, with type **struct list_node_s*** refers to the first node in the list. The **next** member of the last node is **NULL** (which is indicated by a slash (/) in the **next** member).

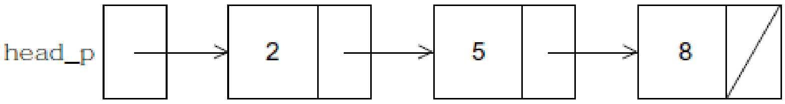


Figure 4.4 A linked list**.**

The Member function uses a pointer to traverse the list until it either finds the desired value or determines that the desired value cannot be in the list. Since the list is sorted, the latter condition occurs when the **curr_p** pointer is **NULL** or when the data member of the current node is larger than the desired value.

The **Insert** function begins by searching for the correct position in which to insert the new node. Since the list is sorted, it must search until it finds a node whose **data** member is greater than the **value** to be inserted. When it finds this node, it needs to insert the new node in the position *preceding* the node that’s been found. Since the list is singly-linked, we can’t “back up” to this position without traversing the list a second time. There are several approaches to dealing with this, the approach we use is to define a second pointer **pred_p**, which, in general, refers to the predecessor of the current node.

When we exit the loop that searches for the position to insert, the next member of the node referred to by **pred_p** can be updated so that it refers to the new node. See **Figure 4.5**.

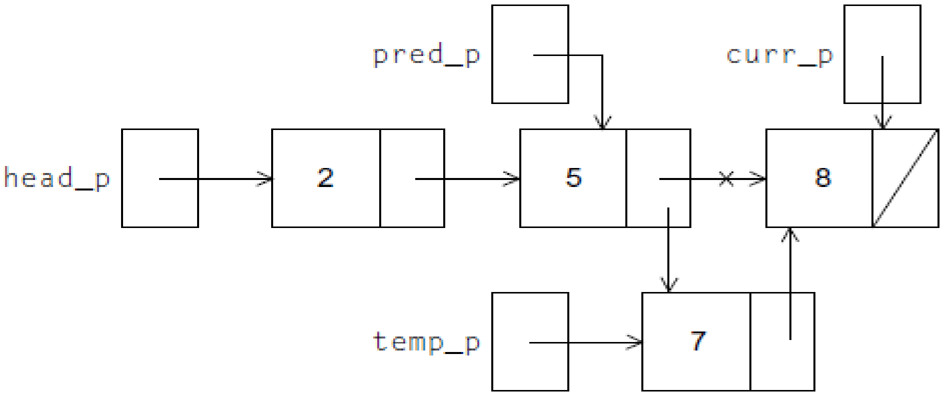


Figure 4.5 Inserting a new node into a.

The Delete function is similar to the **Insert** function in that it also needs to keep track of the predecessor of the current node while it’s searching for the node to be deleted. The predecessor node’s **next** member can then be updated after the search is completed. See **Figure 4.6.**

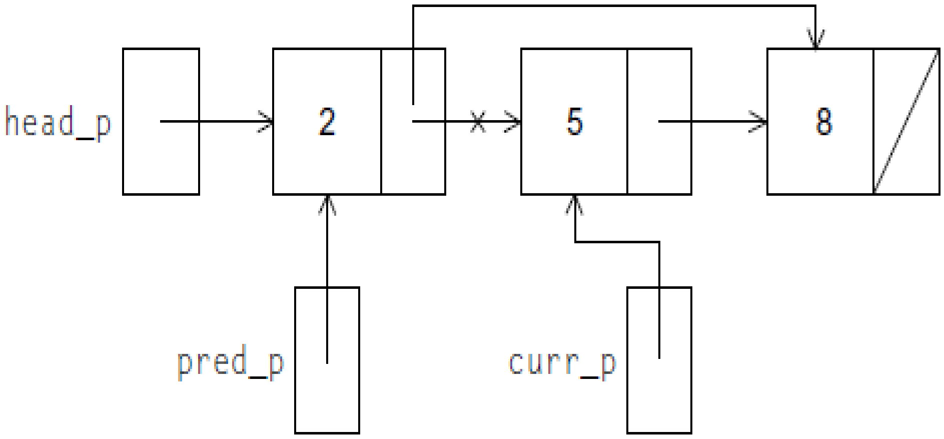


Figure 4.6 Deleting a node from the list.

A Multithreaded Linked List

Now let's try to use these functions in a Pthreads program. In order to share access to the list, we can define **head-p** to be a global variable. This will simplify the function headers for **Member, Insert, and Delete**, since we won't need to pass in either **head_p** or a pointer to **head_p**, we'll only need to pass in the value of interested .

Q: What now are the consequences of having multiple threads simultaneously execute the three functions?

Since multiple threads can simultaneously *read* a memory location without conflict, it should be clear that multiple threads can simultaneously execute **Member**. On the other hand, **Delete** and **Insert** also *write* to memory locations, so there may be problems if we try to execute either of these operations at the same time as another operation. As an example, suppose that thread 0 is executing **Member** (5) at the same time that thread 1 is executing **Delete** (5). The current state of the list is shown in **Figure 4.7**. An obvious problem is that if thread 0 is executing **Member** (5), it is going to report that 5 is in the list, when, in fact, it may be deleted even before thread 0 returns.

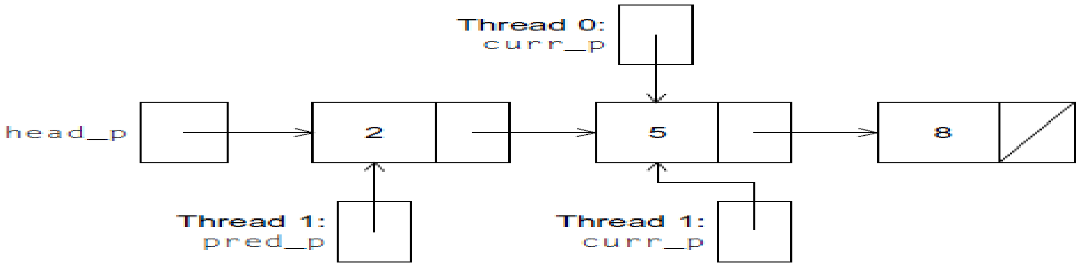


Figure 4.7 Simultaneous access by two threads**.**

A second obvious problem is if thread 0 is executing **Member** (8), thread 1may free the memory used for the node storing 5 before thread 0 can advance to the node storing 8. Although typical implementations of **free** don't overwrite the freed memory, if the memory is reallocated before thread 0 advances, there can be serious problems. For example, if the memory is reallocated for use in something other than a list node, what thread 0 "thinks" is the **next** member may be set to utter garbage,
and after it executes.

Dereferencing **curr_p** may result in a segmentation violation. More generally, we can run into problems if we try to simultaneously execute another operation while we're executing an **Insert** or a **Delete**. It's OK for multiple threads to simultaneously execute **Member** that is, *read* the list nodes but it's unsafe for multiple threads to access the list if at least one of the threads is executing an **Insert** or a **Delete** that is, is *writing* to the list nodes. How can we deal with this problem? An obvious solution is to simply lock the list any time that a thread attempts to access it. For example, a call to each of the three functions can be protected by a mutex, so we might execu**te.**

Instead of simply calling **Member**(value). An equally obvious problem with this solution is that we are serializing access to the list, and if the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism. On the other hand, if most of our operations are calls to Insert and **Delete**, then this may be the best solution, since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement**.**

An alternative to this approach involves "finer-grained" locking. Instead of locking the entire list, we could try to lock individual nodes. We would add, for example, a mutex to the list node struct**.**

Now each time we try to access a node we must first lock the mutex associated with the node. Note that this will also require that we have a mutex associated with the head p pointer. So, for example, we might implement Member as shown in Program below. Admittedly this implementation is *much* more complex than the original Member function. It is also much slower, since, in general, each time a node is accessed, a mutex must be locked and unlocked. At a minimum it will add two function calls to the node access, but it can also add a substantial delay if a thread **has.**

To wait for a lock. A further problem is that the addition of a mutex field to each node will substantially increase the amount of storage needed for the list. On the other hand, the finer-grained locking might be a closer approximation to what we want. Since we're only locking the nodes of current interest, multiple threads can simultaneously access different parts of the list, regardless of which operations they're executin**g.**

Pthreads read-write locks

Neither of our multithreaded linked lists exploits the potential for simultaneous access to *any* node by threads that are executing Member. The first solution only allows one thread to access the entire list at any instant, and the second only allows one thread to access any given node at any instant. An alternative is provided by Pthreads' **read-write locks**. A read-write lock is **somewhat like a mutex except that it provides two lock functions.** The first lock function locks the read-write lock for *reading*, while the second locks it for *writing.* Multiple threads can thereby simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function. Thus, if any threads own the lock for reading, any threads that want to obtain the lock for writing will block in the call to the write-lock function. Furthermore, if any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Performance of the Various Implementations

We really want to know which of the three implementations is "best," so we included our implementations in a small program in which the main thread first inserts a user-specified number of randomly generated keys into an empty list. After being started by the main thread, each thread carries out a user-specified number of operations on the list. The user also specifies the percentages of each type of operation (**Member, Insert, Delete**).

Implementing read-write locks

The original Pthreads specification didn't include read-write locks, so some of the early texts describing Pthreads include implementations of read-write locks. A typical implementation6 defines a data structure that uses two condition variables one for "readers" and one for "writers" and a mutex. The structure also contains members that indicate**:**

1. how many readers own the lock, that is, are currently reading,
2. how many readers are waiting to obtain the lock,
3. whether a writer owns the lock, and

4. how many writers are waiting to obtain the lock.

The mutex protects the read-write lock data structure, whenever a thread calls one of the functions (read-lock, write-lock, unlock), it first locks the mutex, and whenever a thread completes one of these calls, it unlocks the mutex. After acquiring the mutex, the thread checks the appropriate data members to determine how to proceed. As an example, if it wants read-access, it can check to see if there’s a writer that currently owns the lock. If not, it increments the number of active readers and proceeds.

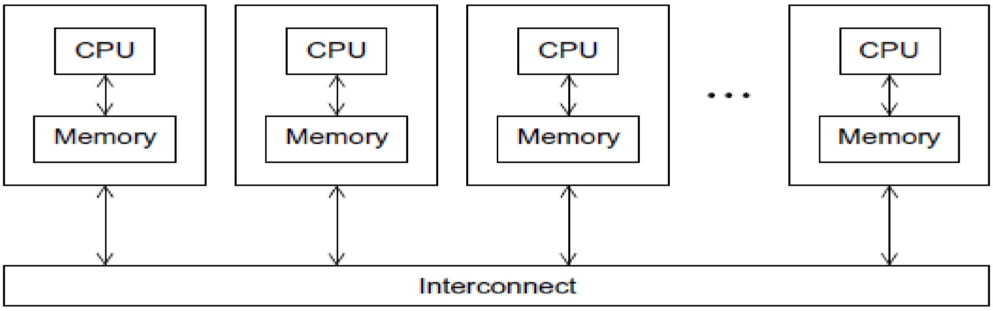
Lecture 4 Distributed-Memory

Distributed-Memory Programming with MPI

The world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. See **Figure 3.1.**

On the other hand, from a programmer's point of view, a shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location. See **Figure 3.2**. Distributed-memory systems using **message-passing** , in message-passing programs, a program running on one core-memory pair is usually called a **process**, and two processes can communicate by calling functions, one process calls a *send* function and the other calls a *receive* function.

The implementation of message-passing that we'll be using is called MPI, which is an abbreviation of Message-Passing Interface.



Figure(3.1) distributed-memory system.

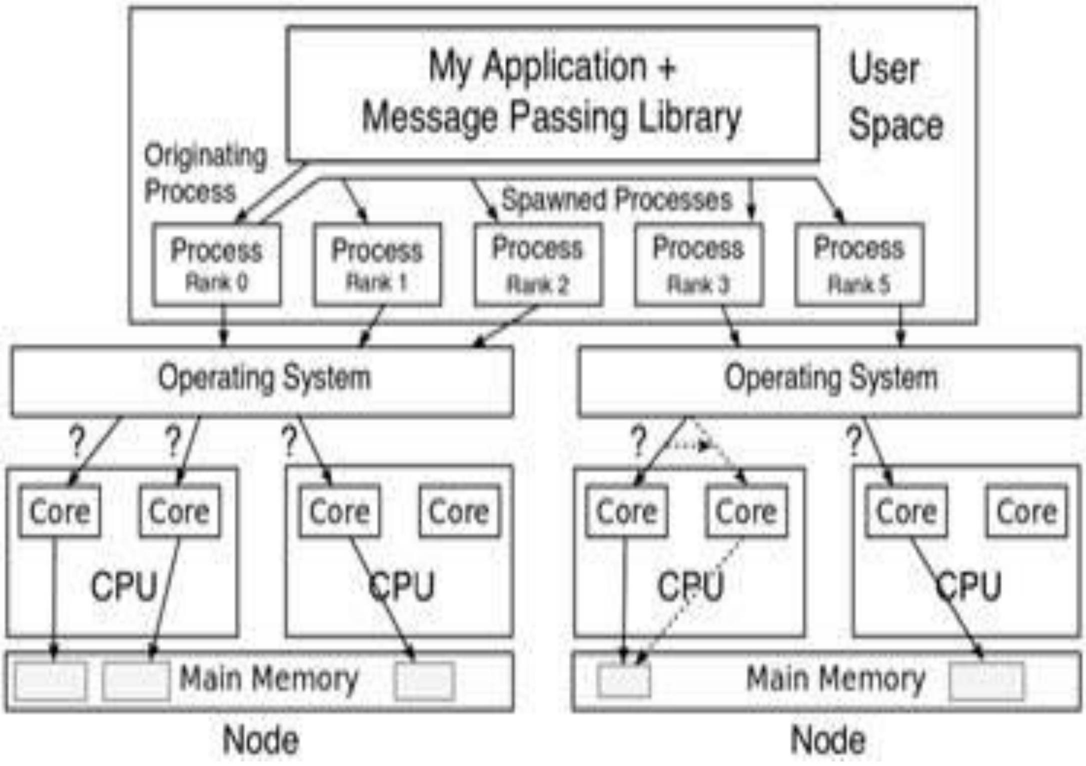
Getting Started

Perhaps the first program that many of us saw was some variant of the "hello, world" program:

Hello Program

```
#include <stdio.h>

int main(void)

{ printf("hello, world ");

return 0; }
```

In parallel programming, its common (one might say standard) for the processes to be identified by nonnegative integer ***ranks***. So if there are *p* processes, the processes will have ranks 0, 1, 2, *p*-1. For our parallel "hello, world," let's make process 0 the designated process and the other processes will send it messages.



Compilation and Execution

Some times we use a **wrapper** for the C compiler. A **wrapper function** is a subroutine in a software library or a computer program whose main purpose is to call a second subroutine or a system call with little or no additional computation. Wrapper functions are used to make writing computer programs easier by abstracting away the details of a subroutine's underlying implementation.

A **wrapper script** is a script whose main purpose is to run some program. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files and which libraries to link with the object file.

MPI program that prints greetings from the processes

```
#include <stdio.h>
#include <string.h> /∗ For strlen ∗/
#include <mpi.h> /∗ For MPI functions, etc ∗/

const int MAX_STRING = 100;

int main(void) {
char greeting[MAX_STRING];
int comm_sz; /∗ Number of processes ∗/
int my_rank; /∗ My process rank ∗/
MPI_Init(NULL, NULL);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
if (my_rank != 0) {

sprintf(greeting, "Greetings from process %d of %d!", my_rank, comm_sz);
19 MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
MPI_COMM_WORLD); }
21  else {

22 printf("Greetings from process %d  of  %d !\n", my_rank, comm_sz);
23   for (int q = 1; q < comm_sz; q++) {
24   MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0,
MPI_COMM_WORLD,    MPI_STATUS_IGNORE);

26     printf("%s\n", greeting);  }   }
30  MPI_Finalize();
31  return 0;    } /* main */
```

MPI Programs

- The first thing to observe is that this is a **C program**. For example, it includes the standard C header files **stdio.h** and **string.h.**
- It has a main function just like any other C program.
- There are many parts of the program which are new. Line 3 includes the **mpi.h** header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.
- All of the identifiers defined by MPI start with the string **MPI** the first letter following the underscore is capitalized for function names and **MPI-defined types**.
- All of the letters in MPI-defined macros and constants are capitalized, so there’s no question about what is defined by MPI and what’s defined by the user program.

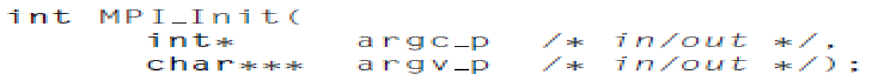
MPI Init and MPI Finalize

The call to MPI Init tells the MPI system to do all of the necessary setup.

- It allocate storage for message buffers
- it might decide which process gets which rank.
- It define global communicators.

As a rule of thumb, no other MPI functions should be called before the program calls MPI_Init. Its syntax is :

MPI Initial code



```
int MPI_Init(
      int*      argc_p  /* in/out */,
      char***   argv_p  /* in/out */);
```

- The arguments**, argc_p** and **argv_p**, are pointers to the arguments to main, argc, and argv. When our program doesn't use these arguments, we can just pass NULL for both.
- Like most MPI functions, **MPI_Init** returns an ***<u>int</u>*** *<u>error code</u>*, and in most cases we'll ignore these error codes.

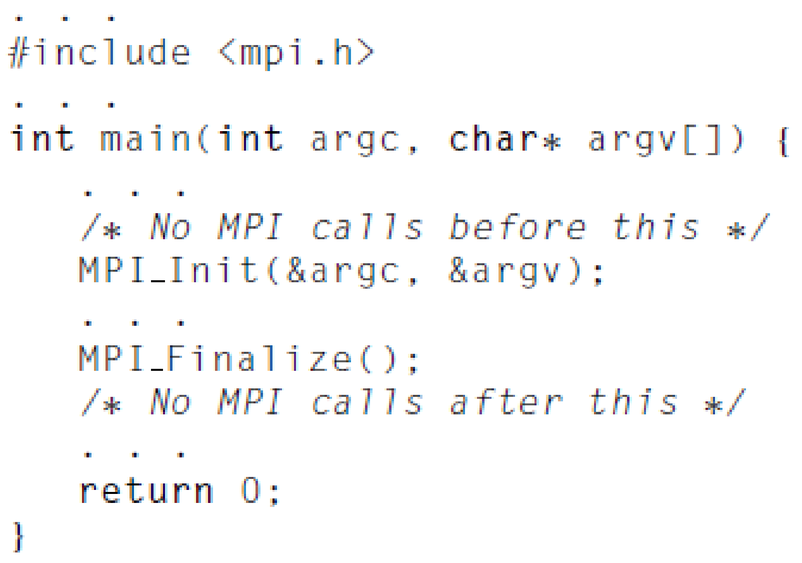
The call to MPI Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

MPI Finalize
int MPI_Finalize(**void**);

In general, *no MPI functions should be called after the call to MPI Finalize.*

A typical MPI program has the following basic outline:

MPI program basic outline



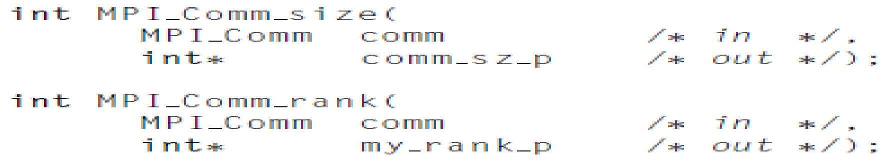
```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

It's not necessary to pass pointers to *argc and argv* to **MPI_Init**. It's also not necessary that the calls to **MPI Init** and **MPI Finalize** be in main.

Communicators, MPI_Comm_size and MPI _Comm_rank

In MPI a ***communicator*** is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when it started the program. This communicator is called **MPI_COMM_WORLD**, their syntax is:

MPI_COMM_WORLD



```
int MPI_Comm_size(
      MPI_Comm    comm         /* in   */,
      int*        comm_sz_p    /* out  */);

int MPI_Comm_rank(
      MPI_Comm    comm         /* in   */,
      int*        my_rank_p    /* out  */);
```

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators,

- ***MPI_Comm_size*** returns in its second argument the number of processes in the communicator, and ***MPI_Comm_rank*** returns in its second argument the calling process' rank in the communicator.
- We'll often use the variable **comm_sz** for the number of processes in **MPI_COMM_WORLD**, and the variable **my_rank** for the process rank**.**

Notice that we compiled a single program we didn't compile a different program for each process and we did this in spite of the fact that process 0 is doing something fundamentally different from the other processes:

- it's receiving a series of messages and printing them, while each of the other processes is creating and sending a message.
- This is quite common in parallel programming. In fact, *most* MPI programs are written in this way.
- That is, a single program is written so that different processes carry out different actions, and this is achieved by simply having the processes branch on the basis of their process rank.
- Recall that this approach to parallel programming is called single program, multiple data, or SPMD.
- The **if_else** statement in Lines 16 through 28 makes our program SPMD.

Communication

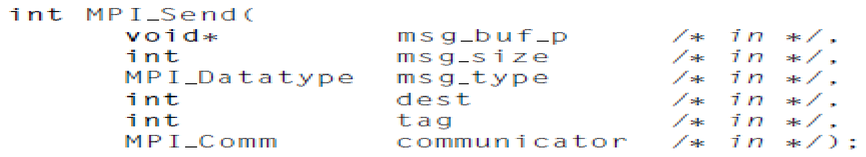
In Lines 17 and 18, each process, other than process 0, creates a message it will send to process 0. (The function sprintf is very similar to printf, except that instead of writing to stdout, it writes to a string.) Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using printf, and then uses a **for** loop to receive and print the messages sent by processes 1, 2, … ,comm_sz_1. Lines 24–25 receive the message sent by process *q*, for *q* = 1, 2, … ,comm_sz_1.

MPI Send

Each of the sends is carried out by a call to MPI_Send, whose syntax is:

MPI_Send(msg_buf_p, msg_size, msg_type, dest, tag, communicator);

MPI Send



```
int MPI_Send(
        void*           msg_buf_p       /* in */,
        int             msg_size        /* in */,
        MPI_Datatype    msg_type        /* in */,
        int             dest            /* in */,
        int             tag             /* in */,
        MPI_Comm        communicator    /* in */);
```

- The first three arguments, **msg_buf_p, msg_size**, and **msg_type,** determine the contents of the message.
- The remaining arguments, **dest, tag**, and **communicator**, determine the destination of the message.
- The first argument, **msg_buf_p**, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, **greeting**. (Remember that in C an array, such as a string, is a pointer).
- The second and third arguments, **msg_size** and **msg_type**, determine the amount of data to be sent.
- In our program, the **msg_size** argument is the number of characters in the message plus one character for the '\0' character that terminates C strings. The **msg_type** argument is **MPI_CHAR**. These two arguments together tell the system that the message contains **strlen(greeting)+1 char**s. Since C types (**int**, **char**, and so on.) can't be passed as arguments to functions, MPI defines a special

type, **MPI_Datatype**, that is used for the **msg_type** argument. MPI also defines a number of constant values for this type.

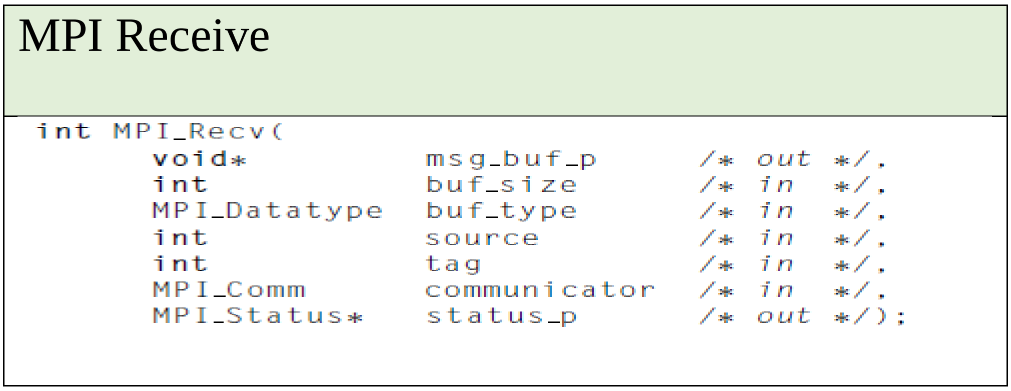
- Notice that the size of the string **greeting** is not the same as the size of the message specified by the arguments **msg_size** and **msg_type**.
- For example, when we run the program with four processes, the length of each of the messages is 31 characters**,** while we've allocated storage for 100 characters in greetings.

Of course, the size of the message sent should be less than or equal to the amount of storage in the buffer in our case the string greeting. The fourth argument, **dest**, specifies the rank of the process that should receive the message. The fifth argument, **tag**, is a nonnegative **int**. It can be used to distinguish messages that are otherwise identical. For example, suppose process 1 is sending floats to process 0. Some of the floats should be printed, while others should be used in a computation. Then the first four arguments to **MPI_Send** provide no information regarding which floats should be printed and which should be used in a computation. So process 1 can use, say, a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation. The final argument to **MPI_Send** is a communicator. All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that's using a different communicator. Since MPI provides functions for creating new communicators, this feature can be used in complex programs to insure that messages aren't "accidentally received" in the wrong place.

MPI Recv

The first six arguments to **MPI_Recv** correspond to the first six arguments of **MPI_Send:**

MPI_Recv(msg_buf_p, buf_size, buf_type, source, tag, communicator, status_p);



MPI Receive

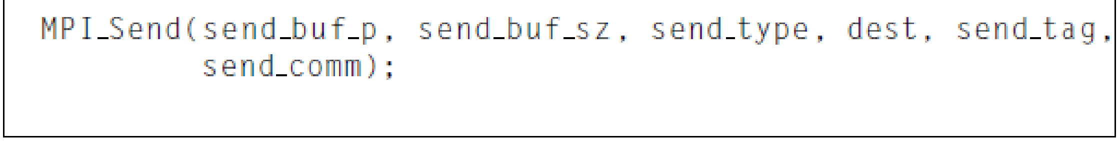
```
int MPI_Recv(
        void*          msg_buf_p      /* out */,
        int            buf_size       /* in  */,
        MPI_Datatype   buf_type       /* in  */,
        int            source         /* in  */,
        int            tag            /* in  */,
        MPI_Comm       communicator   /* in  */,
        MPI_Status*    status_p       /* out */);
```

- The first three arguments specify the memory available for receiving the message: **msg_buf_p** points to the block of memory, **buf_size** determines the number of objects that can be stored in the block, and **buf_type** indicates the type of the objects.
- The next three arguments identify the message.
- The source argument specifies the process from which the message should be received.
- The **tag** argument should match the **tag** argument of the message being sent, and the **communicator** argument must match the communicator used by the sending process.
- We'll talk about the status p argument shortly.
- In many cases it won't be used by the calling function, and, as in our **"greetings"** program, the special MPI constant **MPI_STATUS_IGNORE** can be passed.

Message Matching

Suppose process *q* calls **MPI_Send** with

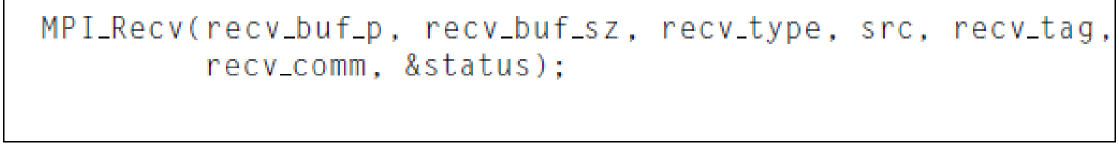
Message Matching



```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
         send_comm);
```

Also suppose that process *r* calls **MPI_Recv** with:

Message Matching



```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
         recv_comm, &status);
```

Then the message sent by *q* with the above call to **MPI_Send** can be received by *r* with the call to **MPI_Recv** if:

- Recv_comm = send_comm,
- Recv_tag = send_tag,
- dest = r
- src = q.

Q: Write an MPI program that prints hello world from processor rank of size **.**

MPI Hello world

```
#include <mpi.h>

#include <stdio.h>

int main(int argc, char** argv) {    // Initialize the MPI environment

    MPI_Init(NULL, NULL);        // Get the number of processes

    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    // Get the rank of the process

    int world_rank;

    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

    // Get the name of the processor

    char processor_name[MPI_MAX_PROCESSOR_NAME];

    int name_len;

    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message

    printf("Hello world from processor %s, rank %d out of %d processors\n",

    processor_name, world_rank, world_size);

    // Finalize the MPI environment.

    MPI_Finalize( ); }
```

Performance Evaluation of MPI Programs

For the most part we write parallel programs because we expect that they’ll be faster than a serial program that solves the same problem.

Taking timings

We’re usually not interested in the time taken from the start of program execution to the end of program execution. For example, in the matrix-vector multiplication, we’re not interested in the time it takes to type in the matrix or print out the product We’re only interested in the time it takes to do the actual multiplication, so we need to modify our source code by adding in calls to a **function that will tell us the amount of time that**

elapses from the beginning to the end of the actual matrix-vector multiplication. MPI provides a function, **MPI_Wtime**, that returns the number of seconds that have elapsed since some time in the past:

```
double MPI Wtime(void);
```

Thus, we can time a block of MPI code as follows:

MPI Time

```
double start, finish;
.. .
start = MPI_Wtime( );
/* Code to be timed */
. . .
finish = MPI_Wtime( );
printf("Proc %d > Elapsed time = %e seconds\n" my_rank, finish-start);
```

When we run a program several times, we’re likely to see a substantial variation in the times. This will be true even if for each run we use the same input, the same number of processes, and the same system. This is because the **interaction** of the program with the rest of the system, especially the operating system, is unpredictable. Since this interaction will almost certainly not make the program run faster than it would run on a “quiet” system, we usually report the *minimum* run-time rather than the mean or median.

When we run an MPI program on a hybrid system in which the nodes are multicore processors, we’ll only run **one MPI process** on each node. This may reduce contention for the interconnect and result in somewhat better run-times, It may also reduce variability in run-times.

Results

The parallel program will divide the work of the serial program among the processes, and add in some overhead time, in MPI programs, the parallel overhead typically comes from communication, and it can depend on both the problem size and the number of p

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

Speedup and efficiency

The goal is to equally distribute the workload among all the processors, whereas resulting in no extra load on the cores. If this goal is reached, and the program runs with a P number of cores, one thread or process on each core, then the parallel application will be executed P times faster than the sequential application. If sequential execution time is named **T_{serial}** and the parallel execution time called **T_{parallel}**, then the ultimate case of the resulting parallel time is calculated from:

$$\mathbf{T_{parallel} = T_{serial} / P \; \ldots\ldots \; (2.1)}$$

If this happens, then this parallel program has **linear speedup**. Practically, this case is unlikely to happen because the exploitation of a number of processes or threads usually introduces some inevitable overhead.

Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It’s just the ratio of the serial run-time to the parallel run-time:

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}.$$

The ideal value for ***S*(*n*,*p*)** is ***p***. If ***S*(*n*,*p*) = *p***, then our parallel program with comm_sz = *p* processes is running *p* times faster than the serial program. This speedup, sometimes called **linear speedup**, is rarely achieved. For small *p* and large *n*, our program obtained nearly linear speedup ,on the other hand, for large *p* and small *n*, the speedup was considerably less than *p*. The worst case was *n* = 1024 and *p* = 16, when we only managed a speedup of 2.4.

$$\mathbf{E= S/P = (T_{serial} / T_{parallel})/P = T_{serial} / (T_{parallel} *P)}$$

We also recall that another widely used measure of parallel performance is parallel **efficiency**. This is “per process” speedup:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}.$$

Linear speedup corresponds to a parallel efficiency of *p*/*p* = 1.0, and, in general, we expect that our efficiencies will be less than 1.

Amdahl’s Law calculates the speedup of parallel code based on three variables:

■ Duration of running the application on a single-core machine

■ The percentage of the application that is parallel

■ The number of processor cores

Here is the formula, which returns the ratio of single-core versus multicore performance.

$$\text{Speedup} = \frac{1}{1 - P + (P/N)}$$

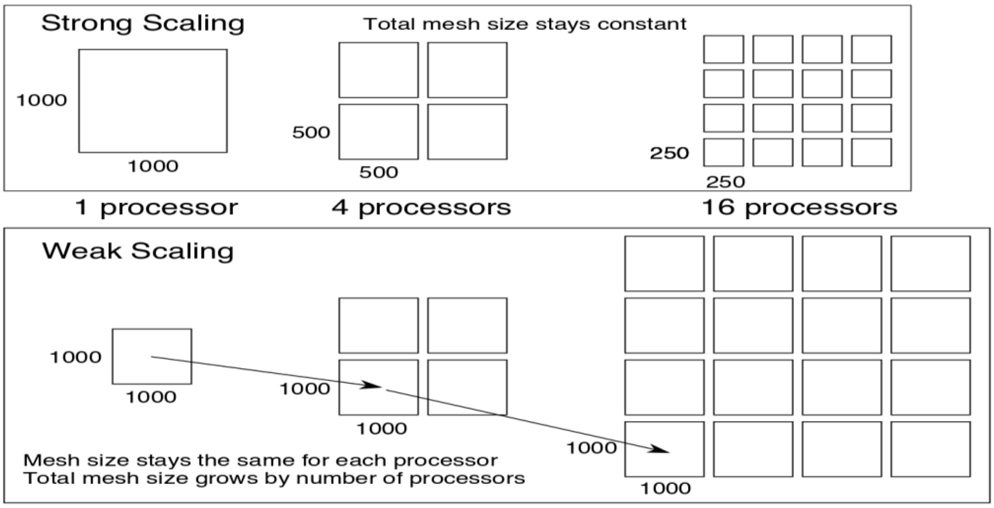
As an example scenario, suppose you have an application that is 75 percent parallel and runs on a machine with three processor cores. The first iteration to calculate Amdahl’s Law is shown below. In the formula, *P* is *.75* (the parallel portion) and *N* is *3* (the number of cores).

Scalability

Our parallel program doesn't come close to obtaining linear speedup for small *n* and large *p*. Does this mean that it's not a good program? Many computer scientists answer this question by looking at the "scalability" of the program.

A program is scalable if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase. Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.

The term scalability is often used to refer to whether more parallelism can be added in either the hardware or software and whether there is an overall limit to how much improvement can occur. While the traditional focus has been on the run-time scaling, we will make the argument that memory scaling is often more important.


Strong Scaling
Total mesh size stays constant
1000
1000
500
500
250
250
1 processor
4 processors
16 processors
Weak Scaling
1000
1000
1000
1000
1000
1000
Mesh size stays the same for each processor
Total mesh size grows by number of processors

Lecture 5 Development

Parallel Program Development

In parallel programming there are problems that we need to solve for which there is no serial analog. We’ll see that there are instances in which, as parallel programmers, we’ll have to start “from scratch.

Tree Search

Many problems can be solved using a tree search. As a simple example, consider the traveling salesperson problem, or TSP. In TSP, a salesperson is given a list of cities she needs to visit and a cost for traveling between each pair of cities. Her problem is to visit each city once, returning to her hometown, and she must do this with the least possible cost. A route that starts in her hometown, visits each city once and returns to her hometown is called a *tour*; thus, the TSP is to find a minimum-cost tour. Unfortunately, TSP is what’s known as an **NP-complete** problem. From a practical standpoint, this means that there is no algorithm known for solving it that, in all cases, is significantly better than exhaustive search. Exhaustive search means examining all possible solutions to the problem and choosing the best. The number of possible solutions to TSP grows exponentially as the number of cities is increased.

Program : Create Binary search tree using Thread programming

وصف البرنامج :

انشاء شجرة ضمن برنامج الmain واستدعاء دالة insert لاضافة ابن , يهدف البرنامج للاطلاع على عمل tree
من خلال استعمال البرمجة المتوازية.

واجب المختبر :

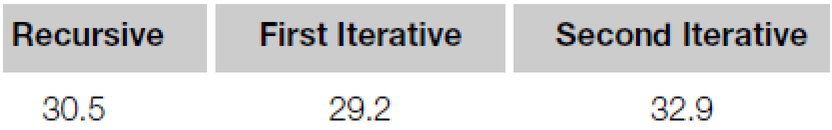
تنفيذ البرنامج والاطلاع على كيفية عمل الدوال

```
// Insertion in Threaded Binary Search Tree.
#include<bits/stdc++.h>
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;

    // True if left pointer points to predecessor
    // in Inorder Traversal
    bool lthread;

    // True if right pointer points to successor
    // in Inorder Traversal
    bool rthread;
};

// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
    // Searching for a Node with given value
    Node *ptr = root;
    Node *par = NULL; // Parent of key to be inserted
    while (ptr != NULL)
    {
        // If key already exists, return
        if (ikey == (ptr->info))
        {
            printf("Duplicate Key !\n");
            return root;
        }

        par = ptr; // Update parent pointer

        // Moving on left subtree.
        if (ikey < ptr->info)
        {
            if (ptr -> lthread == false)
                ptr = ptr -> left;
            else
                break;
        }

        // Moving on right subtree.
        else
        {
            if (ptr->rthread == false)
                ptr = ptr -> right;
            else
                break;
        }
    }

    // Create a new node
    Node *tmp = new Node;
    tmp -> info = ikey;
    tmp -> lthread = true;
    tmp -> rthread = true;
    if (par == NULL)
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))
    {
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
    }
    else
    {
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
    }

    return root;
}

// Returns inorder successor using rthread
struct Node *inorderSuccessor(struct Node *ptr)
{
    // If rthread is set, we can quickly find
    if (ptr -> rthread == true)
        return ptr->right;

    // Else return leftmost child of right subtree
    ptr = ptr -> right;
    while (ptr -> lthread == false)
        ptr = ptr -> left;
    return ptr;
}

// Printing the threaded tree
void inorder(struct Node *root)
{
    if (root == NULL)
        printf("Tree is empty");

    // Reach leftmost node
    struct Node *ptr = root;
    while (ptr -> lthread == false)
        ptr = ptr -> left;

    // One by one print successors
    while (ptr != NULL)
    {
        printf("%d ",ptr -> info);
        ptr = inorderSuccessor(ptr);
    }
}

// Driver Program
int main()
{
    struct Node *root = NULL;

\\ create Tree
    root = insert(root, 20);
    root = insert(root, 10);
    root = insert(root, 30);
    root = insert(root, 5);
    root = insert(root, 16);
    root = insert(root, 14);
    root = insert(root, 17);
    root = insert(root, 13);

    inorder(root);

    return 0;
}
```

Performance of the serial implementations

The run-times of the three serial implementations are the input digraph contained 15 vertices (including the hometown), and all three algorithms visited approximately 95,000,000 tree nodes. The first iterative version is less than 5% faster than the recursive version, and the second iterative version is about 8% slower than the recursive version. As expected, the first iterative solution eliminates some of the overhead due to repeated function calls, while the second iterative solution is slower because of the repeated copying of tour data structures. However, as we'll see, the second iterative solution is relatively easy to parallelize, so we'll be using it as the basis for the parallel versions of tree search.

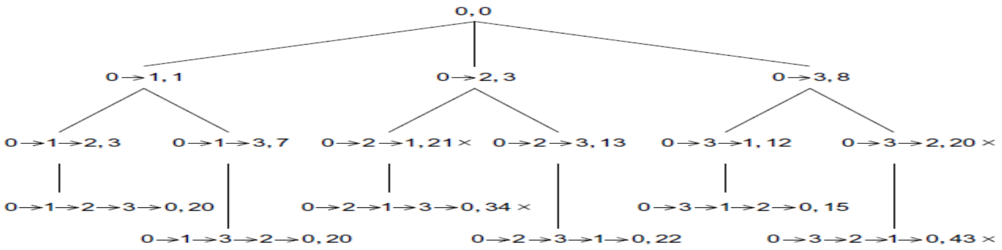
Table 5.1



Recursive	First Iterative	Second Iterative
30.5	29.2	32.9

Parallelizing Tree Search

The tree structure suggests that we identify tasks with tree nodes. If we do this, the tasks will communicate down the tree edges: a parent will communicate a new partial tour to a child, but a child, except for terminating, doesn't communicate directly with a parent. We also need to take into consideration the updating and use of the best tour. Each task examines the best tour to determine whether the current partial tour is feasible or the current complete tour has lower cost. If a leaf task determines its tour is a better tour, then it will also update the best tour. Although all of the actual computation can be considered to be carried out by the tree node tasks, we need to keep in mind that the best tour data structure requires additional communication that is not explicit in the tree edges. Thus, it's convenient to add an additional task that corresponds to the best tour. It "sends" data to every tree node task, and receives data from some of the leaves. This latter view is convenient for shared-memory, but not so convenient for distributed-memory. A natural way to agglomerate and map the tasks is to assign a subtree to each thread or process, and have each thread/process carry out all the tasks in its subtree. For example, if we have three threads or processes, as shown earlier in Figure 5.1, we might map the subtree rooted at 0→1 to thread/process 0, the subtree rooted at 0→2 to thread/process 1, and the subtree rooted at 0→3 to thread/process 2.



OpenMP - Open Multi-Processing

- An API for developing multi-threaded (MT) applications
- Consists of a set of compiler directives and library routines for parallel application programmers
- Simplifies writing MT programs in Fortran, C and C++
- Augments vectorization and standardizes programming of various platforms
- Embedded systems, accelerator devices (GPU), multi-core systems (CPU)
- Name and specification maintained by OpenMP Architecture Review Board

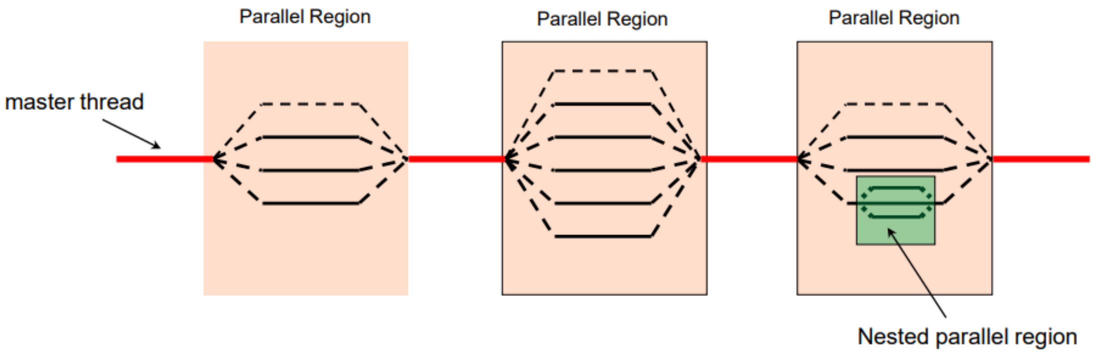
OpenMP Programming Model

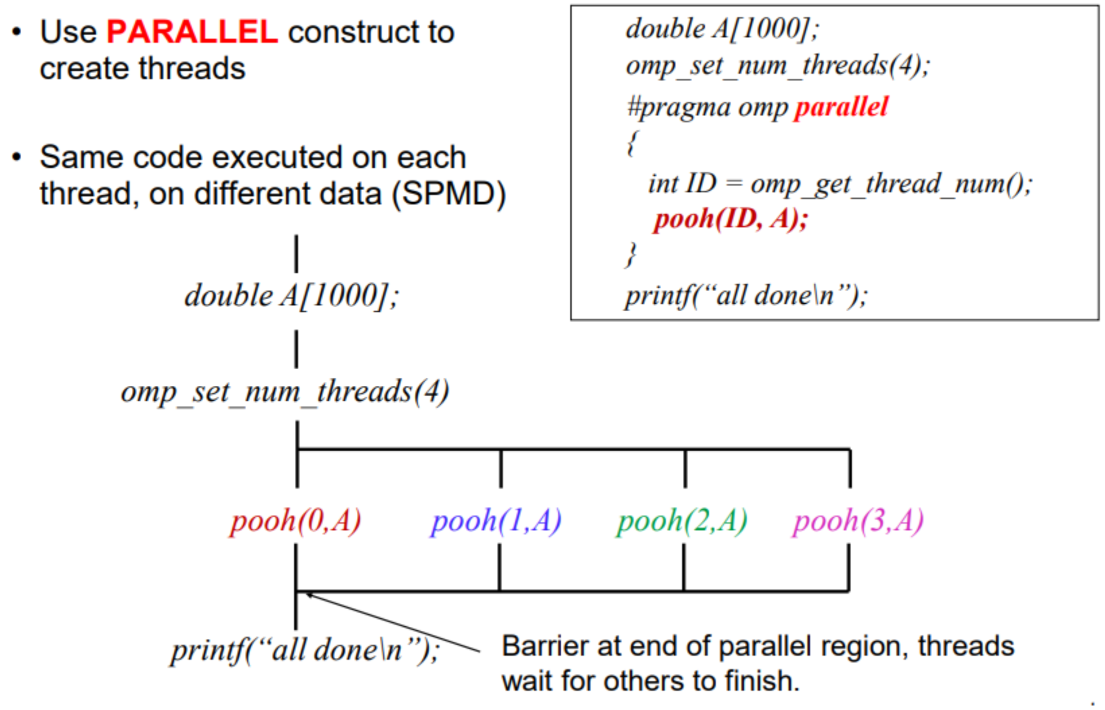
Fork-Join Parallelism:

◆ Master thread spawns a team of threads as needed.

◆ Parallelism added incrementally until performance goals are met.

◆ Threads within a parallel region can spawn more threads – nested parallelism





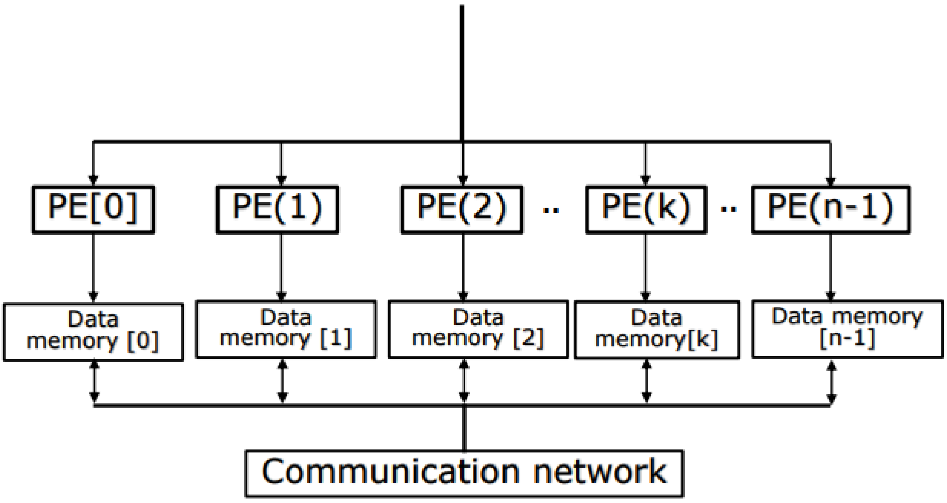
- Use **PARALLEL** construct to create threads
- Same code executed on each thread, on different data (SPMD)

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int ID = omp_get_thread_num();
   pooh(ID, A);
}
printf(“all done\n”);
```

Barrier at end of parallel region, threads wait for others to finish.

What is SPMD?

- SPMD – Single Program Multiple Data
- Part of the MIMD category in Flynn’s taxonomy
- Multiple Processing Elements (PE) that run a copy of the same program and operate on different data elements



Hello world openmp

```
// OpenMP program to print Hello World

// using C language

// OpenMP header

#include <omp.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char* argv[])

{       // Beginning of parallel region

        #pragma omp parallel

        {               printf("Hello World... from thread = %d\n",

                        omp_get_thread_num());

        }       // Ending of parallel region

}
```

Appendix A

MPI

mpi.h header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.

MPI Init and MPI Finalize

- The call to MPI Init tells the MPI system to do all of the necessary setup. مهم
  - allocate storage for message buffers.
  - decide which process gets which rank.
  - define a global communicator .
- No other MPI functions should be called before the program calls MPI_Init.

Int MPI Init(**int*** argc_p **, char***** argv_p); *in\out*\

- The arguments**, argc_p** and **argv_p**, are pointers to the arguments to main, argc, and argv.
- When our program doesn't use these arguments, we can just pass NULL for both.
- MPI functions, **MPI_Init** returns an **int** error code, and in most cases we'll ignore these error codes. مهم
- The call to MPI Finalize tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. مهم

The syntax is quite simple:

Int MPI_Finalize(**void**);

Out line of mpi program

```
#include <mpi.h>
. . .
int main(int argc, char* argv[])
{
. . .
/* No MPI calls before this */
MPI_Init(&argc, &argv);
. . .
MPI_Finalize();
```

/∗ No MPI calls after this ∗/

```
. . .
return 0;
}
```

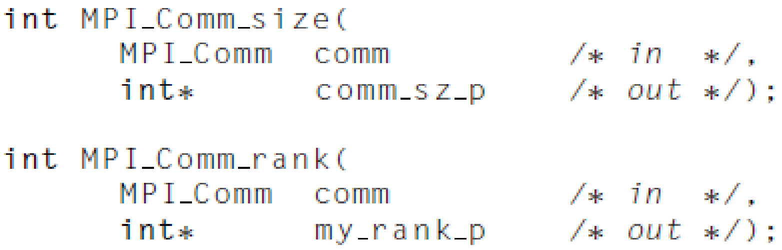
It’s also not necessary that the calls to **MPI Init** and **MPI Finalize** be in main.

- **Communicators, MPI_Comm_size and MPI _Comm_rank**

- In MPI a **communicator** is a collection of processes that can send messages to each other.

- One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when it started the program.

- This communicator is called **MPI_COMM_WORLD**, their syntax is:



```
int MPI_Comm_size(
      MPI_Comm   comm         /* in   */,
      int*       comm_sz_p    /* out  */);

int MPI_Comm_rank(
      MPI_Comm   comm         /* in   */,
      int*       my_rank_p    /* out  */);
```

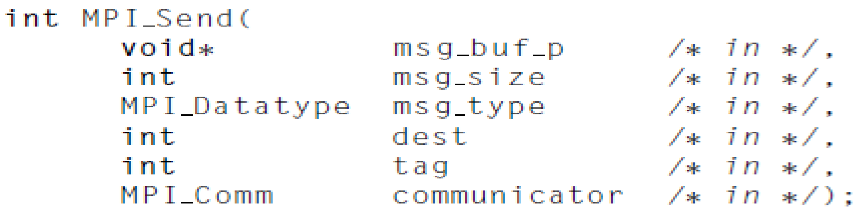
- **MPI_Comm_size** returns the number of processes in the communicator, **MPI_Comm_rank** returns the calling process’ rank in the communicator. The variable **comm_sz** for the number of processes in **MPI_COMM_WORLD**, and the variable **my_rank** for the process rank**.**

Ex : Comm(commsize,commrank)

ملاحظة - MPI_Comm_rank يرجع كالنك بل بروسسر المستدعاة اي الرانك مالتهة

- MPI_Comm_size يرجع عدد البروسس بل كوميونيكايتر ونستعملة لمعرفة عدد البروسس بل comm_world

MPI Send



```
int MPI_Send(
      void*           msg_buf_p      /* in */,
      int             msg_size       /* in */,
      MPI_Datatype    msg_type       /* in */,
      int             dest           /* in */,
      int             tag            /* in */,
      MPI_Comm        communicator   /* in */);
```

- The first three arguments, **msg_buf_p, msg_size**, and **msg_type,** determine the contents of the message.
- The remaining arguments, **dest, tag**, and **communicator**, determine the destination of the message.
- **msg_buf_p**, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, **greeting**.
- The second and third arguments, **msg_size** and **msg_type**, determine the amount of data to be sent. In our program, the **msg_size** argument is the number of characters in the message plus one character for the '\0' character that terminates C strings.
- The **msg_type** argument is **MPI_CHAR**. These two arguments together tell the system that the message contains **strlen(greeting)+1 char**s. Since C types (**int**, **char**, and so on.) can't be passed as arguments to functions, MPI defines a special type, **MPI_Datatype**, that is used for the **msg_type** argument. MPI also defines a number of constant values for this type.

The program send function

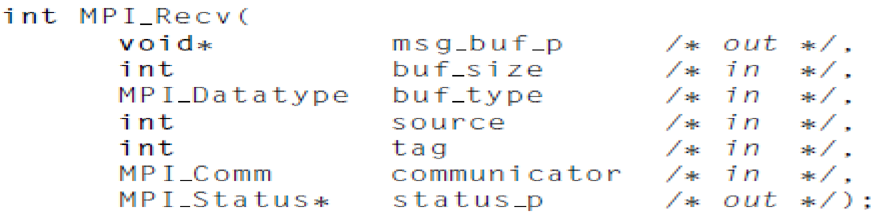
- **MPI_Send**(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
- **MPI_Send**(**msg_buf_p**, **msg_size**, **msg_type**, dest, tag, communicator);

ملاحظة اول 3 اللي بالاصفر هي محتويات الرسالة اللي بالاحمر هي وجهة الرسالة.

ملاحظة: نلاحظ ان (Dest) هو يحدد رانك البروسس اللي يستلم الرسالة بينما ال(tag) هو رقم موجب يميز تطابق الرسالة.

- One of the most important purposes of communicators is to specify communication universes; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that's using a different communicator.

- **MPI Recv**



```
int MPI_Recv(
      void*          msg_buf_p     /* out */,
      int            buf_size      /* in  */,
      MPI_Datatype   buf_type      /* in  */,
      int            source        /* in  */,
      int            tag           /* in  */,
      MPI_Comm       communicator  /* in  */,
      MPI_Status*    status_p      /* out */);
```

MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Recv(**msg_buf_p**, **buf_size**, **buf_type**, **source**, **tag**, **communicator**, status_p);

- The first three arguments specify the memory available for receiving the message **msg_buf_p** points to the block of memory.
- **buf_size** determines the number of objects that can be stored in the block.
- **buf_type** indicates the type of the objects
- The next three arguments identify the message. The **source** argument specifies the process from which the message should be received.
- The **tag** argument should match the **tag** argument of the message being sent.
- The **communicator** argument must match the communicator used by the sending process.
- The status **p** argument in many cases it won't be used by the calling function, and, as in our **"greetings"** program, the special MPI constant **MPI_STATUS_IGNORE** can be passed.

Performance

- **Taking timings**
- We're usually not interested in the time taken from the start of program execution to the end of program execution. We're only interested in the time it takes to do the actual processing, a **function that will tell us the amount of time that elapses from the beginning to the end of the** actual processing. MPI provides a function, **MPI_Wtime**, (فائدة مهم)that returns the number of seconds that have elapsed since some time in the past:

```
double MPI Wtime(void);
```

Thus, we can time a block of MPI code as follows:

MPI Time

```
double start, finish;
.. .
start = MPI_Wtime( );
/* Code to be timed */
. . .
finish = MPI_Wtime( );
printf("Proc %d > Elapsed time = %e seconds\n" my_rank, finish-start);
```

- When we run a program several times, we're likely to see a substantial variation in the times.(depend on the HW&SW)
- This will be true even if for each run we use the same input, the same number of processes, and the same system. This is because the **interaction** of the program with the rest of the system, especially the operating system, is unpredictable. Since this interaction will almost certainly not make the program run faster than it would run on a "quiet" system, we usually report the *minimum* run-time rather than the mean or median.

When we run an MPI program on a hybrid system in which the nodes are multicore processors, we'll only run **one MPI process** on each node. This may reduce contention for the interconnect and result in somewhat better run-times, It may also reduce variability in run-times.

- The parallel program will divide the work of the serial program among the processes, and add in some overhead time, in MPI programs, the parallel overhead typically comes from communication, and it can depend on both the problem size and the number of p

$$T_{\text{parallel}}(n,p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

- **Speedup and efficiency**

- The goal is to equally distribute the workload among all the processors, whereas resulting in no extra load on the cores.
- If this goal is reached, and the program runs with a P number of cores, one thread or process on each core, then the parallel application will be executed P times faster than the sequential application.

- If sequential execution time is named $\mathbf{T_{serial}}$ and the parallel execution time called $\mathbf{T_{parallel}}$, then the ultimate case of the resulting parallel time is calculated from:

$$\mathbf{T_{parallel} = T_{serial} / P \ \ldots\ldots\ (2.1)}$$

- If this happens, then this parallel program has **linear speedup**. Practically, this case is unlikely to happen because the exploitation of a number of processes or threads usually introduces some inevitable overhead.
- Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It's just the ratio of the serial run-time to the parallel run-time:

$$S(n,p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n,p)}.$$

- The ideal value for ***S*(*n*,*p*)** is ***p***. If ***S*(*n*,*p*) = *p***, then our parallel program with comm_sz = *p* processes is running *p* times faster than the serial program.
- In practice, this speedup, sometimes called **linear speedup**, is rarely achieved. For small *p* and large *n*, our program obtained nearly linear speedup ,on the other hand, for large *p* and small *n*, the speedup was considerably less than *p*. The worst case was *n* = 1024 and *p* = 16, when we only managed a speedup of 2.4.

$$\mathbf{E = S/P = (T_{serial} / T_{parallel})/P = T_{serial} / (T_{parallel} * P)}$$

We also recall that another widely used measure of parallel performance is parallel **efficiency**. This is “per process” speedup:

$$E(n,p) = \frac{S(n,p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n,p)}.$$

Linear speedup corresponds to a parallel efficiency of *p*/*p* = 1.0, and, in general, we expect that our efficiencies will be less than 1.

Amdahl’s Law calculates the speedup of parallel code based on three variables:

■ Duration of running the application on a single-core machine
■ The percentage of the application that is parallel
■ The number of processor cores

Here is the formula, which returns the ratio of single-core versus multicore performance.

$$\text{Speedup} = \frac{1}{1 - P + (P/N)}$$

- **Scalability**
- **Our parallel program doesn’t come close to obtaining linear speedup for small *n* and large *p*. Does this mean that it’s not a good program?** Many computer scientists answer this question by looking at the “scalability” of the program.
- A program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn’t decrease as the number of processes increase.
- Programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**.
- Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**.
- **Q: What do we mean by a parallel sorting algorithm in a distributed-memory environment? What would its “input” be and what would its “output” be?** The answers depend on where the keys are stored. We can start or finish with the keys distributed among the processes or assigned to a single process.