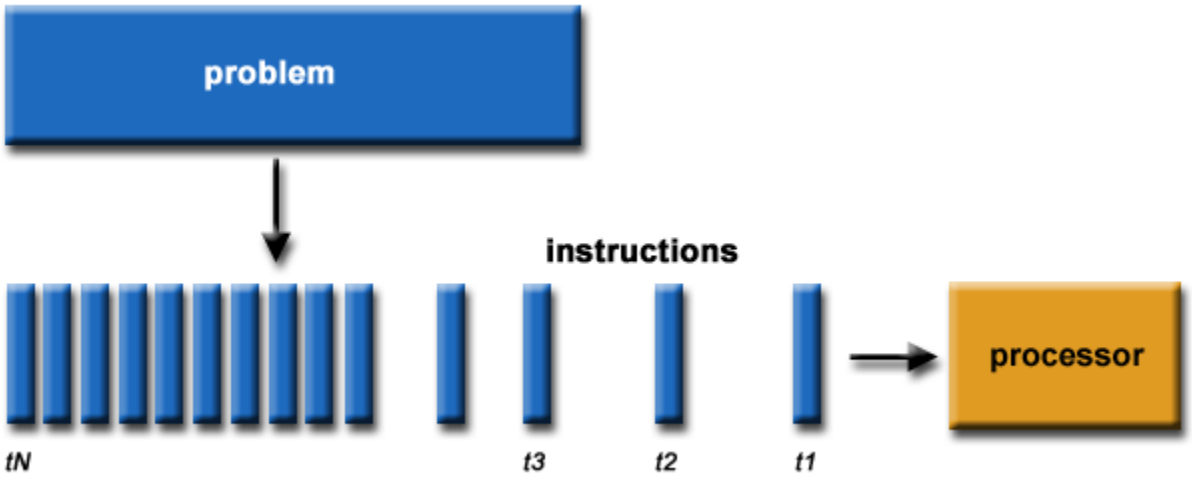
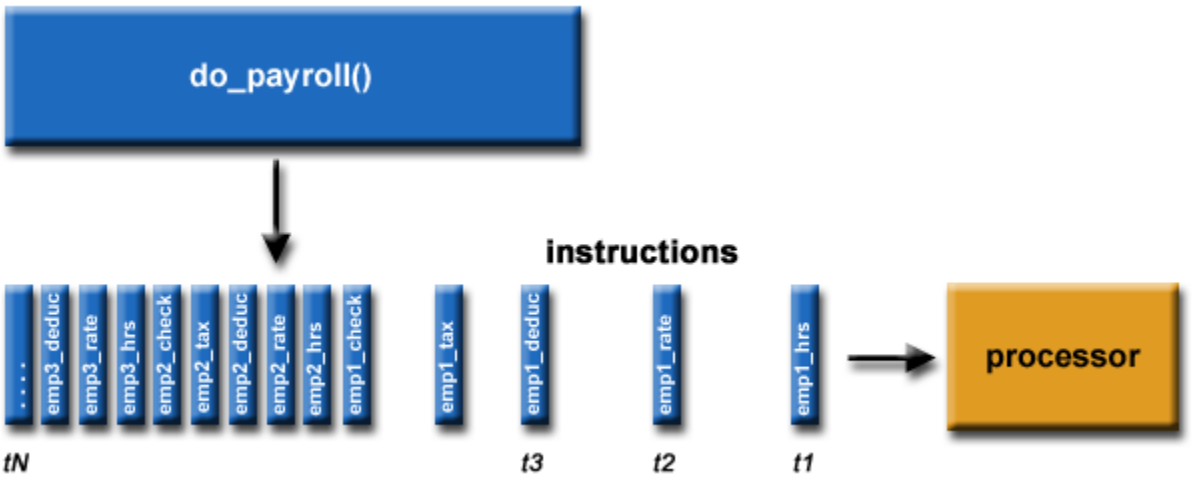
What is Parallel Computing?

Serial Computing:

- Traditionally, software has been written for ***serial*** computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time



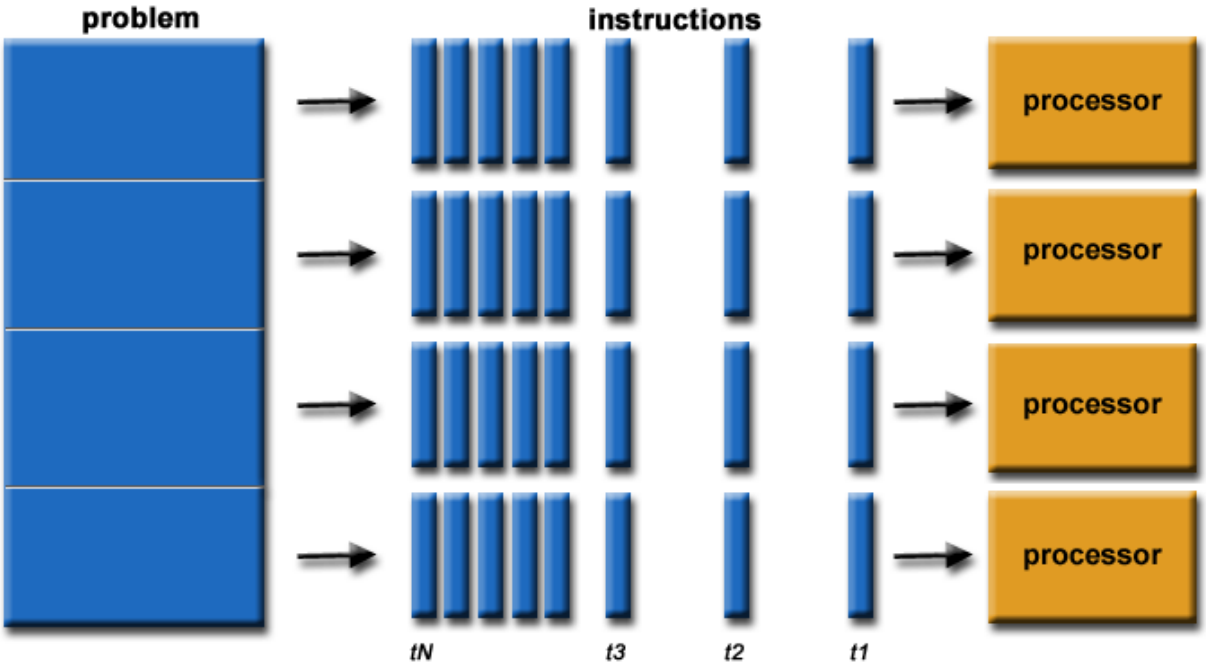
For example:



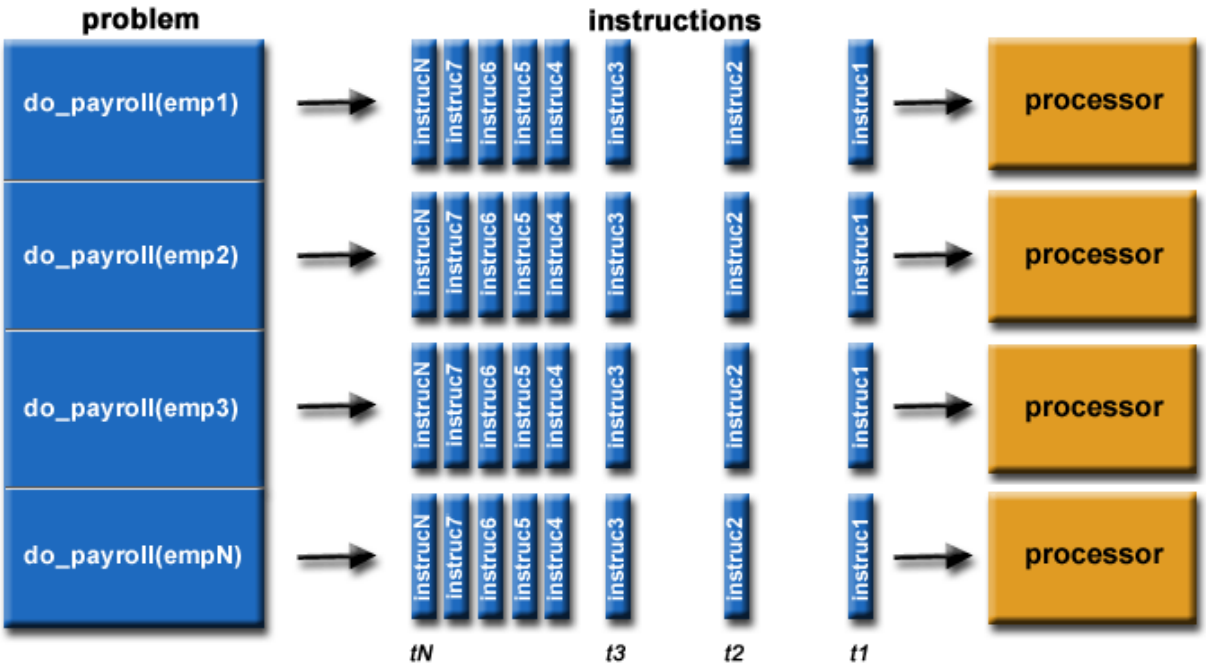
Parallel Computing:

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:

- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed



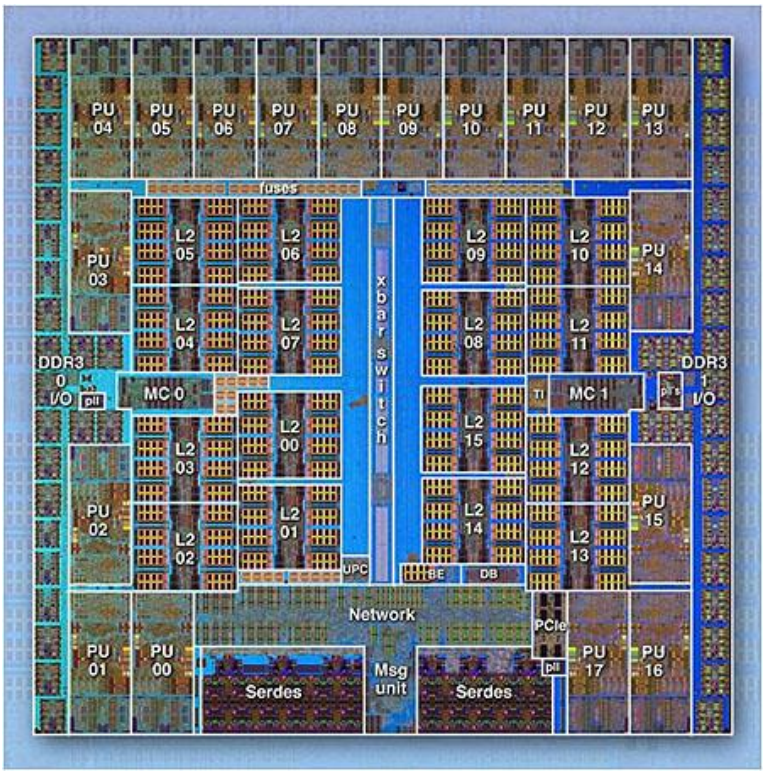
For example:



- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

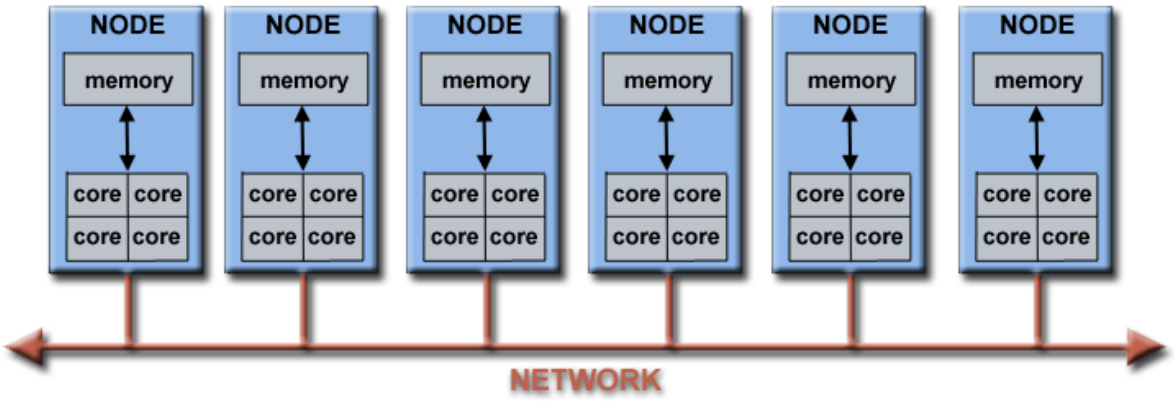
Parallel Computers:

- Virtually all stand-alone computers today are parallel from a hardware perspective:
  - Multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.)
  - Multiple execution units/cores
  - Multiple hardware threads

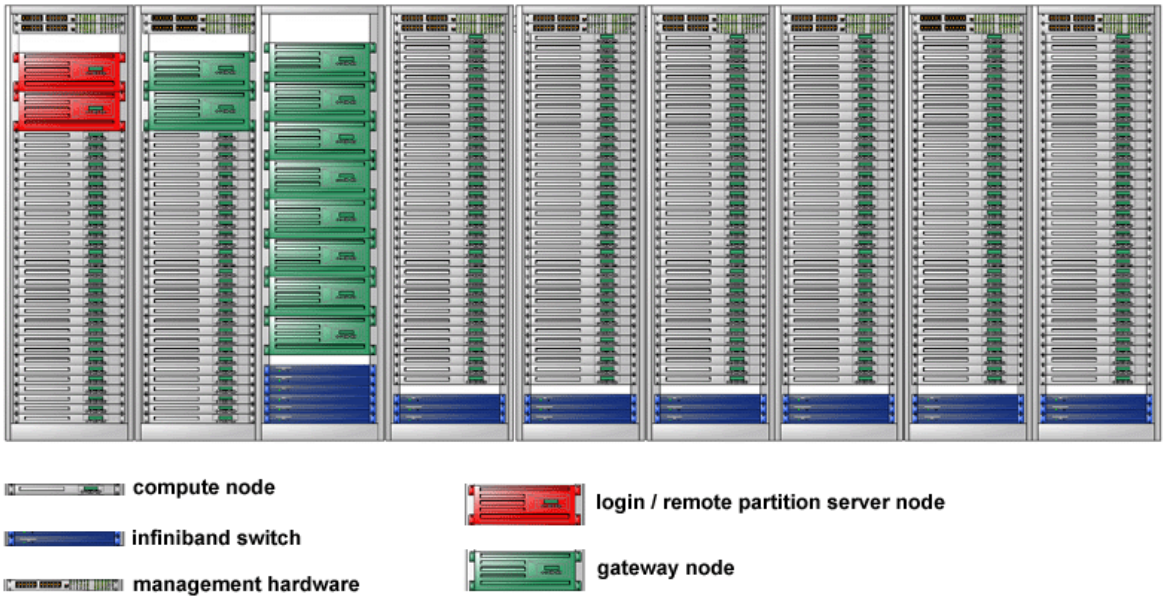


IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



- For example, the schematic below shows a typical LLNL parallel computer cluster:
  - Each compute node is a multi-processor parallel computer in itself
  - Multiple compute nodes are networked together with an Infiniband network
  - Special purpose nodes, also multi-processor, are used for other purposes

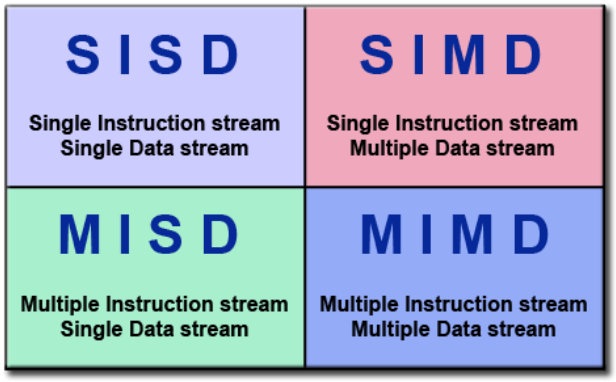


- The majority of the world's large parallel computers (supercomputers) are clusters of hardware produced by a handful of (mostly) well known vendors.

Concepts and Terminology

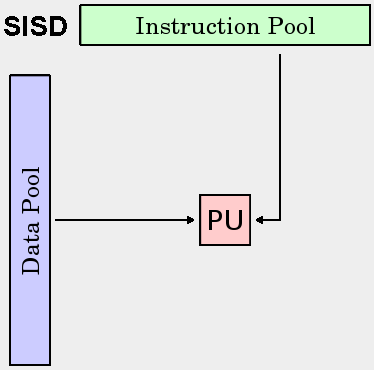
Flynn's Classical Taxonomy

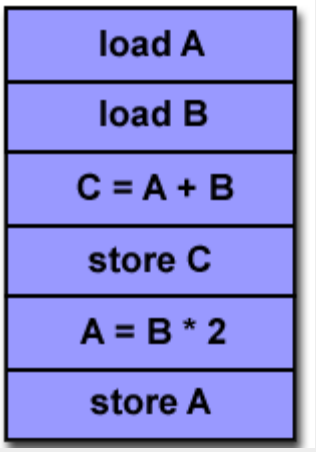
- There are different ways to classify parallel computers. Examples available HERE.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.
- The matrix below defines the 4 possible classifications according to Flynn:



Single Instruction, Single Data (SISD):

- A serial (non-parallel) computer
- **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.





UNIVAC1

IBM 360

CRAY1

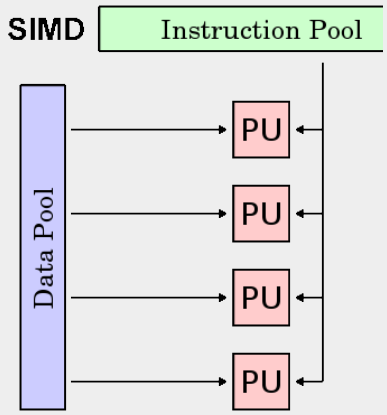
CDC 7600

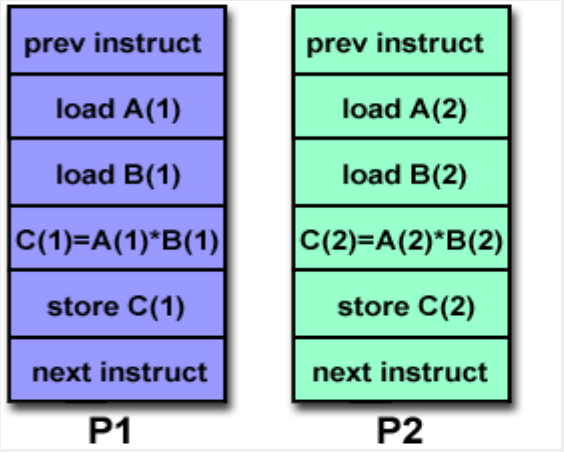
PDP1

Dell Laptop

Single Instruction, Multiple Data (SIMD):

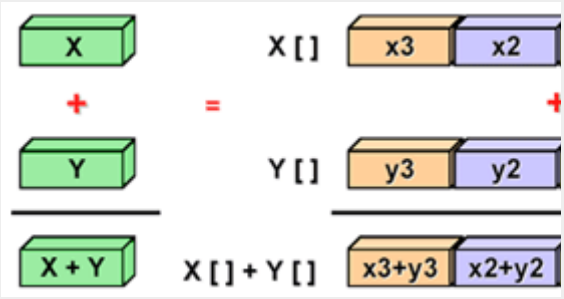
- A type of parallel computer
- **Single Instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple Data:** Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
    - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
    - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.





ILLIAC IV

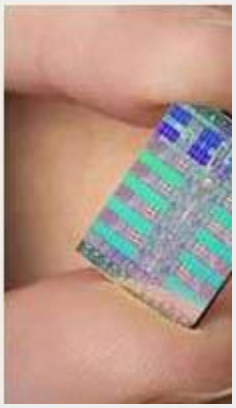
MasPar



Cray X-MP

Cray Y-MP

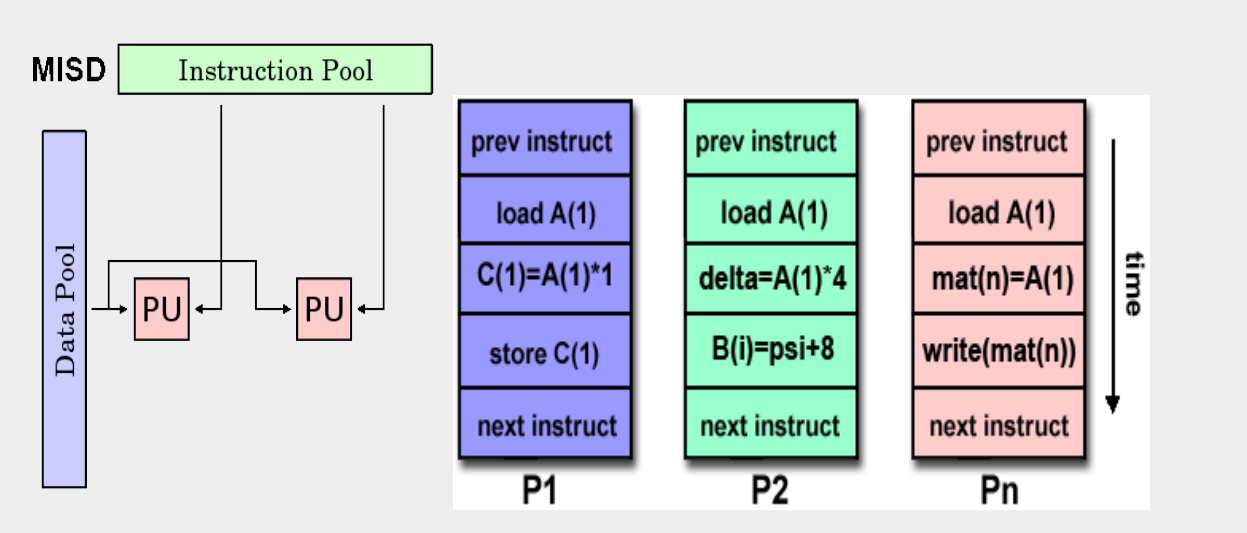
Thinking Machines CM-2



Cell Processor (GPU)

Multiple Instruction, Single Data (MISD):

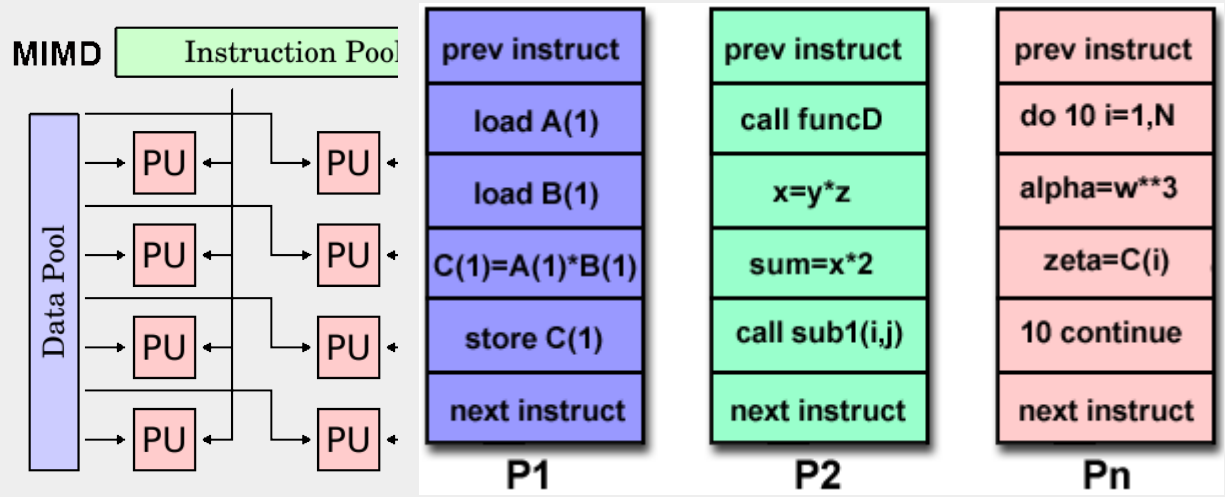
- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.



Multiple Instruction, Multiple Data (MIMD):

- A type of parallel computer
- **Multiple Instruction:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.

- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components



IBM POWER5

HP/Compaq Alphaserver

Intel IA32

AMD Opteron

Cray XT3

IBM BG/L

Concepts and Terminology

Some General Parallel Terminology

- Like everything else, parallel computing has its own "jargon". Some of the more commonly used terms associated with parallel computing are listed below.
- Most of these will be discussed in more detail later.

Supercomputing / High Performance Computing (HPC)

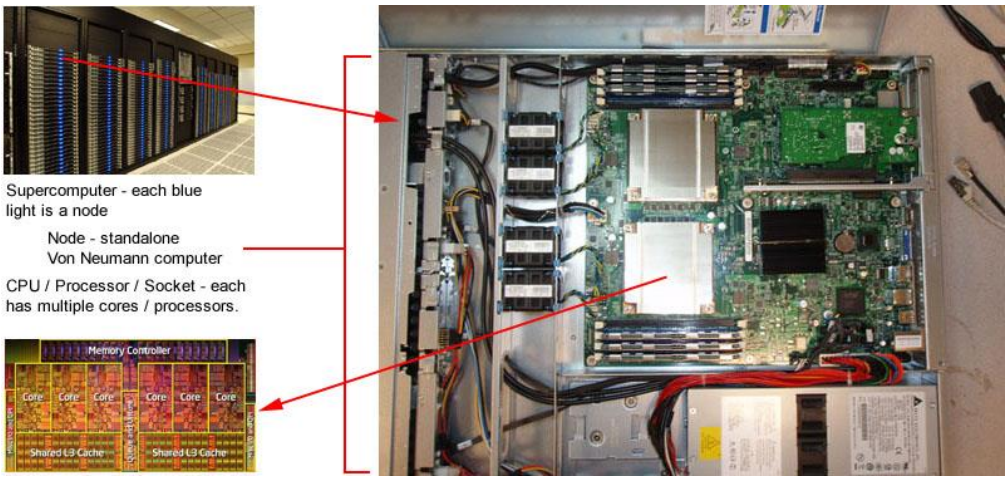
Using the world's fastest and largest computers to solve large problems.

Node

A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer.

CPU / Socket / Processor / Core

This varies, depending upon who you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confused at times. Wonder why?



Task
A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

Pipelining
Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Shared Memory
From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP)
Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Distributed Memory
In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communications
Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization
The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Granularity
In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

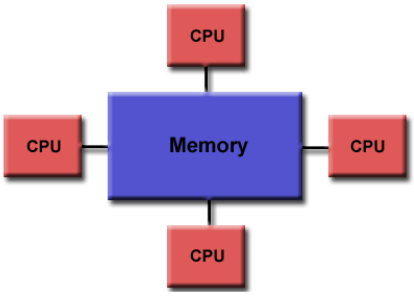
- ***Coarse:*** relatively large amounts of computational work are done between communication events
- ***Fine:*** relatively small amounts of computational work are done between communication events

Parallel Computer Memory Architectures

Shared Memory

General Characteristics:

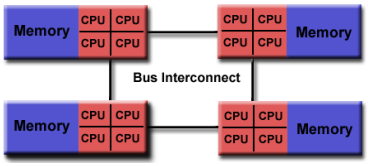
- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
- Multiple processors can operate independently but share the same memory resources.
- Changes in a memory location effected by one processor are visible to all other processors.
- Historically, shared memory machines have been classified as ***UMA*** and ***NUMA***, based upon memory access times.



Shared Memory (UMA)

Uniform Memory Access (UMA):

- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know about the update. Cache coherency is accomplished at the hardware level.



Shared Memory (NUMA)

Non-Uniform Memory Access (NUMA):

- Often made by physically linking two or more SMPs
- One SMP can directly access memory of another SMP
- Not all processors have equal access time to all memories
- Memory access across link is slower
- If cache coherency is maintained, then may also be called CC-NUMA - Cache Coherent NUMA

Advantages:

- Global address space provides a user-friendly programming perspective to memory

- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

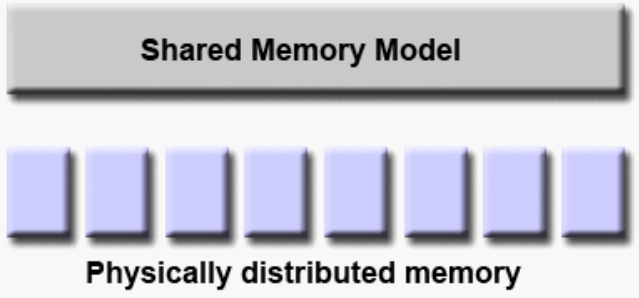
- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Parallel Programming Models

Overview

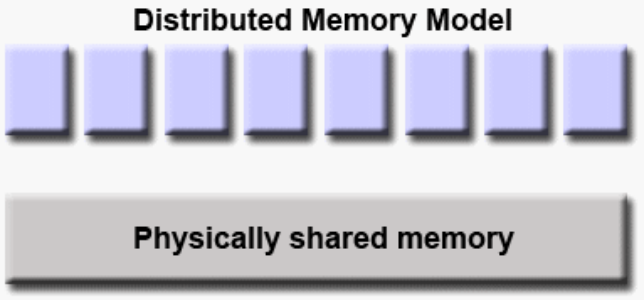
- There are several parallel programming models in common use:
  - Shared Memory (without threads)
  - Threads
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- **Parallel programming models exist as an abstraction above hardware and memory architectures.**
- Although it might not seem apparent, these models are **NOT** specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples from the past are discussed below.

SHARED memory model on a DISTRIBUTED memory machine:
Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across networked machines, but appeared to the user as a single shared memory global address space. Generically, this approach is referred to as "virtual shared memory".



DISTRIBUTED memory model on a SHARED memory machine:
Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of shared memory architecture, where every task has direct access to global address space spread across all machines. However, the ability to send and receive messages

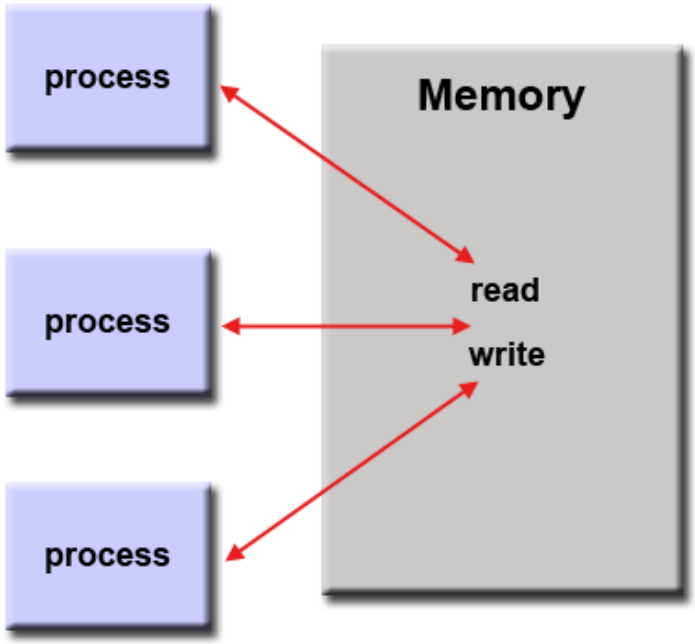
using MPI, as is commonly done over a network of distributed memory machines, was implemented and commonly used.



- **Which model to use?** This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Parallel Programming Models

Shared Memory Model (without threads)



- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see

and have equal access to shared memory. Program development can often be simplified.

- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage ***data locality***:
  - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.

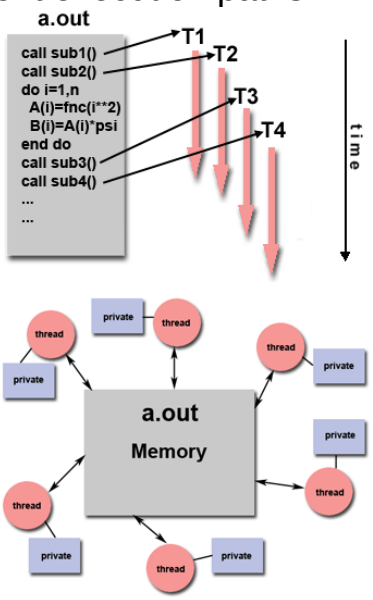
Implementations:

- On stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming. For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc).
- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software. A variety of SHMEM implementations are available: http://en.wikipedia.org/wiki/SHMEM.

Parallel Programming Models

Threads Model

- This programming model is a type of shared memory programming.
- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
- For example:
  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.
  - **a.out** performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
  - Each thread has local data, but also, shares the entire resources of **a.out**. This saves the overhead associated with replicating a program's resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of **a.out**.
  - A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
  - Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.



Implementations:

- From a programming perspective, threads implementations commonly comprise:
    - A library of subroutines that are called from within parallel source code
    - A set of compiler directives imbedded in either serial or parallel source code

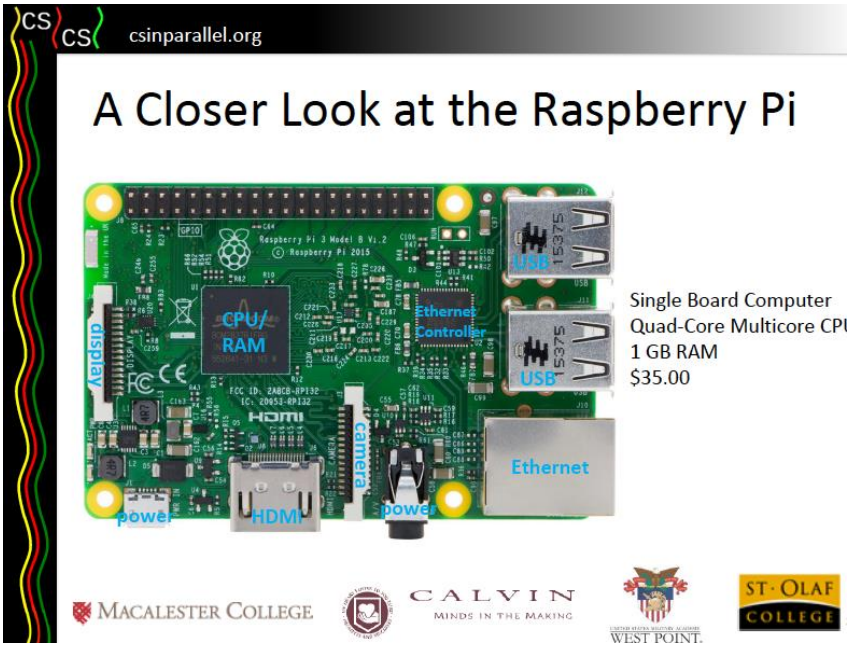
  In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).

- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.
- **POSIX Threads**
    - Specified by the IEEE POSIX 1003.1c standard (1995). C Language only.
    - Part of Unix/Linux operating systems
    - Library based
    - Commonly referred to as Pthreads.
    - Very explicit parallelism; requires significant programmer attention to detail.
- **OpenMP**
    - Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals.
    - Compiler directive based
    - Portable / multi-platform, including Unix and Windows platforms
    - Available in C/C++ and Fortran implementations
    - Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.
- Other threaded implementations are common, but not discussed here:
    - Microsoft threads
    - Java, Python threads
    - CUDA threads for GPUs

More Information:

- POSIX Threads tutorial: computing.llnl.gov/tutorials/pthreads

- OpenMP tutorial: computing.llnl.gov/tutorials/openMP

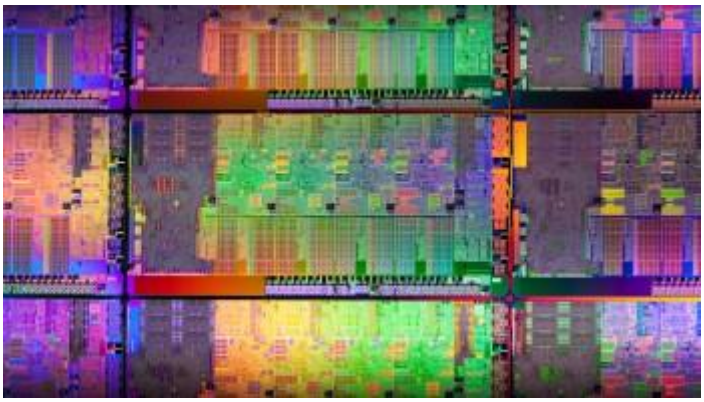


The Raspberry Pi does not have a separate **CPU (Central Processing Unit)**, **RAM (Random Access Memory)** or **GPU (Graphics Processing Unit)**. Instead they are all squeezed into one component called a **System on Chip or SoC unit**. This is essentially the entire computer on one chip.

The Raspberry Pi uses an **ARM1176JZF-S 700MHz CPU** which is also installed in a wide variety of mobile phones, hand held games consoles and eReaders. This CPU is **single core**, however it does have a **co-processor** to perform **floating point calculations**. Many calculations required by a program involve whole numbers (integers). These are easier for the CPU to handle. Integer calculations produce accurate results. Floating point or real numbers have a fractional part e.g. 1.5. They are more demanding for the CPU to process.

The **Model B Raspberry Pi** has **512MB SDRAM (Synchronous Dynamic RAM)**. This is working memory that is used to store programs that are currently being run in the CPU.

SoC vs. CPU – The battle for the future of computing



After more than 50 years at the top of the heap, the CPU finally has some competition from an upstart called the SoC. For decades, you could walk into a shop and confidently pick out a new computer based on its CPU — and now, everywhere you look, from smartphones to tablets and even some laptops, there are SoCs.

Don’t worry, though, CPUs and SoCs are actually rather similar, and almost everything you know about CPUs can also be applied to SoCs.

What is a CPU?

Despite the huge emphasis put on CPU technology and performance, it is ultimately a very fast calculator. It fetches data from memory, and then performs some kind of arithmetic (add, multiply) or logical (and, or, not) operation on that data. The more expensive/complex the CPU, the more data it can process, the faster your computer.

A CPU itself is not a personal computer, however — a whole framework of other silicon chips is required for that. There must be memory to hold the data, an audio chip to decode and amplify your music, a graphics processor to draw pictures on your monitor, and hundreds of smaller components that all have a very important task.

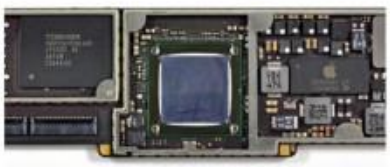
What is an SoC?

An SoC, or system-on-a-chip to give its full name, integrates almost all of these components into a single silicon chip. Along with a CPU, an SoC usually contains a GPU (a graphics processor), memory, USB controller, power management circuits, and wireless radios (WiFi, 3G, 4G LTE, and so on). Whereas a CPU cannot function without dozens of other chips, it's possible to build complete computers with just a single SoC.

The difference between an SoC and CPU

The number one advantage of an SoC is its size: An SoC is only a little bit larger than a CPU, and yet it contains a lot more functionality. If you use a CPU, it's very hard to make a computer that's smaller than 10cm (4 inches) squared, purely because of the number of individual chips that you need to squeeze in. Using SoCs, we can put complete computers in smartphones and tablets, and still have plenty of space for batteries.

Due to its very high level of integration and much shorter wiring, an SoC also uses considerably less power — again, this is a big bonus when it comes to mobile computing. Cutting down on the number of physical chips means that it's much cheaper to build a computer using an SoC, too.



Conventional PC motherboard (left) vs. the main iPad 3 circuit board (right). This image is roughly to scale.

The only real disadvantage of an SoC is a complete lack of flexibility. With your PC, you can put in a new CPU, GPU, or RAM at any time — you cannot do the same for your smartphone. In the future you might be able to buy SoCs that you can slot in, but because everything is integrated this will be wasteful and expensive if you only want to add more RAM.

CPUs are on the way out

Ultimately, SoCs are the next step after CPUs. Eventually, SoCs will almost completely consume CPUs. We are already seeing this with AMD's Llano and Intel's Ivy Bridge CPUs, which integrate a memory controller, PCI Express, and a graphics processor onto the same chip. There will always be a market for general purpose CPUs, especially where power and footprint are less of an issue (such as supercomputers). Mobile and wearable devices are the future of computers, though, and so are SoCs.