AL-MAMOON UNIVERSITY COLLEGE

Computer Science Department

3 rd Class

Parallel Programming

Dr Taif Sami

Lecture 1

2020-2021

Parallel Programming

In computing, **a parallel programming model** is an abstraction of parallel computer architecture, with which it is convenient to express algorithms and their composition in programs. The value of a programming model can be judged on its *generality*: how well a range of different problems can be expressed for a variety of different architectures, and its *performance*: how efficiently the compiled programs can execute.

Classification of parallel programming model

Classifications of parallel programming models can be divided broadly into two areas:

process interaction and problem decomposition

Process interaction

Process interaction relates to the mechanisms by which parallel processes are able to communicate with each other. The most common forms of interaction are shared memory and message passing, but interaction can also be implicit (invisible to the programmer).

Shared memory

- Shared memory is an efficient means of passing data between processes.
- In a shared-memory model,
- parallel processes share a global address space that they read and write to asynchronously.
- Asynchronous concurrent access can lead to

race conditions, and mechanisms such as locks, semaphores and monitors can be used to avoid these. Conventional multi-core processors directly support shared memory, which many parallel programming languages and libraries, such as Cilk, OpenMP and Threading Building Blocks, are designed to exploit.

Message passing

In a message-passing model, parallel processes exchange data through passing messages to one another. These communications can be

- asynchronous, where a message can be sent before the receiver is ready,
- or synchronous, where the receiver must be ready.

The Communicating sequential processes (CSP) formalisation of message passing uses synchronous communication channels to connect processes, and led to important languages such as Occam, Limbo and Go.

In contrast, the actor model uses asynchronous message passing and has been employed in the design of languages such as D, Scala and SALSA.

Implicit interaction

In an implicit model, no process interaction is visible to the programmer and instead the compiler and/or runtime is responsible for performing it.

Two examples of implicit parallelism are with domain-specific languages where the concurrency within high-level operations is prescribed, and with functional programming languages because the absence of side-effects allows non-dependent functions to be executed in parallel.

Thank You

AL-MAMOON UNIVERSITY COLLEGE

Computer Science Department

Parallel Programming

Lecture 5

2020-2021

1. Basic Concepts

Single core processor: is a microprocessor with a single core in one chip, running a single thread in one time.

Multi-core processor: is a microprocessor with multiple cores in one chip, running multiple threads in one time.

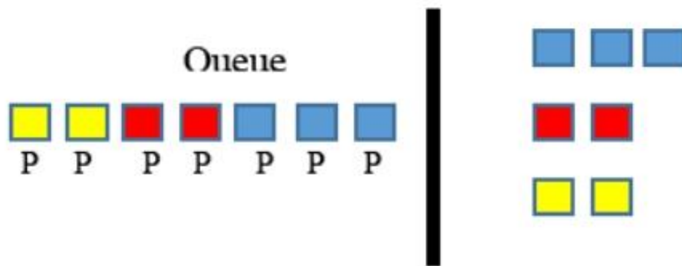


Figure 1 Parallel Computing example

Parallel Computing: is a type of computation to proceed many calculations in simultaneously. As a benefit,
is often used to divide larger problems in smaller once, which can then be solved at the same time (see fig1).

Process(Computing): is the instance of a running program that is being executed by one or many threads. Each process is an independent entity to which system resources such as CPU time, memory, etc. are allocated and each process is executed in a separate address space.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory (see fig2).

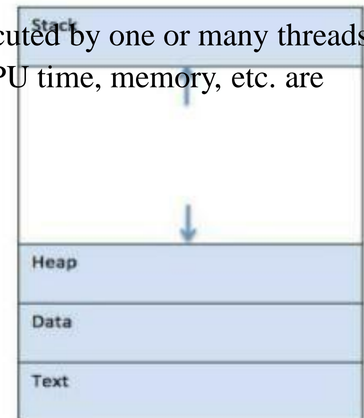


Figure 2 Process layout inside memory.

Component & Description

1. Stack The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2. Heap This is dynamically allocated memory to a process during its run time.
3. Text This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4. Data This section contains the global and static variables.

Program: is a piece of code which may be a single line or millions of lines.

```
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

Process Life Cycle

When a process executes, it passes through different states. In general, a process can have one of the following five states at a time (see fig 3).

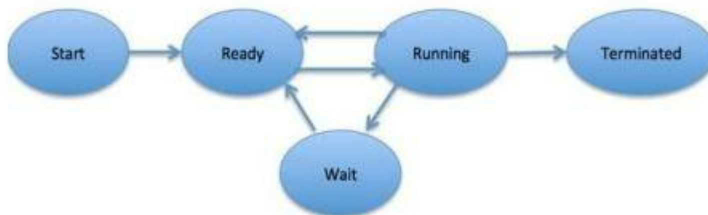


Figure 3 Process life cycle

Thread: is the smallest sequence of programmed instructions that can be managed independently by a scheduler. Multiple threads can exist within one process (see fig 4).

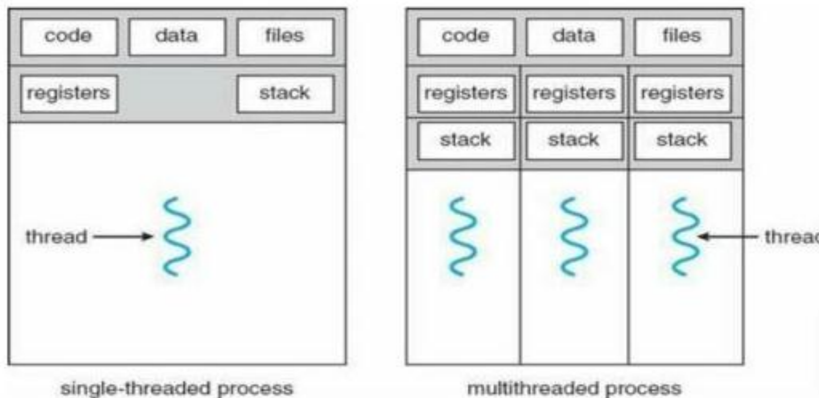


Figure 4 Process with single and multithread example

Task: is a unit of execution or a unit of work, that occurs during the first step of executing parallel program.

A first step in designing the parallel program is to break the problem into separate "chunks or pieces" of work that can be distributed to multiple tasks.

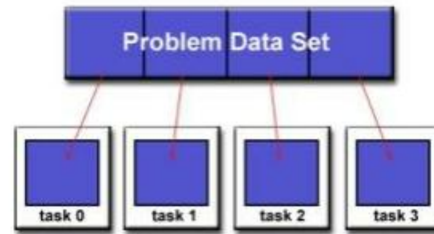
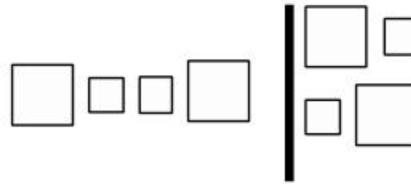


Figure 5 Tasks description

Task granularity: The size of tasks (e.g., in terms of the number of instructions) and there is typically the possibility of choosing tasks of different sizes.



Scheduling: is the assignment of tasks to processes or threads with a fixed order to execute the tasks. Scheduling can be done by hand in the source code or by the programming environment, at compile time or dynamically at runtime.

Mapping: is the assignment of processes or threads onto the physical units, processors or cores, and it is usually done by the runtime system but can sometimes be influenced by the programmer.

Note: To execute any parallel programs correctly need to consider the synchronization and coordination of threads and processes.

Parallel execution time: is the time elapsed between the start of the application on the first processor and the end of the execution of the application on all processors. computation time on processors or cores and synchronization time for data exchange are important aspects in parallel computing.

The parallel execution time should be smaller than the sequential execution time on one processor so that designing a parallel program is worth the effort.

Parallel program must have →parallel execution time < sequential execution time

Less than

Idle times: is an unproductive time where the processor cannot do anything useful, only wait for an event to happen.

Load balancing: is good synchronization and mapping strategy to the parallel program with small idle time. This occurs when the work load is assigned equally to processors or cores.

2. Types of Parallel Programming

Parallel processing can be subdivided into two groups as explained below.

a) Task based: In this type every processor in the system will execute different jobs or tasks in a parallel way. For example, in MS Word, doing two tasks (printing and Spell Checking) at the same time considered as separated tasks in different processors.
b) Data based: In this type, each processor in the system will perform different data sets in parallel way. For example, in image processing, convert the image color to grayscale by dividing the image into upper and lower halves and each half converted by different processors.

3. Shared memory or distributed memory models

In the shared memory; in case of desktop or laptop computers; there is only one memory unit and multiprocessing units, all processors in this model have the right access to the memory unit which is shared between them.

In the distributed memory; as in the computer cluster; every multiple processing unit have their own memory unit to store the data. In this model, the information is passed between these units.

The computer clusters; are separate units of computers that are connected through the network, working as a single system.

Thus, the multiprocessors can be divided into two shared-memory model as main categories- UMA (Uniform Memory Access), NUMA (Non-uniform Memory Access) as illustrated in Fig 6.

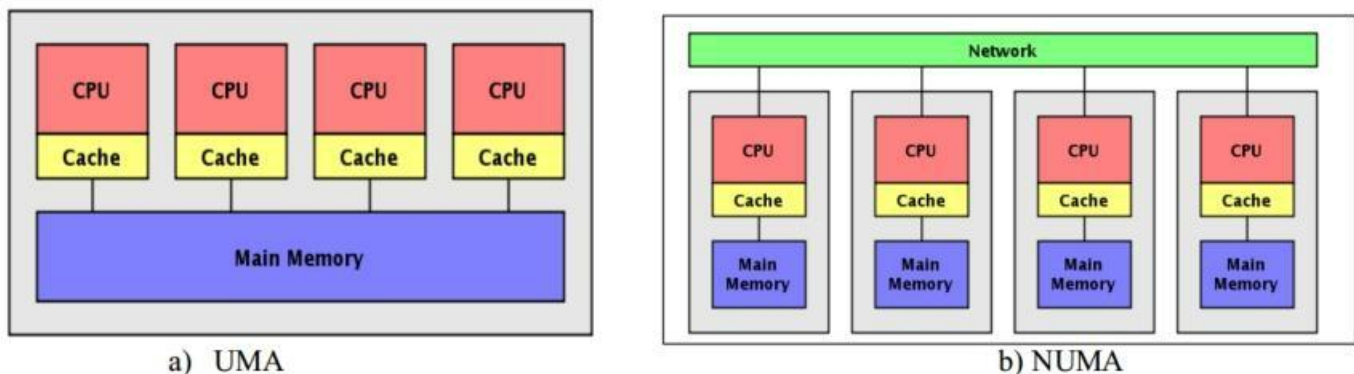


Figure 6 Shared (UMA) and distributed (NUMA) memory models

Each processor needs its own private cache to be fed quickly.

4. Flynn's Taxonomy

First proposed by Michael Flynn in 1966, Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of synchronized instruction (single or multiple) and data streams (single or multiple) available in the architecture.

The four categories in Flynn's taxonomy are the following:

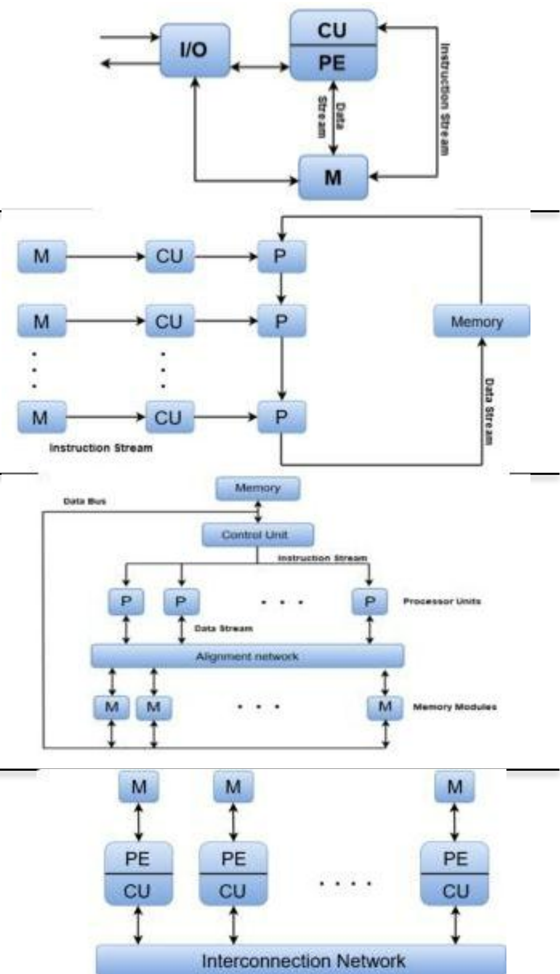
a) Single instruction, Single data (SISD): a single instruction, operating on a single data stream. All the instructions and data to be processed have to be stored in primary memory.

b) Multiple instruction, Single data (MISD): multiple instructions operate on one data stream. Each processing unit operates on the data independently via separate instruction stream.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

c) Single instruction, Multiple data (SIMD): a single instruction operates on multiple different data streams. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

d) Multiple instruction, Multiple data (MIMD): multiple autonomous processors simultaneously executing different instructions on different data. In MIMD, each processor has a separate program and an instruction stream is generated from each program.



Where, CU = Control Unit, PE = Processing Element, M = Memory

The motivations of using parallel processing can be summarized as follows:

- Faster execution time, parallel processing allowing the execution of applications in a shorter time.
- Higher computational power, to solve larger problems in a short point of time.
- Higher throughput, to solve many problems or large problems in short time.
- Working on different tasks simultaneously, you can do many things simultaneously by using multiple computing resources.

5. Effectiveness of Parallel processing

Throughout the book, we will be using certain measures to compare the effectiveness of various parallel algorithms or architectures for solving desired problems. The following definitions and notations are applicable.

6

p	Number of processors
$W(p)$	Total number of unit operations performed by the p processors; this is often referred to as computational work or energy
$T(p)$	Execution time with p processors; clearly, $T(1) = W(1)$ *and* $T(p) \leq W(p)$
$S(p)$	$Speed - up = \frac{T(1)}{T(p)}$
$E(p)$	$Efficiency = \frac{T(1)}{pT(p)}$
$R(p)$	$Redundancy = \frac{W(p)}{W(1)}$
$U(p)$	$Utilization = \frac{W(p)}{pT(p)}$
$Q(p)$	$Quality = \frac{T^3(1)}{pT^2(p)W(p)}$ (1)

Example: Finding the sum of 16 numbers can be represented by the binary-tree computation graph of Figure 4 with $T(1) = W(1) = 15$. Assume unit-time additions and ignore all else. With $p = 8$ processors, we have

$W(8) = 15 \quad T(8) = 4$
$E(8) = 15/(8 \cdot 4) = 47\%$
$S(8) = 15/4 = 3.75$
$R(8) = 15/15 = 1$
$Q(8) = 1.76$

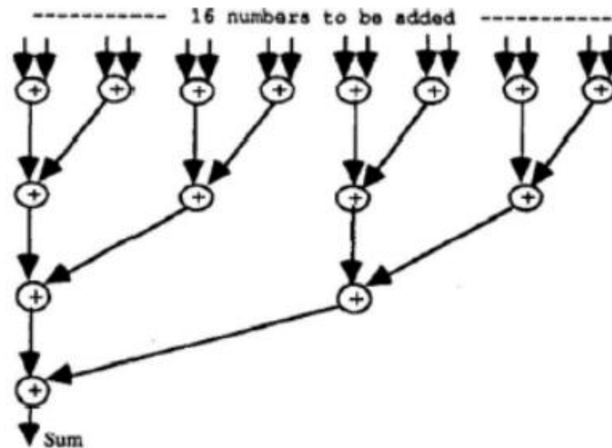


Figure 7 Computation graph for finding the sum of 16 numbers

Essentially, the 8 processors perform all of the additions at the same tree level in each time unit, beginning with the leaf nodes and ending at the root. The relatively low efficiency is the result of limited parallelism near the root of the tree.

Speedup

Speedup is the expected performance benefit from running an application on a multicore versus a single-core machine. When speedup is measured, single-core machine performance is the baseline. For example, assume that the duration of an application on a single-core machine is six hours. You might expect that an application running on a single-core machine would run twice as quickly on a dual-core machine, and that a quad core machine would run the application four times as fast. But that's not exactly correct.

Here are some of the limitations to linear speedup (full parallelization) of parallel code:
Serial code

- Overhead from parallelization
- Synchronization
- Sequential input/output

Predicting speedup is important in designing, benchmarking, and testing your parallel application. Fortunately, there are formulas for calculating speedup. One such formulas are Amdahl's Law and Gustafson's Law.

- **Amdahl's Law**

It 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

$$Speedup = \frac{1}{1 - p(\frac{P}{N})}$$

P : is the percent of the application that runs in parallel.
N : is the number of processor cores.

Example: 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).

$$Speedup = \frac{1}{1 - 0.75(\frac{0.75}{N3})} = 2$$

The application will run twice as fast on a three processor-core machine.

❖ Gustafson's Law

You may have an application that's split consistently into a sequential and parallel portion. Amdahl's Law maintains these proportions as additional processors are added. The serial and parallel portions each remain half of the program. But in the real world, as computing power increases.

Amdahl's Law does not account for the overhead required to schedule, manage, and execute parallel tasks. Gustafson's Law takes both of these additional factors into account. Here is the formula to calculate speedup by using Gustafson's Law.

$$Speedup = \frac{S + N(1 - S)}{S + (1 - S)} - O_n$$

In the above formula, S is the percentage of the serial code in the application, N is the number of processor cores, and O_n is the overhead from parallelization.

Also speedup can be computed using the bellow equation

Speedup = s + p x N
s--> time spend in executing serial
p--> time spend in executing parallel
N--> no of processor
also s + p = 1

AL-MAMOON UNIVERSITY COLLEGE

Computer Science Department

4 th Class

Parallel Programming

Dr Taif Sami

Lecture 3

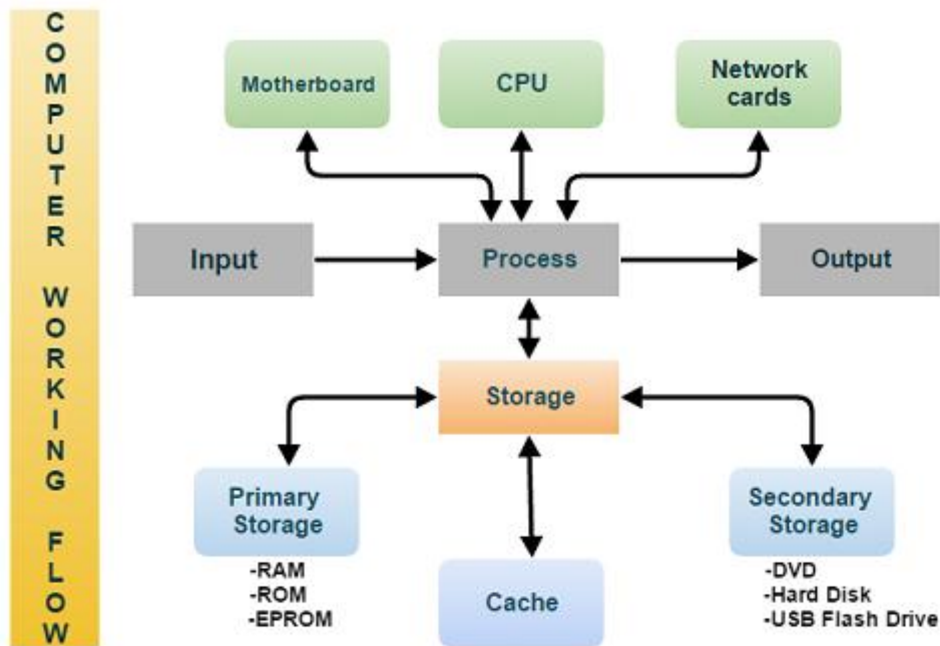
2020-2021

Program

- A computer program is a collection of instructions that performs a specific task when executed by a computer.
- A computer program is usually written by a computer programmer in a programming language.

Process

- In computing, a process is the instance of a computer program that is being executed by one or many threads.
- It contains the program code and its activity.
- Depending on the operating system (OS), a process may be made up of multiple threads of execution that execute instructions concurrently.



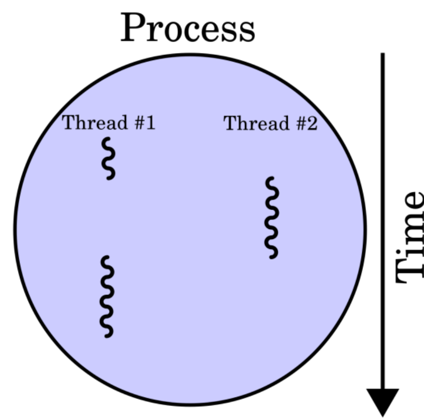
- Any running instance of a program is called as process.
- Or it can also be described as a program under execution.
- A program can have N processes.
- Process resides in main memory & hence disappears whenever machine reboots.
- Multiple processes can be run in parallel on a multiprocessor system.

Task

A task is a set of program instructions that are loaded in memory. Short answer: A thread is a scheduling concept, it's what the CPU actually 'runs' (you don't run a process). A process needs at least one thread that the CPU/OS executes

Thread

- In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.
- A Thread is commonly described as a lightweight process.
- 1 process can have N threads.
- All threads which are associated with a common process share same memory as of process.



Thread

Definition:

- A thread is a single sequential flow of control within a program.

A Program

A Thread

Difference between Process and Thread

- The essential difference between a thread and a process is the work that each one is used to accomplish.
- Threads are being used for small & compact tasks, whereas processes are being used for more heavy tasks.
- One major difference between a thread and a process is that threads within the same process consume the same address space, whereas different processes do not. This allows threads to read from and write to the common shared and data structures and variables, and also increases ease of communication between threads.
- Communication between two or more processes – also known as Inter-Process Communication i.e. IPC – is quite difficult and uses intensive resources.

Difference between Task and Thread

- Tasks are very much similar to threads, the difference is that they generally do not interact directly with OS.
- Like a Thread Pool, a task does not create its own OS thread. A task may or may not have more than one thread internally.
- If you want to know when to use Task and when to use Thread: Task is simpler to use and more efficient that creating your own Threads. But sometimes, you need more control than what is offered by Task. In those cases, it makes more sense to use Thread directly.
- The bottom line is that Task is almost always the best option; it provides a much more powerful API and avoids wasting OS threads.
- The only reasons to explicitly create your own Threads in modern code are setting per-thread options, or maintaining a persistent thread that needs to maintain its own identity.

Process vs. Thread

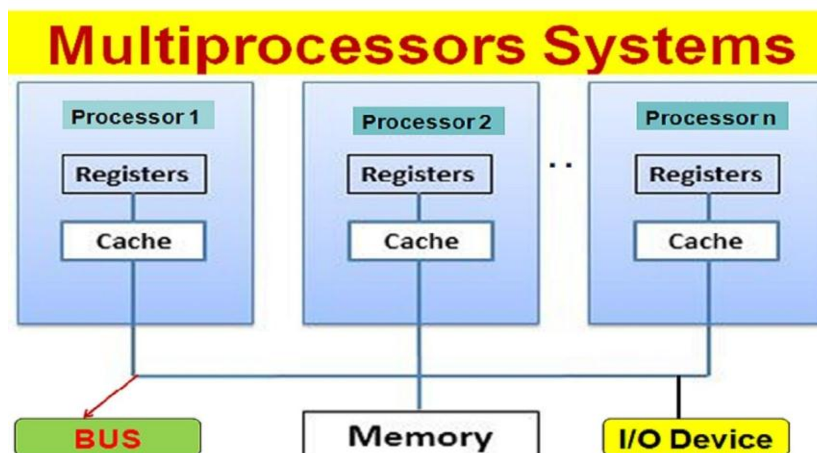
- Process
  - address space, program code, global variables, heap, OS resources : files, I/O devices
- Thread
  - is a single sequential execution stream within a process
- A process is a multithread where
  - Each thread has its **own** (other threads can access but shouldn't) registers, program counter (PC) , stack, stack pointer (SP)
  - All threads ***shares*** process resources
  - Threads executes concurrently
  - Each thread can block, fork, terminate, synchronize

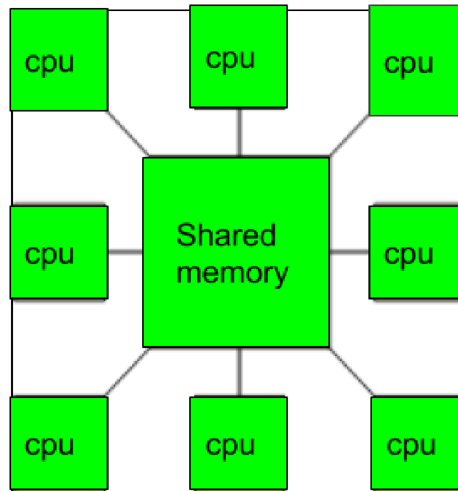
Parallelism In Computer

- The term Parallelism refers to techniques to make programs faster by performing several computations at the same time.
- This requires hardware with multiple processing units. ... To this end, it can even be an advantage to do the same computation twice on different units

Multiprocessing

Multiprocessing, in computing, a mode of operation in which two or more processors in a computer simultaneously process two or more different portions of the same program



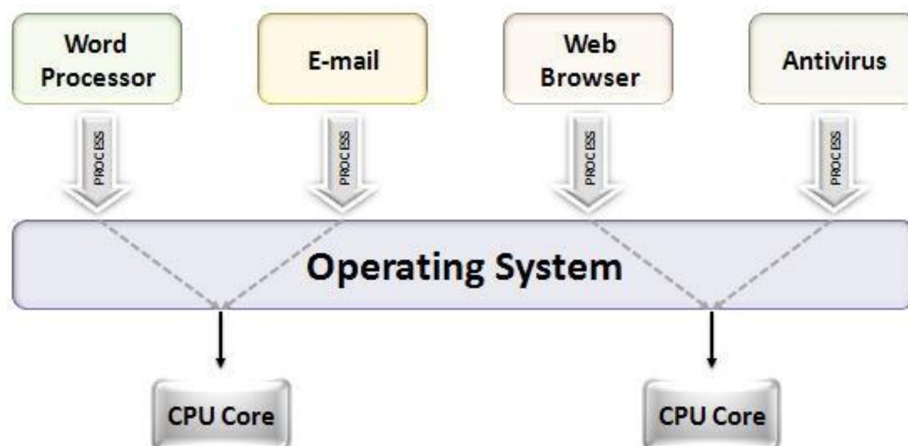


Multithread

In computer architecture, multithreading is the ability of a central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the operating system. This approach differs from multiprocessing.

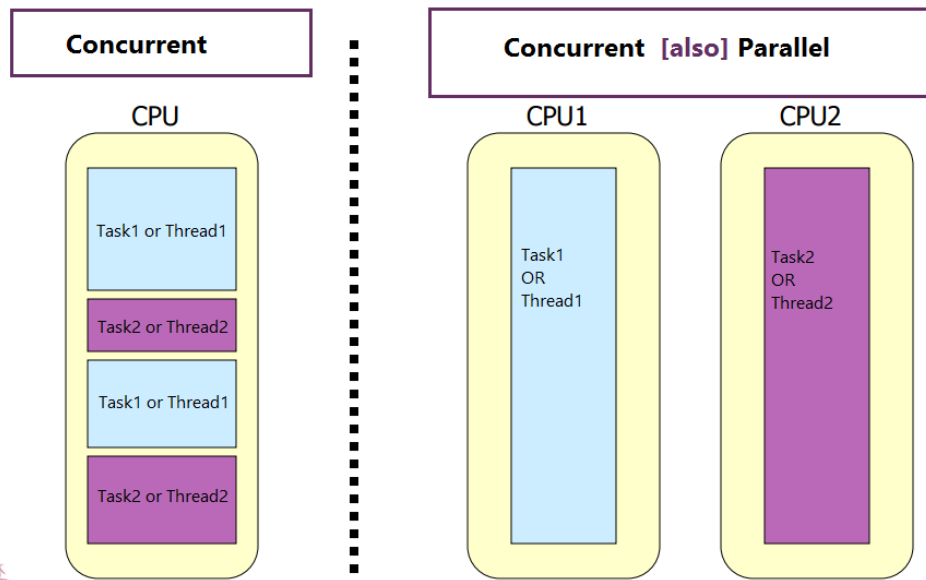
Multitasking

Multitasking, the running of two or more programs (sets of instructions) in one computer at the same time. Multitasking is used to keep all of a computer's resources at work as much of the time as possible.



Concurrency

In computer science, concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome



Parallel Computing

Parallel computing is a type of computation where many calculations or the execution of processes are carried out simultaneously. Large problems can often be divided into smaller ones, which can then be solved at the same time.

CPU Core

- A core is a small CPU or processor built into a CPU or CPU socket.
- It can independently perform or process all computational tasks.
- From this perspective, we can consider a core to be a smaller CPU or a smaller processor within a big processor

Intel Core

What’s a Core Processor?

- Intel Core processors first came to the desktop in mid-2006, replacing the Pentium line that had previously comprised Intel’s high-end processors.
- The Core “i” names are primarily “high level” categorizations that help differentiate processors within a given generation.
- A specific Core “i” name doesn’t mean the processor has a certain number of cores, nor does it guarantee features, like Hyper-Threading, which allows the CPU to process instructions faster.
- Feature specifics can change between generations. As technology advances, it becomes cheaper to create higher-performing, low-end parts.
- It also means that features once found in parts like a Core i3 can disappear from the class entirely.
- Therefore, the differences between Core i3, Core i5, and Core i7 designations matter most within its respective generation.
- For example, a seventh-generation “Kaby Lake” Core i7, and a third-generation “Ivy Bridge” Core i7 might run at similar speeds with similar core counts.
- Intel Core i3 processors are where the Core lineup starts for each generation. Those earlier dual-core Core i3’s also tended to have four threads, also known as Hyper-Threading.
- Intel has elected not to double the thread count in recent Core i3 generations; instead, it’s building CPUs with four cores and four threads.
- Core i3 processors also have lower cache sizes (onboard memory).
- They handle less RAM than other Core processors and have varying clock speeds. At this writing, the ninth-generation, Core i3 desktop processors have a top clock speed of 4.6 GHz
- Core i5: The Lower Mid-Range, a step up from Core i3 is the Core i5. This is often where bargain-hunting PC gamers look for solid deals on processors. An i5 typically lacks Hyper-Threading, but it has more cores (currently, six, rather than four) than Core i3. The i5 parts also generally have higher clock speeds, a larger cache, and can handle more memory. The integrated graphics are also a bit better. You see new Core i5 processors with Hyper-Threading on laptops, but not desktops.

- Core i7: As of 2017, Core i7 CPUs had Hyper-Threading on desktops, but the more recent generations do not. These processors have higher core counts (up to eight in the ninth generation) than the i5's, a larger cache, and a bump in graphics performance, but they have the same memory capacity as the Core i5's (although, that could change in the future).
- Core i9: An Intel-based gaming PC. The Core i9 is at the top of the Intel Core pack. This is where you find many top-performing processors, like the Core i9-9900K—a current favorite for gaming.
- At the Core i9 level in the current ninth-generation CPUs, we see eight cores, 16 threads, a larger cache than the Core i5 processors, faster clock speeds (up to 5 GHz for boost), and another bump in graphics performance. However, Core i9 CPUs still have the same maximum memory capacity as the Core i5.
- Core X: The Ultimate Intel also has a "prosumer" range of fancier, high-end desktop (HEDT) processors for enthusiasts, gamers, content creators, or anyone else who needs that level of performance.

In October 2019, Intel announced new Core X parts that range from 10 to 18 cores (Core i9s max out at eight). They include Hyper-Threading, and high boost clocks, although, not necessarily higher than Core i9 CPUs. They also have a higher number of PCIe lanes and can handle more RAM, and they have a much higher TDP than the other Core parts.

Which Should You Buy?

Core designations refer to relative improvements within a specific generation of processors. As the Core number increases, so do the capabilities of the processors, including higher core counts, faster clock speeds, more cache, and the ability to handle more RAM. At Core X, you also usually get more PCIe lanes.

If you're a gamer, look for Core i7 and higher. You can definitely game with a newer Core i5, but you'll get more future-proofing with a Core i7 and up. Content creators should look at Core i7 and Core i9 CPUs, as you'll want those sweet threads.

For everyday tasks, like web browsing, spreadsheets, and word processing, a Core i3 will get the job done.

Something to keep in mind while you shop, though, is not all Intel Core CPUs have integrated graphics. These processors end with an "F" to designate that they come without a GPU, such as the Core i3-9350KF, i5-9600KF, and i9-9900KF.

Flynn's Taxonomy

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

Single instruction stream, single data stream (SISD)

- A sequential computer which exploits no parallelism in either the instruction or data streams.
- Single control unit (CU) fetches single instruction stream (IS) from memory.
- The CU then generates appropriate control signals to direct single processing element (PE) to operate on single data stream (DS) i.e., one operation at a time.
- Examples of SISD architecture are the traditional uniprocessor machines like older personal computers (PCs; by 2010, many PCs had multiple cores) and mainframe computers.

Single instruction stream, multiple data streams (SIMD)

- A single instruction operates on multiple different data streams. Instructions can be executed sequentially,
- such as by pipelining, or in parallel by multiple functional units.
- Single instruction, multiple threads (SIMT) is an execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading. This is not a distinct classification in Flynn's taxonomy,

where it would be a subset of SIMD. Nvidia commonly uses the term in its marketing materials and technical documents where it argues for the novelty of Nvidia architecture

Multiple instruction streams, single data stream (MISD)

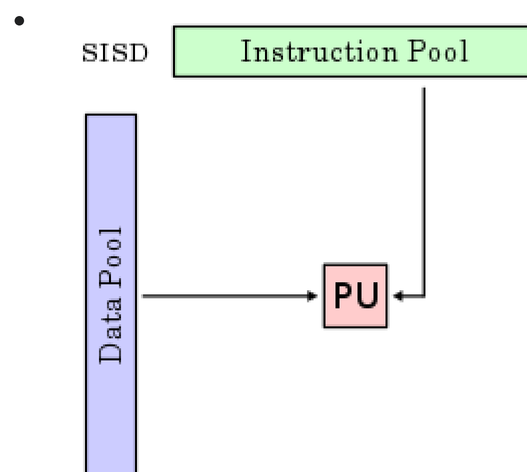
- Multiple instructions operate on one data stream. This is an uncommon architecture which is generally used for fault tolerance.
- Heterogeneous systems operate on the same data stream and must agree on the result.
- Examples include the Space Shuttle flight control computer.

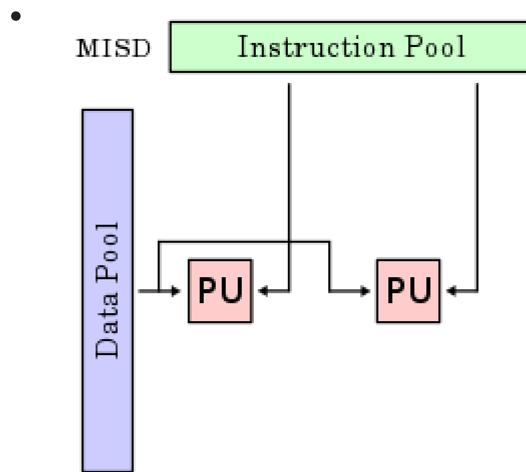
Multiple instruction streams, multiple data streams (MIMD)

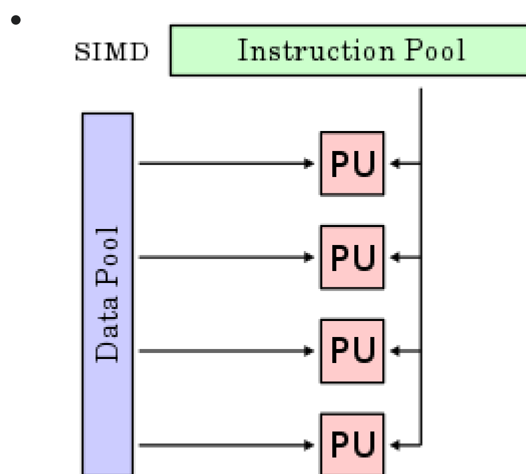
- Multiple autonomous processors simultaneously executing different instructions on different data.
- MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space.

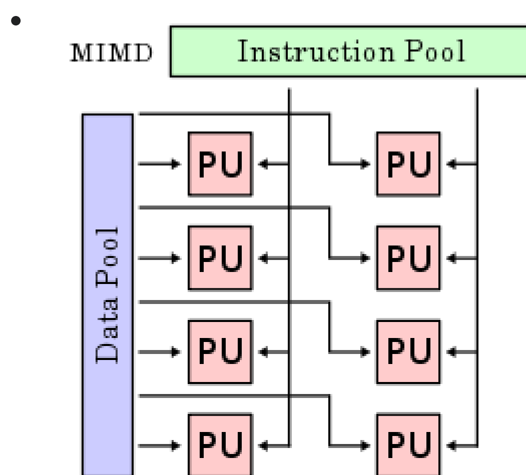
Diagram comparing classifications

These four architectures are shown below visually. Each processing unit (PU) is shown for a uni-core or multi-core computer:

- 

- 

- 

- 

Further divisions

As of 2006, all the top 10 and most of the TOP500 supercomputers are based on a MIMD architecture.

Some further divide the MIMD category into the two categories below, and even further subdivisions are sometimes considered.

Single Program, Multiple Data Streams (SPMD)

- Multiple autonomous processors simultaneously executing the same program (but at independent points, rather than in the lockstep that SIMD imposes) on different data.
- Also termed *single process, multiple data*- the use of this terminology for SPMD is technically incorrect, as SPMD is a parallel execution model and assumes multiple cooperating processors executing a program.
- SPMD is the most common style of parallel programming.
- The SPMD model and the term was proposed by Frederica Darema of the RP3 team.

Multiple programs, multiple data streams (MPMD)

- Multiple autonomous processors simultaneously operating at least 2 independent programs.
- Typically such systems pick one node to be the "host" ("the explicit host/node programming model") or "manager" (the "Manager/Worker" strategy), which runs one program that farms out data to all the other nodes which all run a second program.
- Those other nodes then return their results directly to the manager. An example of this would be the Sony PlayStation 3 game console, with its SPU/PPU processor.

OpenMP

- Application Program Interface (API),
- Jointly defined by a group of major computer hardware and software vendors.
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications.
- The API supports C/C++ and Fortran on a wide variety of architectures

AL-MAMOON UNIVERSITY COLLEGE

Computer Science Department

3 rd Class

Parallel Programming

Dr Taif Sami

Lecture 4

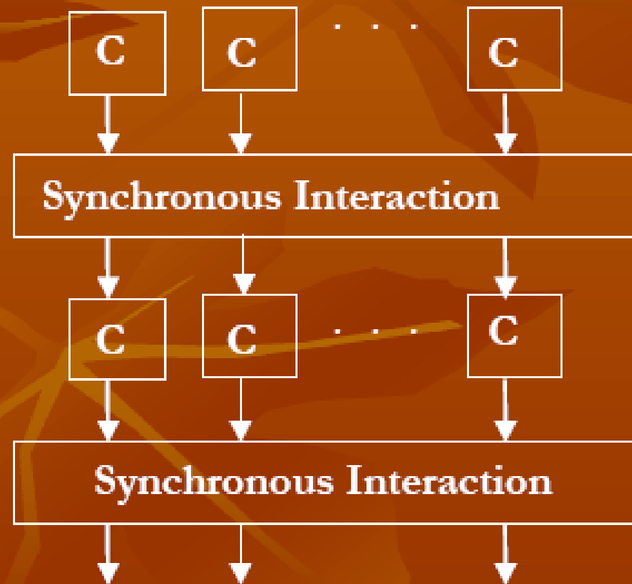
2020-2021

Parallel Programming Paradigm

- Phase parallel
- Divide and conquer
- Pipeline
- Process farm
- Work pool
- Remark :

The parallel program consists of number of super steps, and each super step has two phases : *computation phase and interaction phase*

Phase Parallel Model



- The phase-parallel model offers a paradigm that is widely used in parallel programming.
- The parallel program consists of a number of supersteps, and each has two phases.
- In a computation phase, multiple processes each perform an independent computation *C*.
- In the subsequent interaction phase, the processes perform one or more synchronous interaction operations, such as a barrier or a blocking communication.
- Then next superstep is executed.

Phase Parallel Model

A special case of Phase-Parallel Paradigm is Synchronous Iteration Paradigm where the supersteps are a sequence of iterations in a loop.

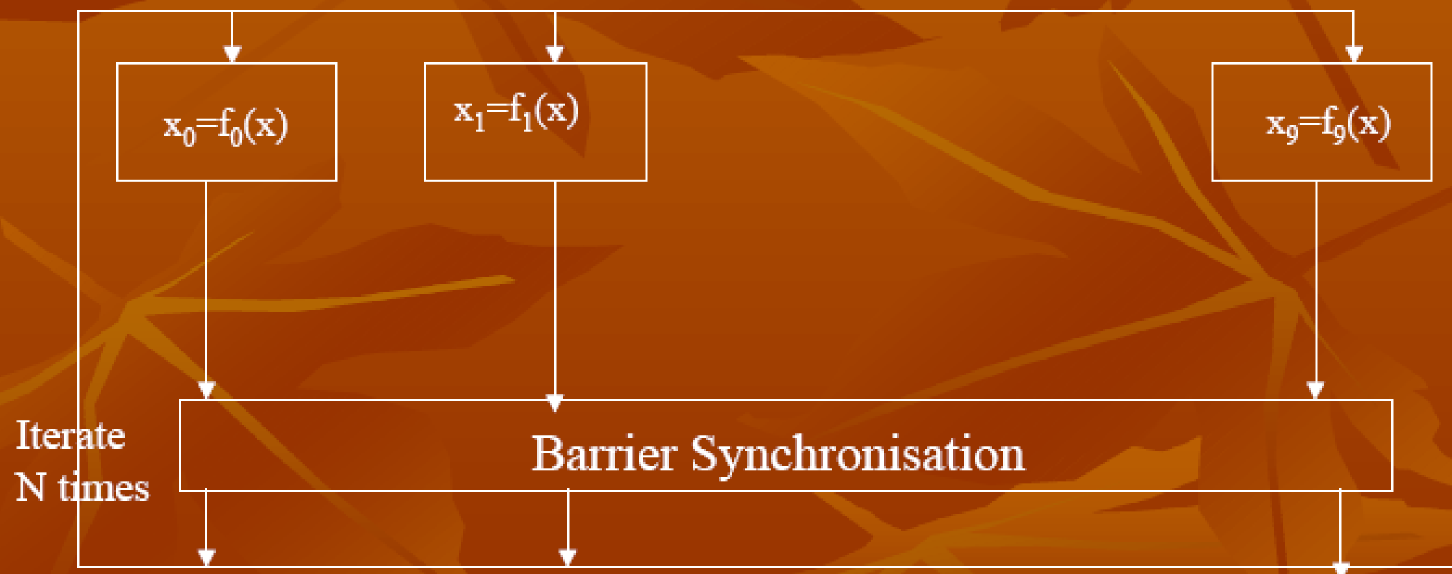
Consider the example of computing x=f(x) where x is an n-dimensional vector.

Synchronous Iteration Paradigm

```
parfor (i=0; i<n; i++)   //create n processes
                           //each executing a for loop
{
  for (j=0; j<N; j++)
    {
      x[i] = f_i(x);
      barrier;
    }
}
```

Synchronous Iteration Paradigm

For n = 9 we have



Asynchronous Iteration Paradigm

```
parfor (i=0; i<n; i++)
{
  for (j=0; j<N; j++)
    x[i] = f_i(x);
  }
```

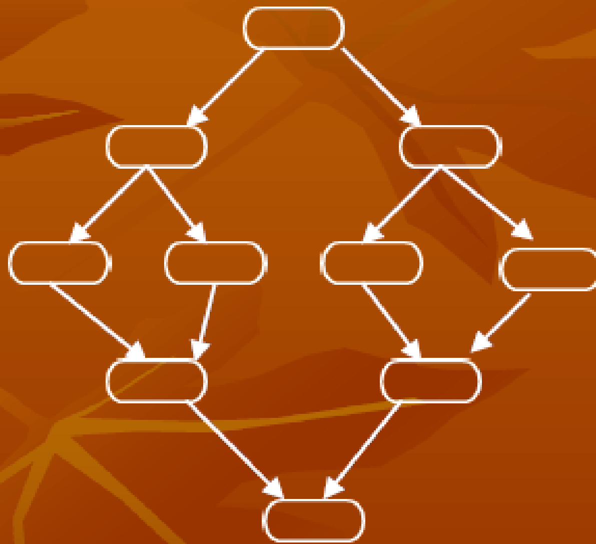
It allows a process to proceed to the next iteration, without waiting for the remaining processes to catch up.

Asynchronous Iteration Paradigm

The above code could be indeterminate, because when a process is computing x[i] in the jth iteration, the x[i-1] value used could be computed by another process in iteration j-1.

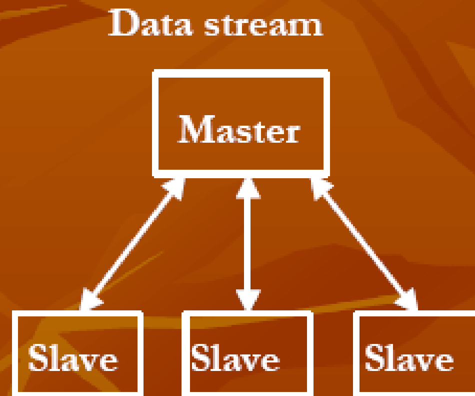
However, under certain conditions, an asynchronous iteration algorithm will converge to the correct results and is faster than the synchronous iteration algorithm.

Divide and Conquer



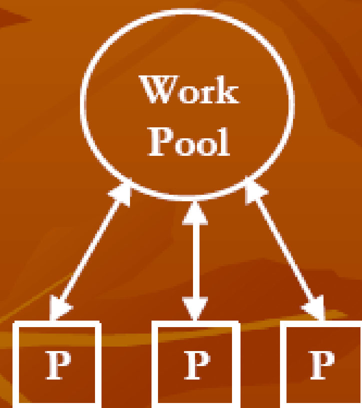
- A parent process divides its workload into several smaller pieces and assigns them to a number of child processes.
- The child processes then compute their workload in parallel and the results are merged by the parent.
- The dividing and the merging procedures are done recursively.
- This paradigm is very natural for computations such as quick sort. Its disadvantage is the difficulty in achieving good load balance.

Process Farm



- This paradigm is also known as the master-slave paradigm.
- A master process executes the essentially sequential part of the parallel program and spawns a number of slave processes to execute the parallel workload.
- When a slave finishes its workload, it informs the master which assigns a new workload to the slave.
- This is a very simple paradigm, where the coordination is done by the master.

Work Pool



- This paradigm is often used in a shared variable model.
- A pool of works is realized in a global data structure.
- A number of processes are created. Initially, there may be just one piece of work in the pool.
- Any free process fetches a piece of work from the pool and executes it, producing zero, one, or more new work pieces put into the pool.
- The parallel program ends when the work pool becomes empty.
- This paradigm facilitates load balancing, as the workload is dynamically allocated to free processes.

Parallel Programming Models

Implicit parallelism

- If the programmer does not explicitly specify parallelism, but let the compiler and the run-time support system automatically exploit it.

Explicit Parallelism

- It means that parallelism is explicitly specified in the source code by the programming using special language constructs, complex directives, or library cells.

20

Implicit Parallel Programming Models

Implicit Parallelism: Parallelizing Compilers

- Automatic parallelization of sequential programs
  - Dependency Analysis
  - Data dependency
  - Control dependency

Remark

Users belief is influenced by the currently disappointing performance of automatic tools (Implicit parallelism) and partly by a theoretical results obtained

Implicit Parallel Programming Models

Implicit Parallelism

- Bernstein's Theorem
  - It is difficult to decide whether two operations in an imperative sequential program can be executed in parallel
  - An implication of this theorem is that there is no automatic technique, compiler time or runtime that can exploit all parallelism in a sequential program

Implicit Parallel Programming Models

To overcome this theoretical limitation, two solutions have been suggested

- The first solution is to abolish the imperative style altogether, and to use a programming language which makes parallelism recognition easier

- The second solution is to use explicit parallelism

Explicit Parallel Programming Models

Three dominant parallel programming models are :

- Data-parallel model
- Message-passing model
- Shared-variable model

25

Explicit Parallel Programming Models

Main Features	Data-Parallel	Message-Passing	Shared-Variable
Control flow (threading)	Single	Multiple	Multiple
Synchrony	Loosely synchronous	Asynchronous	Asynchronous
Address space	Single	Multiple	Multiple
Interaction	Implicit	Explicit	Explicit
Data allocation	Implicit or semiexplicit	Explicit	Implicit or semiexplicit

26

Explicit Parallel Programming Models

<u>Message – Passing</u>

- Message passing has the following characteristics :
  - Multithreading
  - Asynchronous parallelism (MPI reduce)
  - Separate address spaces (Interaction by MPI/PVM)
  - Explicit interaction
  - Explicit allocation by user

Explicit Parallel Programming Models

<u>Message – Passing</u>

- Programs are multithreading and asynchronous requiring explicit synchronization
- More flexible than the data parallel model, but it still lacks support for the work pool paradigm.
- PVM and MPI can be used
- Message passing programs exploit large-grain parallelism

Explicit Parallel Programming Models

Shared Variable Model

- It has a single address space (Similar to data parallel)
- It is multithreading and asynchronous (Similar to message-passing model)
- Data resides in single shared address space, thus does not have to be explicitly allocated
- Workload can be either explicitly or implicitly allocated
- Communication is done implicitly through shared reads and writes of variables. However synchronization is explicit

Explicit Parallel Programming Models

Shared variable model

- The shared-variable model assumes the existence of a single, shared address space where all shared data reside
- Programs are multithreading and asynchronous, requiring explicit synchronizations
- Efficient parallel programs that are loosely synchronous and have regular communication patterns, the shared variable approach is not easier than the message passing model

Other Parallel Programming Models

- Functional programming
- Logic programming
- Computing by learning
- Object oriented programming

31

Thank You

AL-MAMOON UNIVERSITY COLLEGE

Computer Science Department

Parallel Programming

Lecture 5

2020-2021

1. Basic Concepts

Single core processor: is a microprocessor with a single core in one chip, running a single thread in one time.

Multi-core processor: is a microprocessor with multiple cores in one chip, running multiple threads in one time.

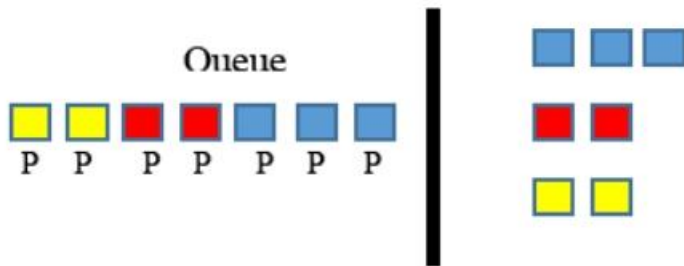


Figure 1 Parallel Computing example

Parallel Computing: is a type of computation to proceed many calculations in simultaneously. As a benefit,
is often used to divide larger problems in smaller once, which can then be solved at the same time (see fig1).

Process(Computing): is the instance of a running program that is being executed by one or many threads. Each process is an independent entity to which system resources such as CPU time, memory, etc. are allocated and each process is executed in a separate address space.

When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data. The following image shows a simplified layout of a process inside main memory (see fig2).

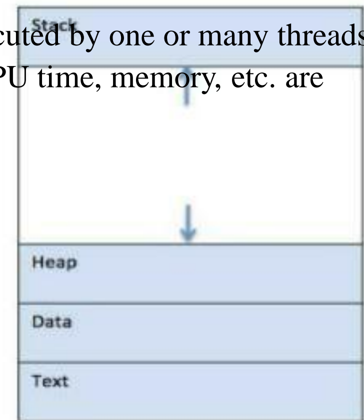


Figure 2 Process layout inside memory.

Component & Description

1. Stack The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2. Heap This is dynamically allocated memory to a process during its run time.
3. Text This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4. Data This section contains the global and static variables.

Program: is a piece of code which may be a single line or millions of lines.

```
#include <stdio.h>

int main() {
   printf("Hello, World! \n");
   return 0;
}
```

Process Life Cycle

When a process executes, it passes through different states. In general, a process can have one of the following five states at a time (see fig 3).

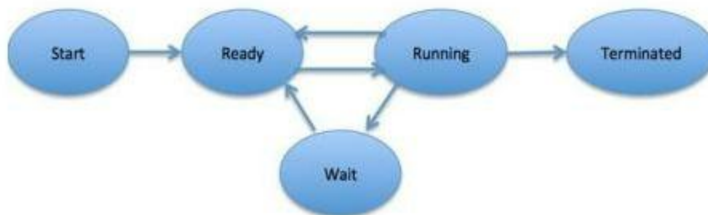


Figure 3 Process life cycle

Thread: is the smallest sequence of programmed instructions that can be managed independently by a scheduler. Multiple threads can exist within one process (see fig 4).

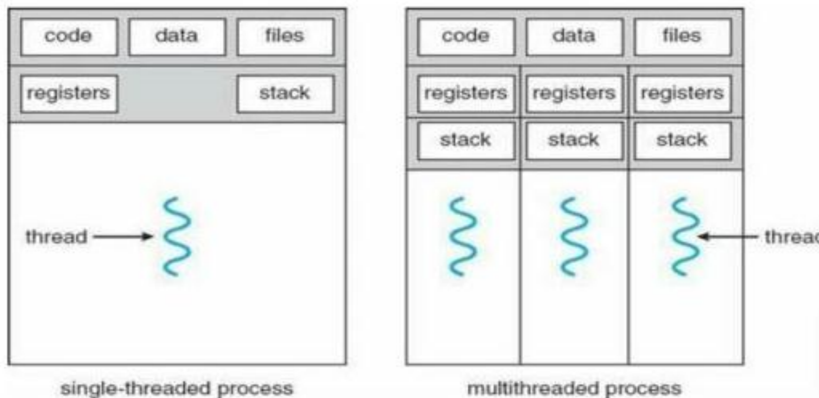


Figure 4 Process with single and multithread example

Task: is a unit of execution or a unit of work, that occurs during the first step of executing parallel program.

A first step in designing the parallel program is to break the problem into separate "chunks or pieces" of work that can be distributed to multiple tasks.

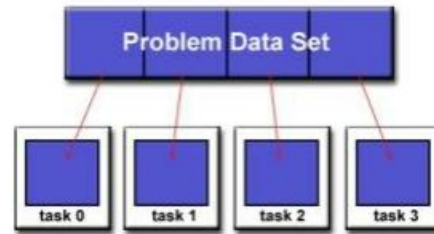
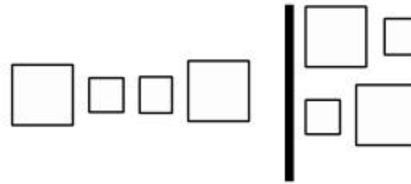


Figure 5 Tasks description

Task granularity: The size of tasks (e.g., in terms of the number of instructions) and there is typically the possibility of choosing tasks of different sizes.



Scheduling: is the assignment of tasks to processes or threads with a fixed order to execute the tasks. Scheduling can be done by hand in the source code or by the programming environment, at compile time or dynamically at runtime.

Mapping: is the assignment of processes or threads onto the physical units, processors or cores, and it is usually done by the runtime system but can sometimes be influenced by the programmer.

Note: To execute any parallel programs correctly need to consider the synchronization and coordination of threads and processes.

Parallel execution time: is the time elapsed between the start of the application on the first processor and the end of the execution of the application on all processors. computation time on processors or cores and synchronization time for data exchange are important aspects in parallel computing.

The parallel execution time should be smaller than the sequential execution time on one processor so that designing a parallel program is worth the effort.

Parallel program must have →parallel execution time $<$ (Less than) sequential execution time

Idle times: is an unproductive time where the processor cannot do anything useful, only wait for an event to happen.

Load balancing: is good synchronization and mapping strategy to the parallel program with small idle time. This occurs when the work load is assigned equally to processors or cores.

2. Types of Parallel Programming

Parallel processing can be subdivided into two groups as explained below.

a) Task based: In this type every processor in the system will execute different jobs or tasks in a parallel way. For example, in MS Word, doing two tasks (printing and Spell Checking) at the same time considered as separated tasks in different processors.
b) Data based: In this type, each processor in the system will perform different data sets in parallel way. For example, in image processing, convert the image color to grayscale by dividing the image into upper and lower halves and each half converted by different processors.

3. Shared memory or distributed memory models

In the shared memory; in case of desktop or laptop computers; there is only one memory unit and multiprocessing units, all processors in this model have the right access to the memory unit which is shared between them.

In the distributed memory; as in the computer cluster; every multiple processing unit have their own memory unit to store the data. In this model, the information is passed between these units.

The computer clusters; are separate units of computers that are connected through the network, working as a single system.

Thus, the multiprocessors can be divided into two shared-memory model as main categories- UMA (Uniform Memory Access), NUMA (Non-uniform Memory Access) as illustrated in Fig 6.

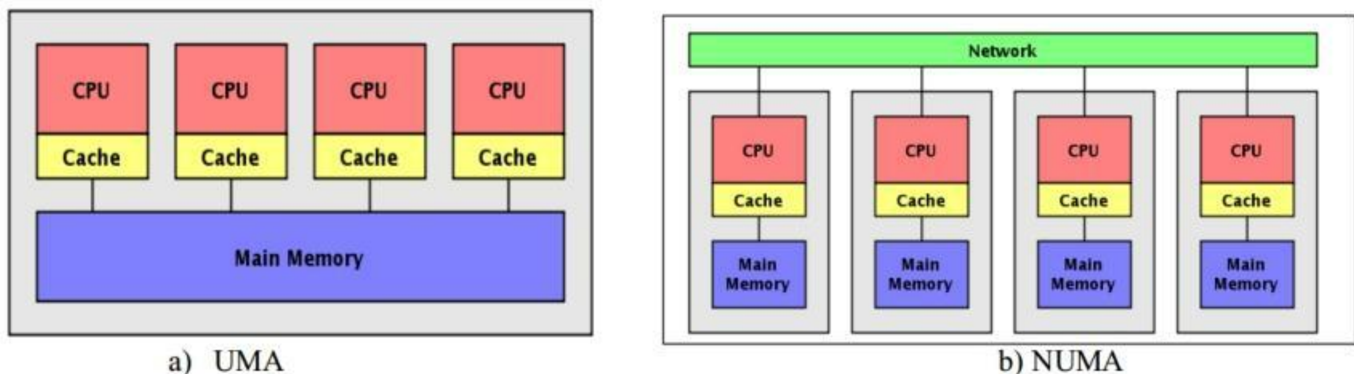


Figure 6 Shared (UMA) and distributed (NUMA) memory models

Each processor needs its own private cache to be fed quickly.

4. Flynn's Taxonomy

First proposed by Michael Flynn in 1966, Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of synchronized instruction (single or multiple) and data streams (single or multiple) available in the architecture.

The four categories in Flynn's taxonomy are the following:

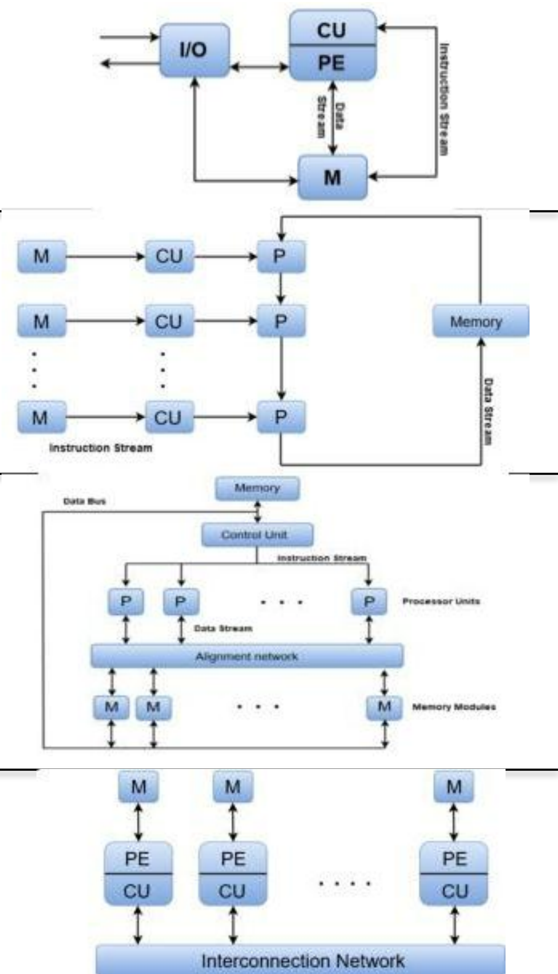
a) Single instruction, Single data (SISD): a single instruction, operating on a single data stream. All the instructions and data to be processed have to be stored in primary memory.

b) Multiple instruction, Single data (MISD): multiple instructions operate on one data stream. Each processing unit operates on the data independently via separate instruction stream.

MISD structure is only of theoretical interest since no practical system has been constructed using this organization.

c) Single instruction, Multiple data (SIMD): a single instruction operates on multiple different data streams. All processors receive the same instruction from the control unit but operate on different items of data. The shared memory unit must contain multiple modules so that it can communicate with all the processors simultaneously.

d) Multiple instruction, Multiple data (MIMD): multiple autonomous processors simultaneously executing different instructions on different data. In MIMD, each processor has a separate program and an instruction stream is generated from each program.



Where, CU = Control Unit, PE = Processing Element, M = Memory

The motivations of using parallel processing can be summarized as follows:

- Faster execution time, parallel processing allowing the execution of applications in a shorter time.
- Higher computational power, to solve larger problems in a short point of time.
- Higher throughput, to solve many problems or large problems in short time.
- Working on different tasks simultaneously, you can do many things simultaneously by using multiple computing resources.

5. Effectiveness of Parallel processing

Throughout the book, we will be using certain measures to compare the effectiveness of various parallel algorithms or architectures for solving desired problems. The following definitions and notations are applicable.

p	Number of processors
$W(p)$	Total number of unit operations performed by the p processors; this is often referred to as computational work or energy
$T(p)$	Execution time with p processors; clearly, $T(1) = W(1)$ *and* $T(p) \leq W(p)$
$S(p)$	$Speed-up = \frac{T(1)}{T(p)}$
E(p)	$Efficiency = \frac{T(1)}{pT(p)}$
$R(p)$	$Redundancy = \frac{W(p)}{W(1)}$
$U(p)$	$Utilization = \frac{W(p)}{pT(p)}$
$Q(p)$	$Quality = \frac{T^3(1)}{pT^2(p)W(p)}$

Example: Finding the sum of 16 numbers can be represented by the binary-tree computation graph of Figure 4 with $T(1) = W(1) = 15$. Assume unit-time additions and ignore all else. With $p = 8$ processors, we have

W(8) = 15 T(8) = 4
E(8) = 15/(8 · 4) = 47%
S(8) = 15/4 = 3.75
R(8) = 15/15 = 1
Q(8) = 1.76

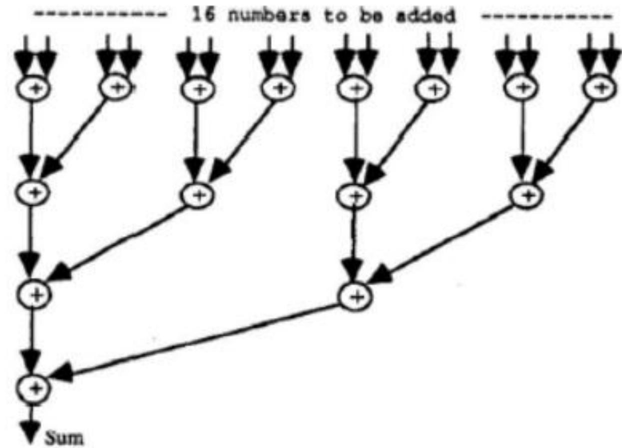


Figure 7 Computation graph for finding the sum of 16 numbers

Essentially, the 8 processors perform all of the additions at the same tree level in each time unit, beginning with the leaf nodes and ending at the root. The relatively low efficiency is the result of limited parallelism near the root of the tree.

Speedup

Speedup is the expected performance benefit from running an application on a multicore versus a single-core machine. When speedup is measured, single-core machine performance is the baseline. For example, assume that the duration of an application on a single-core machine is six hours. You might expect that an application running on a single-core machine would run twice as quickly on a dual-core machine, and that a quad core machine would run the application four times as fast. But that's not exactly correct.

Here are some of the limitations to linear speedup (full parallelization) of parallel code:
Serial code

- Overhead from parallelization
- Synchronization
- Sequential input/output

Predicting speedup is important in designing, benchmarking, and testing your parallel application. Fortunately, there are formulas for calculating speedup. One such formulas are Amdahl's Law and Gustafson's Law.

- **Amdahl's Law**

It 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

$$Speedup = \frac{1}{1 - p(\frac{P}{N})}$$

P : is the percent of the application that runs in parallel.
N : is the number of processor cores.

Example: 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).

$$Speedup = \frac{1}{1 - 0.75(\frac{0.75}{N3})} = 2$$

The application will run twice as fast on a three processor-core machine.

❖ Gustafson's Law

You may have an application that's split consistently into a sequential and parallel portion. Amdahl's Law maintains these proportions as additional processors are added. The serial and parallel portions each remain half of the program. But in the real world, as computing power increases.

Amdahl's Law does not account for the overhead required to schedule, manage, and execute parallel tasks. Gustafson's Law takes both of these additional factors into account. Here is the formula to calculate speedup by using Gustafson's Law.

$$Speedup = \frac{S + N(1 - S)}{S + (1 - S)} - O_n$$

In the above formula, S is the percentage of the serial code in the application, N is the number of processor cores, and O_n is the overhead from parallelization.

Also speedup can be computed using the bellow equation

Speedup = s + p x N
s--> time spend in executing serial
p--> time spend in executing parallel
N--> no of processor
also s + p = 1