An Introduction to Parallel Computing on the Raspberry Pi

Suzanne J. Matthews, West Point
Richard A. Brown, St. Olaf
Joel C. Adams, Calvin College
Elizabeth Shoop, Macalester College

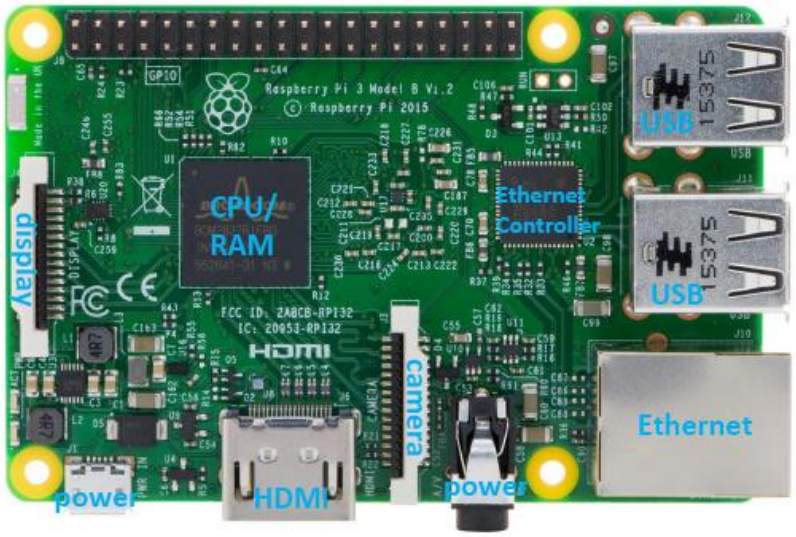
MACALESTER COLLEGE

CALVIN
MINDS IN THE MAKING

WEST POINT

ST · OLAF COLLEGE

A Closer Look at the Raspberry Pi



Single Board Computer
Quad-Core Multicore CPU
1 GB RAM
$35.00

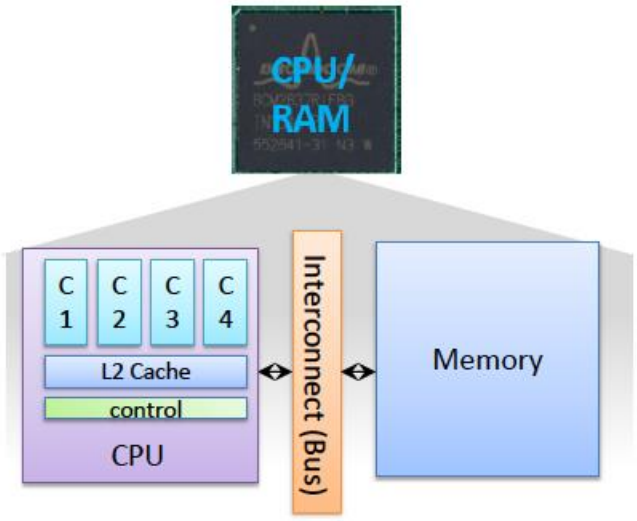
MACALESTER COLLEGE

CALVIN
MINDS IN THE MAKING

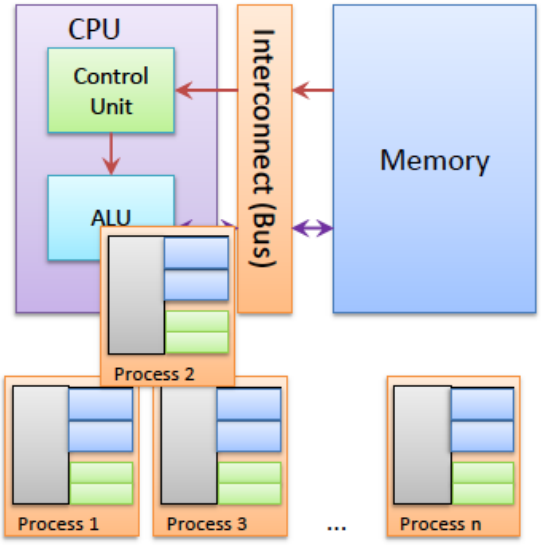
WEST POINT.

ST · OLAF
COLLEGE

What is a Multicore CPU?

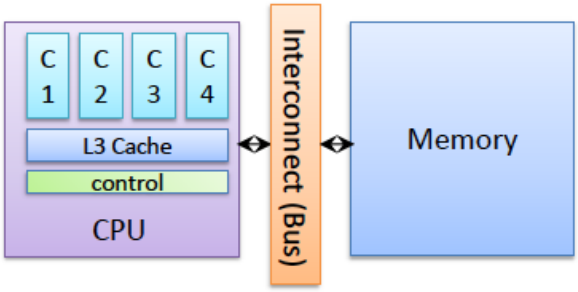


Advantage of Multicore

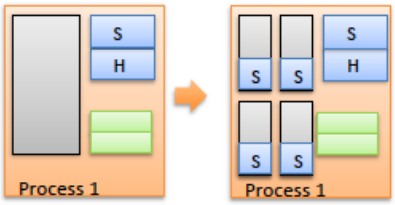


- A process is the abstraction of a running program.
  - Processes do not share memory with each other.
- A single-core CPU only operates on one process at a time.
  - Round-Robin Scheduling Algorithm
- More CPU cores = OS can execute more processes at once! (Concurrency)
  - Increases throughput of system.
  - Does this shorten the amount of time it takes to execute a single process?

Programming Multicore Architectures



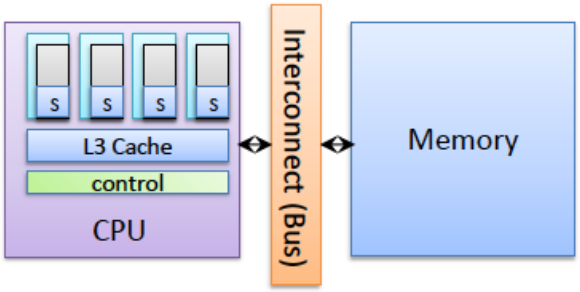
- Thread: a lightweight process that allows a single executable/process to be decomposed to smaller, independent parts.
  - All threads share the common memory of the process they belong to.
- An OS will schedule threads on separate cores/CPUs, as available.



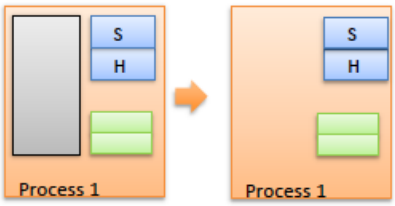
Multithreaded process

csinparallel.org

Programming Multicore Architectures



- Thread: a lightweight process that allows a single executable/process to be decomposed to smaller, independent parts.
  - All threads share the common memory of the process they belong to.
- An OS will schedule threads on separate cores/CPUs, as available.



Multithreaded process

MACALESTER COLLEGE CALVIN MINDS IN THE MAKING WEST POINT ST·OLAF COLLEGE

csinparallel.org

Programming Multicore Architectures

There are many libraries/languages available:

- POSIX Threads
- OpenMP
- C++11 threads, TBB, ...

In this assignment, we will cover OpenMP

- Industry standard since late 1990s.
- Native support with GCC compilers (> 4.3.x)
- Easier to program than POSIX threads.

MACALESTER COLLEGE

CALVIN
MINDS IN THE MAKING

WEST POINT

ST · OLAF
COLLEGE

Shared Memory Parallel Patternlets in OpenMP

When writing programs for shared-memory hardware with multiple cores, a programmer could use a low-level thread package, such as pthreads. An alternative is to use a compiler that processes OpenMP *pragmas*, which are compiler directives that enable the compiler to generate threaded code. Whereas pthreads uses an **explicit** multithreading model in which the programmer must explicitly create and manage threads, OpenMP uses an **implicit** multithreading model in which the library handles thread creation and management, thus making the programmer's task much simpler and less error-prone. OpenMP is a standard that compilers who implement it must adhere to.

The following are examples of C code with OpenMP pragmas. There is one C++ example that is used to illustrate a point about that language. The first three are basic illustrations so you can get used to the OpenMP pragmas and conceptualize the two primary patterns used as **program structure implementation strategies** that almost all shared-memory parallel programs have:

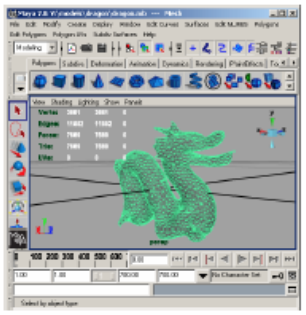
- **fork/join**: forking threads and joining them back, and
- **single program, multiple data**: writing one program in which separate threads maybe performing different computations simultaneously on different data, some of which might be shared in memory.

The rest of the examples illustrate how to implement other patterns along with the above two and what can go wrong when mutual exclusion is not properly ensured.

Note: by default OpenMP uses the **Thread Pool** pattern of concurrent execution control. OpenMP programs initialize a group of threads to be used by a given program (often called a pool of threads). These threads will execute concurrently during portions of the code specified by the programmer. In addition, the **multiple instruction, multiple data** pattern is used in OpenMP programs because multiple threads can be executing different instructions on different data in memory at the same point in time.

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)

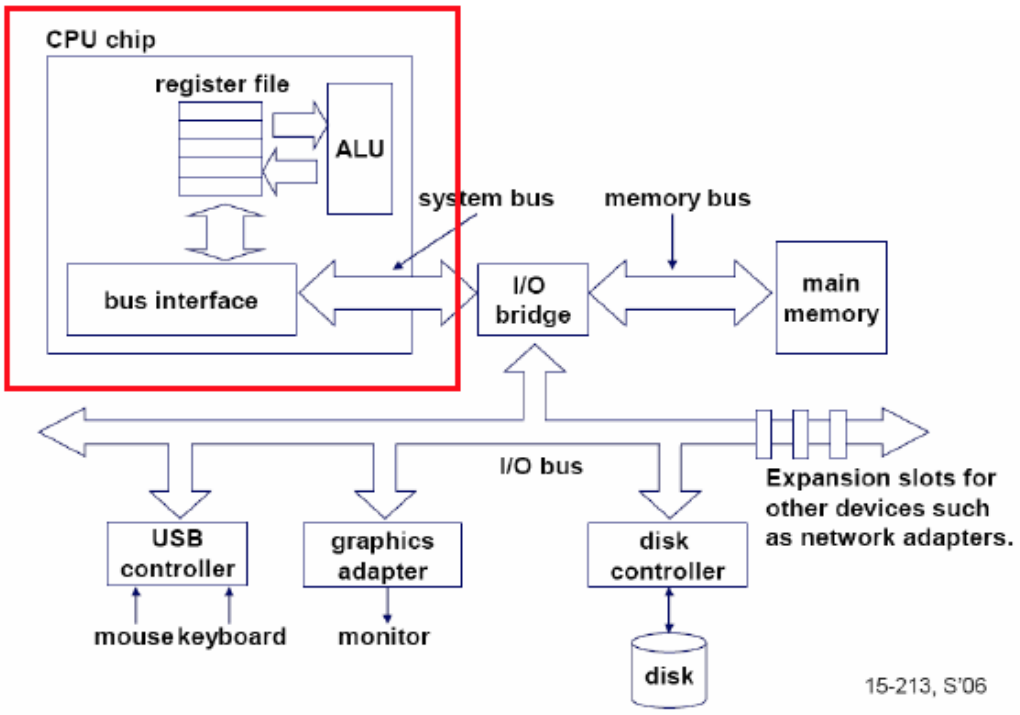


Each can run on its own core

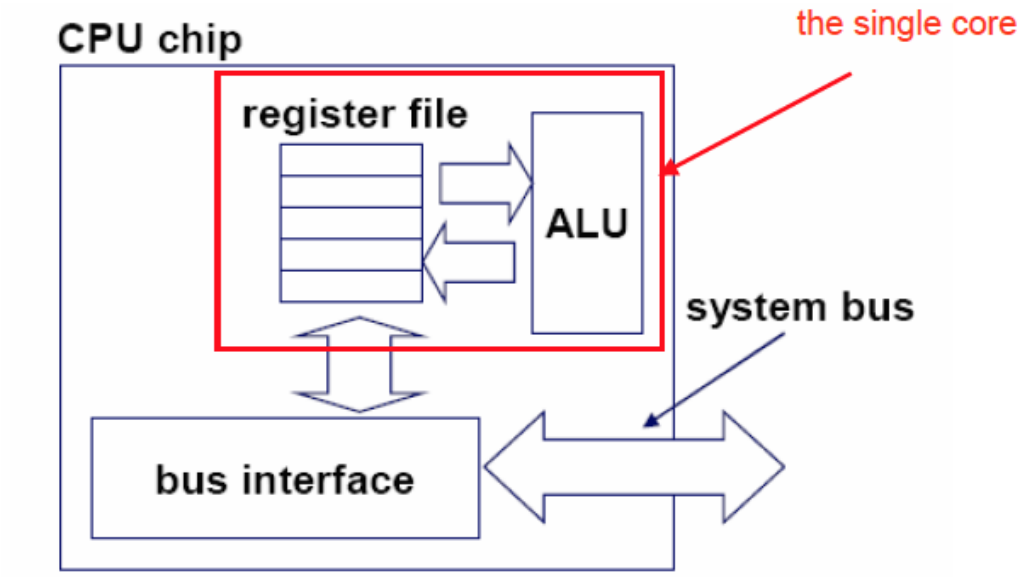
More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- “Anything that can be threaded today will map efficiently to multi-core”
- BUT: some applications difficult to parallelize

Single-core computer

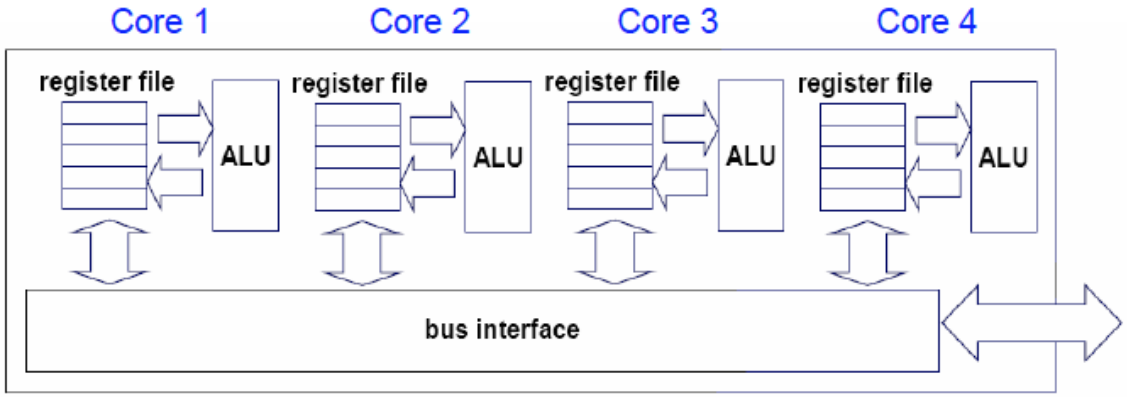


Single-core CPU chip



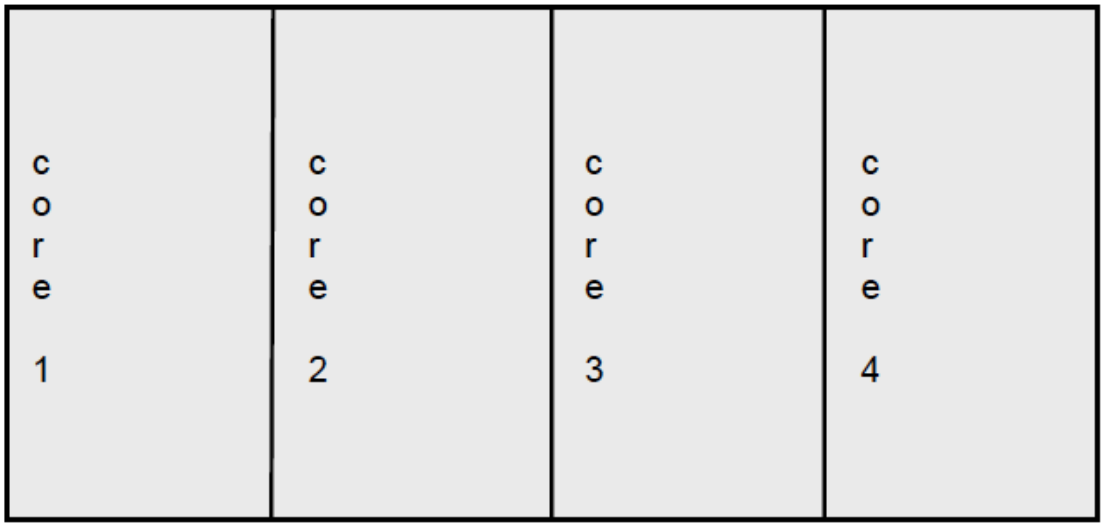
Multi-core architectures

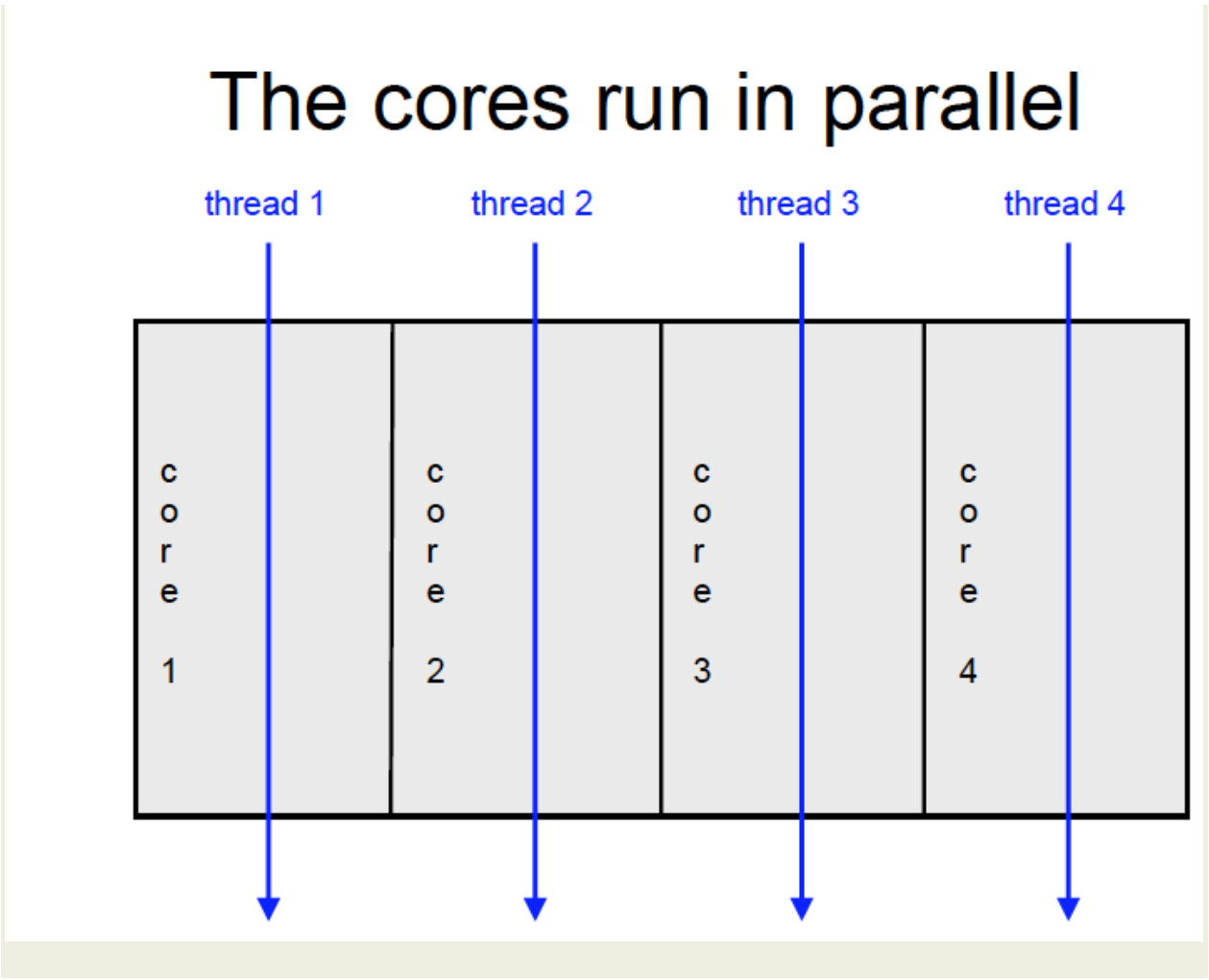
- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



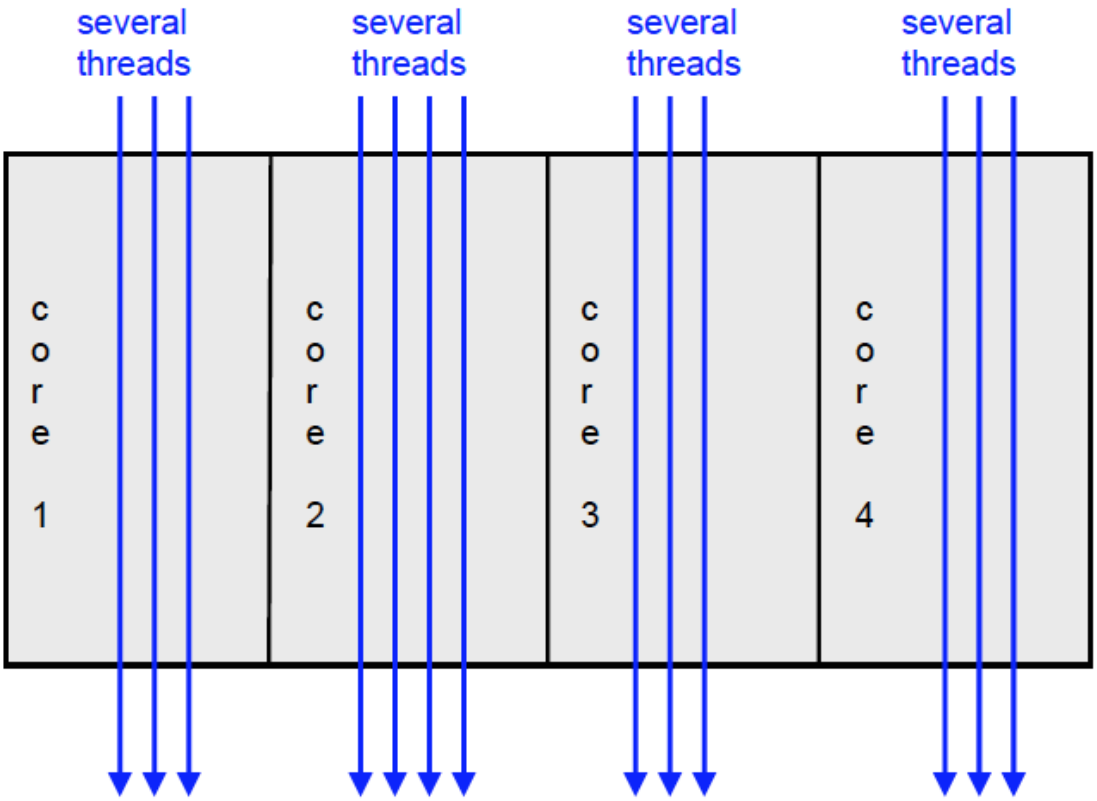
Multi-core CPU chip

- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)




The cores run in parallel
thread 1
thread 2
thread 3
thread 4
core 1
core 2
core 3
core 4

Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today: Windows, Linux, Mac OS X, …

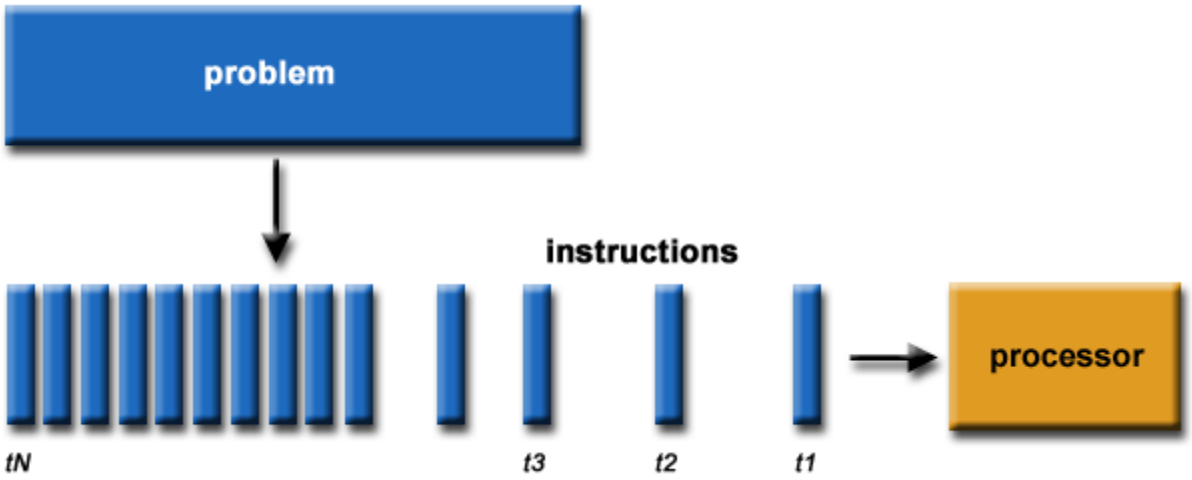
Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

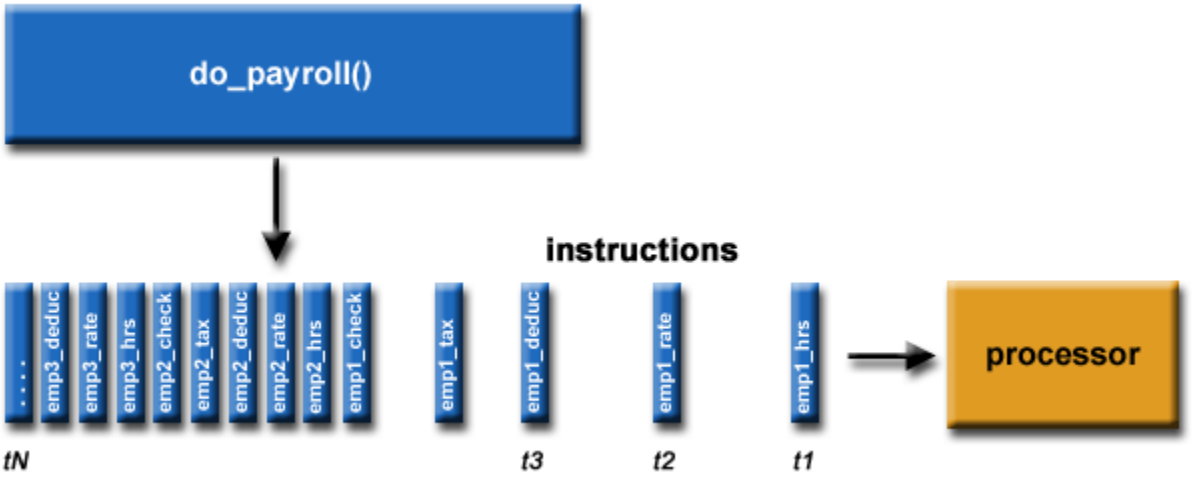
What is Parallel Computing?

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

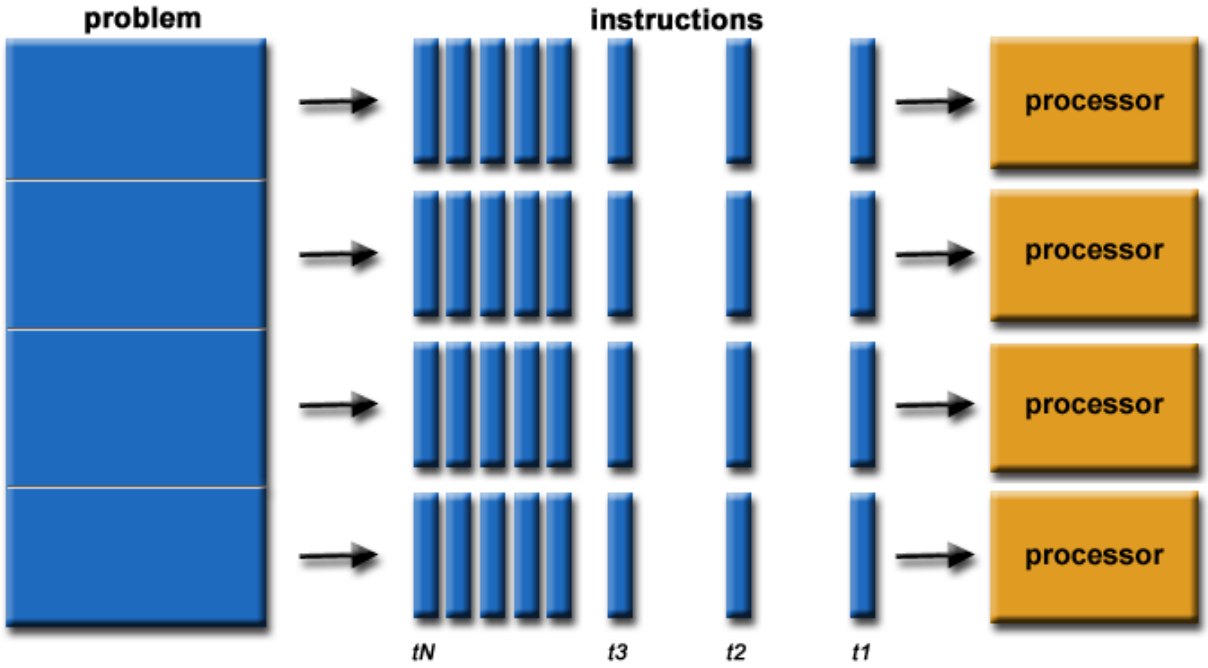


For example:

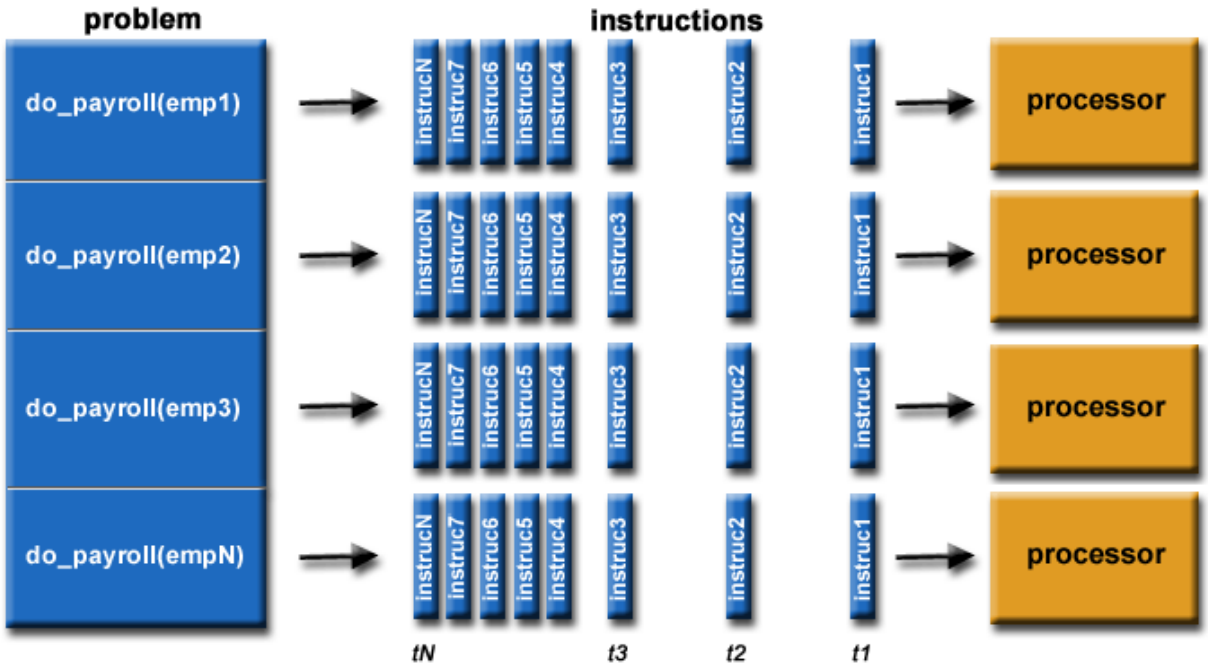


Parallel Computing:

- 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



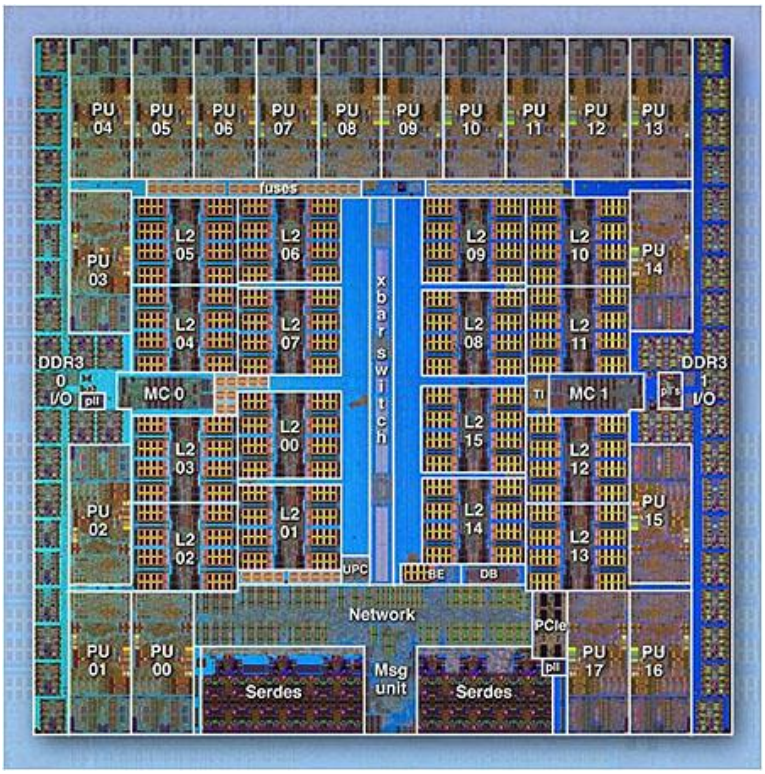
For example:



- 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

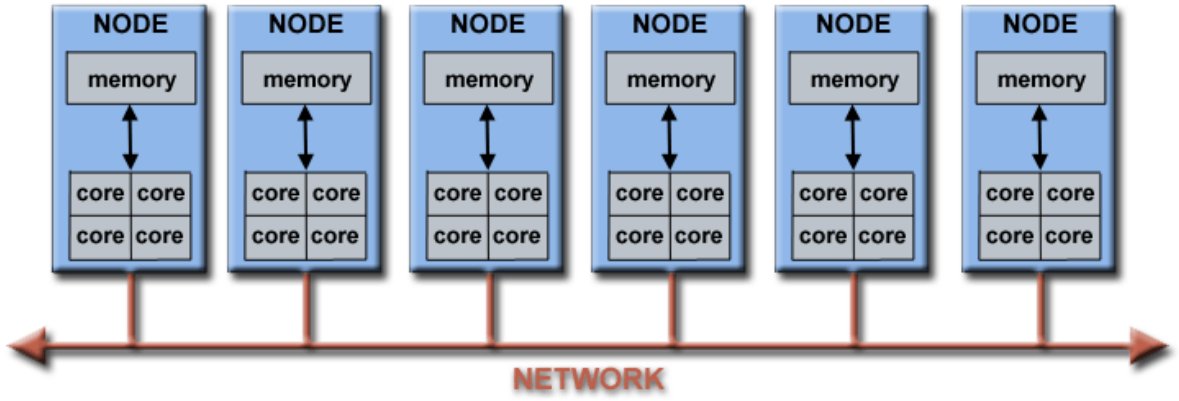
Parallel Computers:

- 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

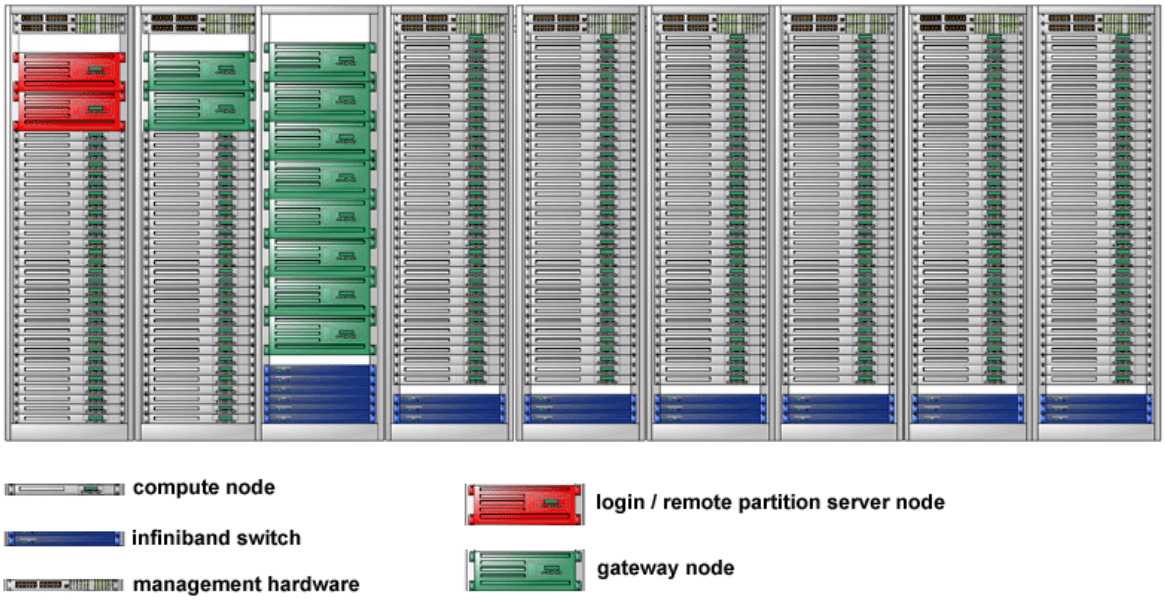


IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

- 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.

Forms of Parallelism

Though we have distinguished between thread-based parallelism and process-based parallelism, we have done so to focus on implementation differences, such as granularity and communication overhead. Now we are concerned with understanding where the parallelism can be found at the algorithmic level. We recognize three general types:

- data parallelism
- task parallelism
- pipelining

We now consider each, realizing that there is overlap among the categories.

Data Parallelism

Data parallelism refers to a broad category of parallelism in which the same computation is applied to multiple data items, so the amount of available parallelism is proportional to the input size, leading to tremendous amounts of potential parallelism. For example, the first chapter's "counting the 3s" computation is a data parallel computation: Each element must be tested equal to 3, which is a fully parallel operation. Once the individual outcomes are known, the number of "trues" can be accumulated using the tree summation technique. Notice that the tree add applies to all result elements only for its initial step

and has logarithmically diminishing parallelism thereafter. Still, the parallelism is generally proportional to the input size, so global sum is considered to be a data parallel operation.

As we observed in our discussion of locality and granularity above, the availability of full concurrency does not imply that the best algorithms will use it all. The Schwartz algorithm showed that foregoing concurrency to increase locality and reduce dependences with other threads produces a better result. Indeed, one of the best features of data parallelism is that it gives programmers flexibility in writing scalable parallel programs: The potential parallelism scales with the size of the input, and since, usually, $n >> P$, programs must be designed to process more data per processor than one item. That is, the program should be able to accommodate whatever parallelism is available. (It has been claimed that writing programs as if $n == P$ leads to effective programs because processors can be virtualized, i.e. the physical processors can simulate any number of logical processors, leading to code—it's claimed—that adapts well to any number of processors. This is not our experience. Virtualizing processors leads to extremely fine grain specifications that miss both the benefits of locality and the "economies of scale" of processing a batch of data. We prefer solutions like Schwartz's that explicitly handle batches of data.)

Task Parallelism

The broad classification of task parallelism applies to solutions where parallelism is organized around the functions to be performed rather than around the data. The term "task" in this case is not to be contrasted necessarily to "thread" as we normally do, because the emphasis is on the functional decomposition, which could be implemented with either tasks or threads.

For example, a client-server system employs task parallelism by assigning some tasks the job of making requests and others the job of servicing requests. As another example, the sub-expressions of a functional program can be evaluated in any order, so functional programs naturally exhibit large amounts of task parallelism. Though it is common for task parallel computations to apply an operation to similar data, as data parallel computations do, the task parallel approach becomes desirable when the context in which the data is evaluate matters significantly.

The challenges to task parallelism are to balance the work and to insure that all the work contributes to the result. In many cases, task parallelism does not scale as well as data parallelism.

Pipelining

Pipelined parallelism is a special form of task parallelism where a problem is divided into sub-problems, which can each be operated on independently, and where there are multiple problem instances to be solved. At any point in time, multiple processes can be busy, each working on a sub-problem of a different problem instance. As is familiar with bucket brigades, assembly lines, and pipelined processors, the solution is to run the operations concurrently, but on different problem instances. As the pipeline fills and

drains, there is less than full parallelism, as the opportunities for concurrency increase (fill) and then diminish (drain). A more crucial issue is the balancing of work of each operation. For pipelining to be maximally effective, the operations (stages) must complete in the same amount of time. Pipeline performance is determined—even for pipelines that are not clocked—by the longest running stage. Balancing the stages equals out the work, allowing all stages to process at the maximum prevailing rate.

Though pipelining is frequently thought of as a parallelism approach for cases defined by only a fixed length sequence of operations, it arises more generally. The number of (potential) stages is often determined by the input size. In such cases data dependences entail receiving input value(s) from one or more neighbors, computing, and then yielding the result(s) to opposite neighbor(s). The schematic in Figure 3.4 illustrates the idea. Clearly, in addition to maximizing the use of the processors, such computations are challenging in terms of avoiding stalls caused by fine grain interactions.

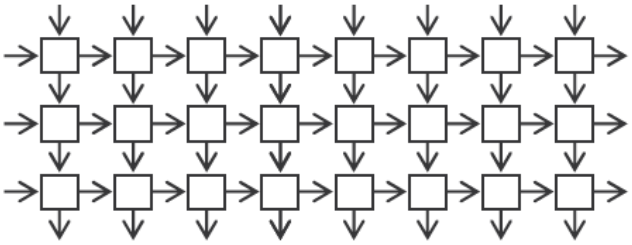
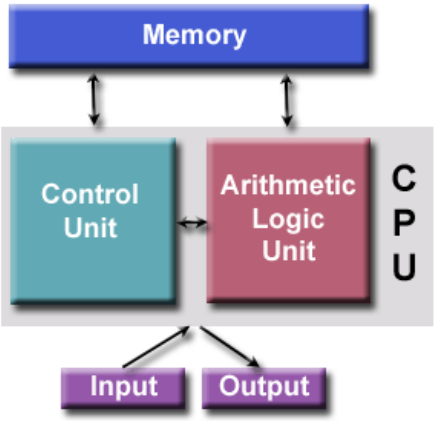


Figure 3.4. Schematic of a 2-dimensional pipelined computation, showing computation (boxes) and data flow (arrows). External data is presumed to be initially present; on the first step only the upper-left computation is enabled.

Concepts and Terminology

von Neumann Architecture

- Named after the Hungarian mathematician/genius John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Also known as "stored-program computer" - both program instructions and data are kept in electronic memory. Differs from earlier computers which were programmed through "hard wiring".
- Since then, virtually all computers have followed this basic design:



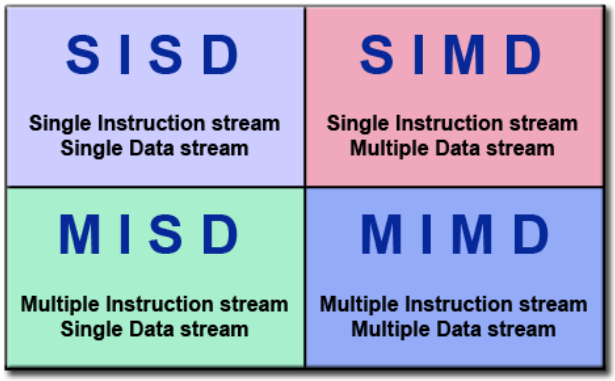
- Comprised of four main components:
  - Memory
  - Control Unit
  - Arithmetic Logic Unit
  - Input/Output
- Read/write, random access memory is used to store both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator

John von Neumann circa 1940s (Source: LANL archives)

Concepts and Terminology

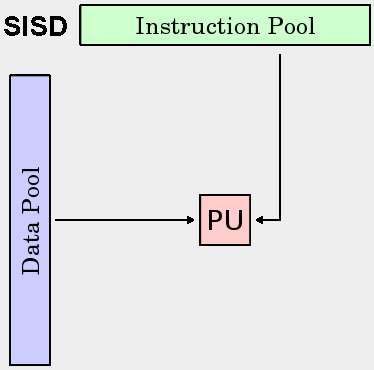
Flynn's Classical Taxonomy

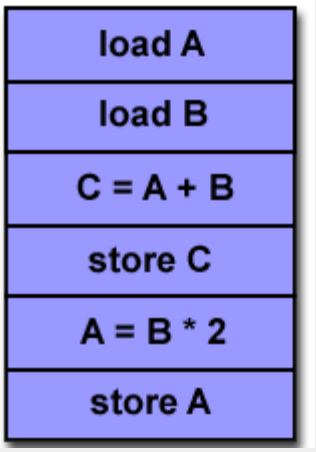
- There are different ways to classify parallel computers. Examples available HERE.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two 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

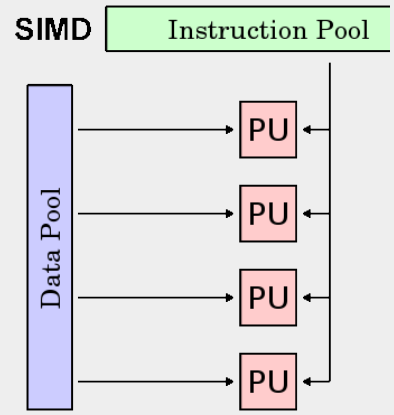
CDC 7600

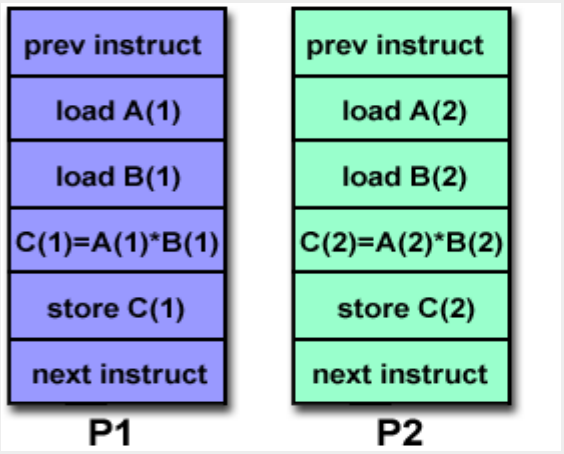
PDP1

Dell Laptop

Single Instruction, Multiple Data (SIMD):

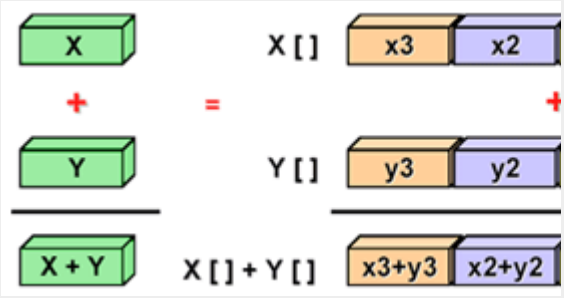
- A type of parallel computer
- **Single Instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple Data:** Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such 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.





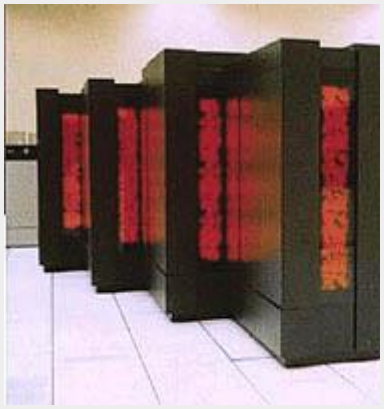
ILLIAC IV

MasPar

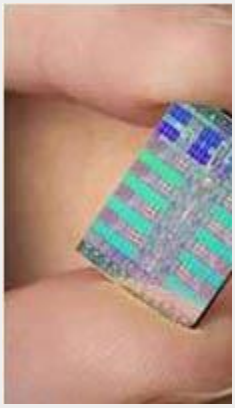


Cray X-MP

Cray Y-MP



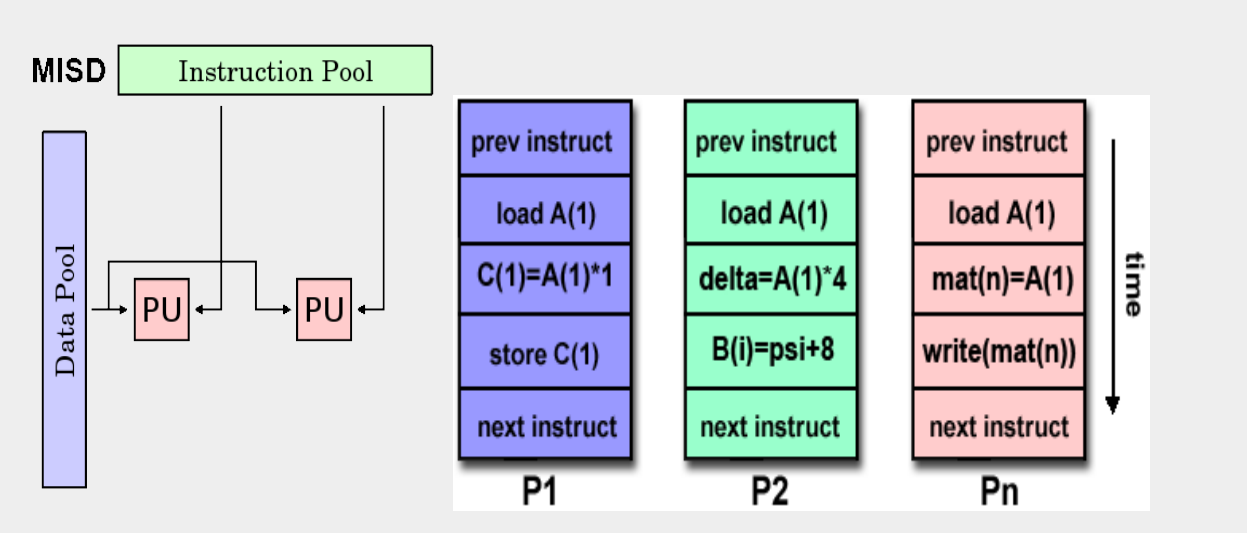
Thinking Machines CM-2



Cell Processor (GPU)

Multiple Instruction, Single Data (MISD):

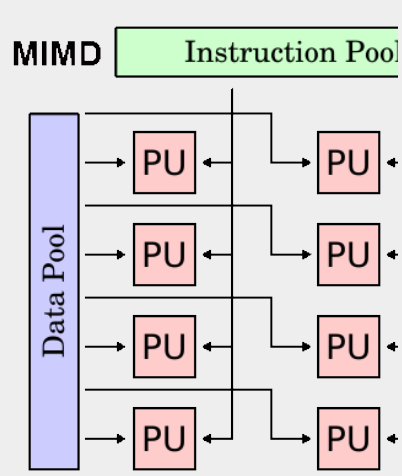
- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever 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.

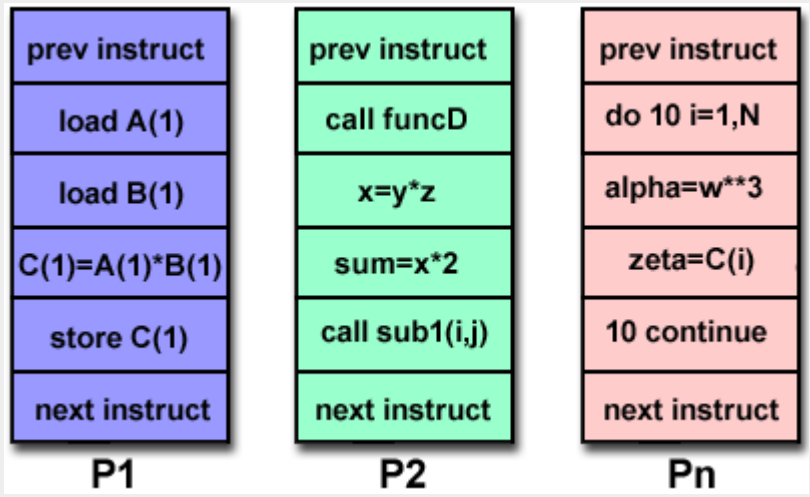


Multiple Instruction, Multiple Data (MIMD):

- 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