Parallel and High Performance Computing

CSE 745

Outline

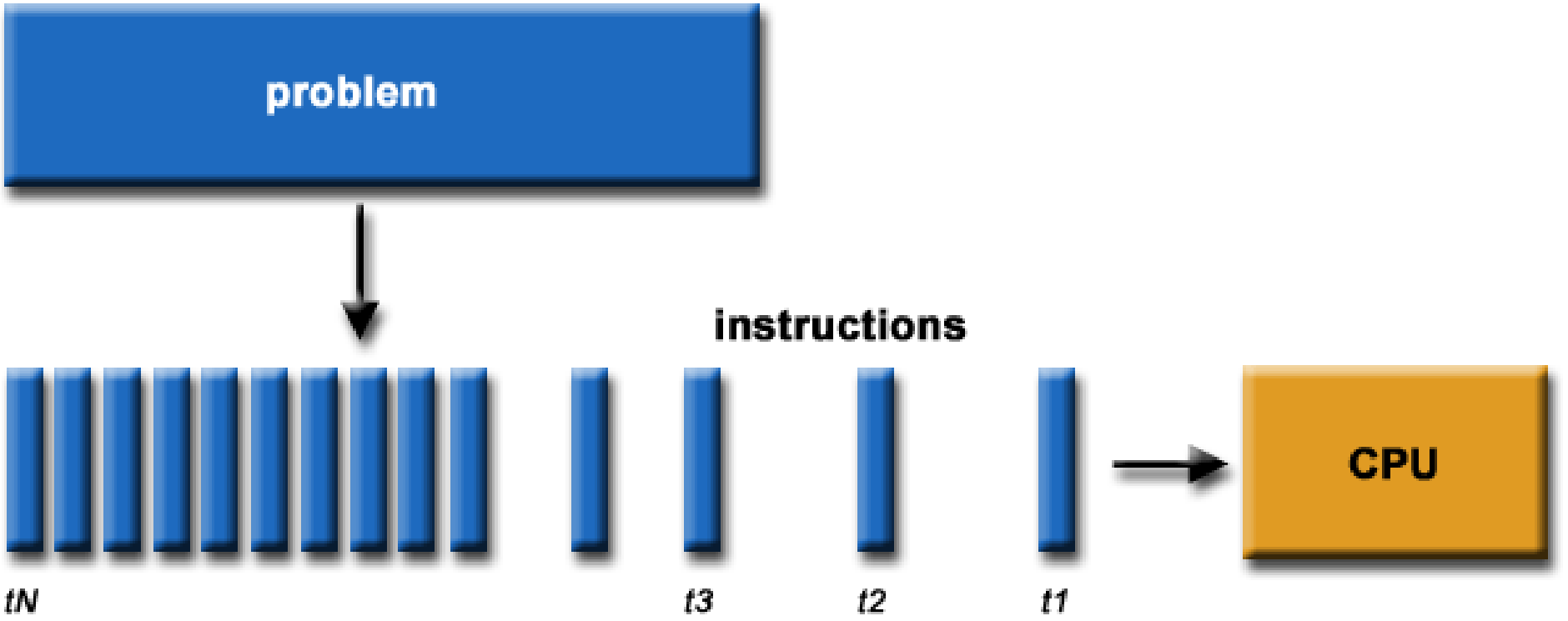
- Introduction to HPC computing
  - Overview
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Parallel examples
- OpenMP
- MPI

Introduction to HPC computing

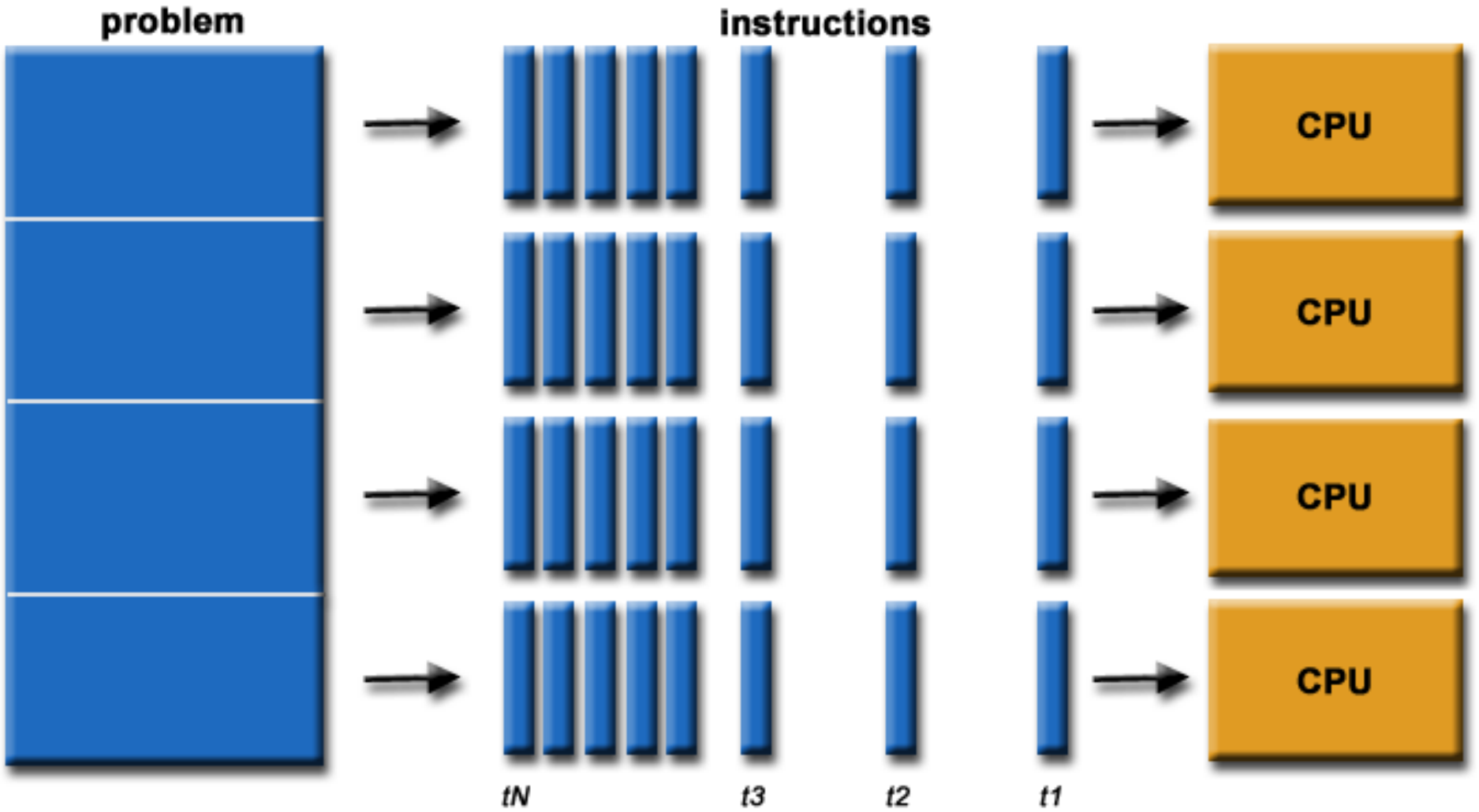
Overview

What is Parallel Computing?

- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.


problem
instructions
tN
t3
t2
t1
CPU

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs / cores
  - 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 CPUs / cores


problem
instructions
CPU
CPU
CPU
CPU
tN
t3
t2
t1

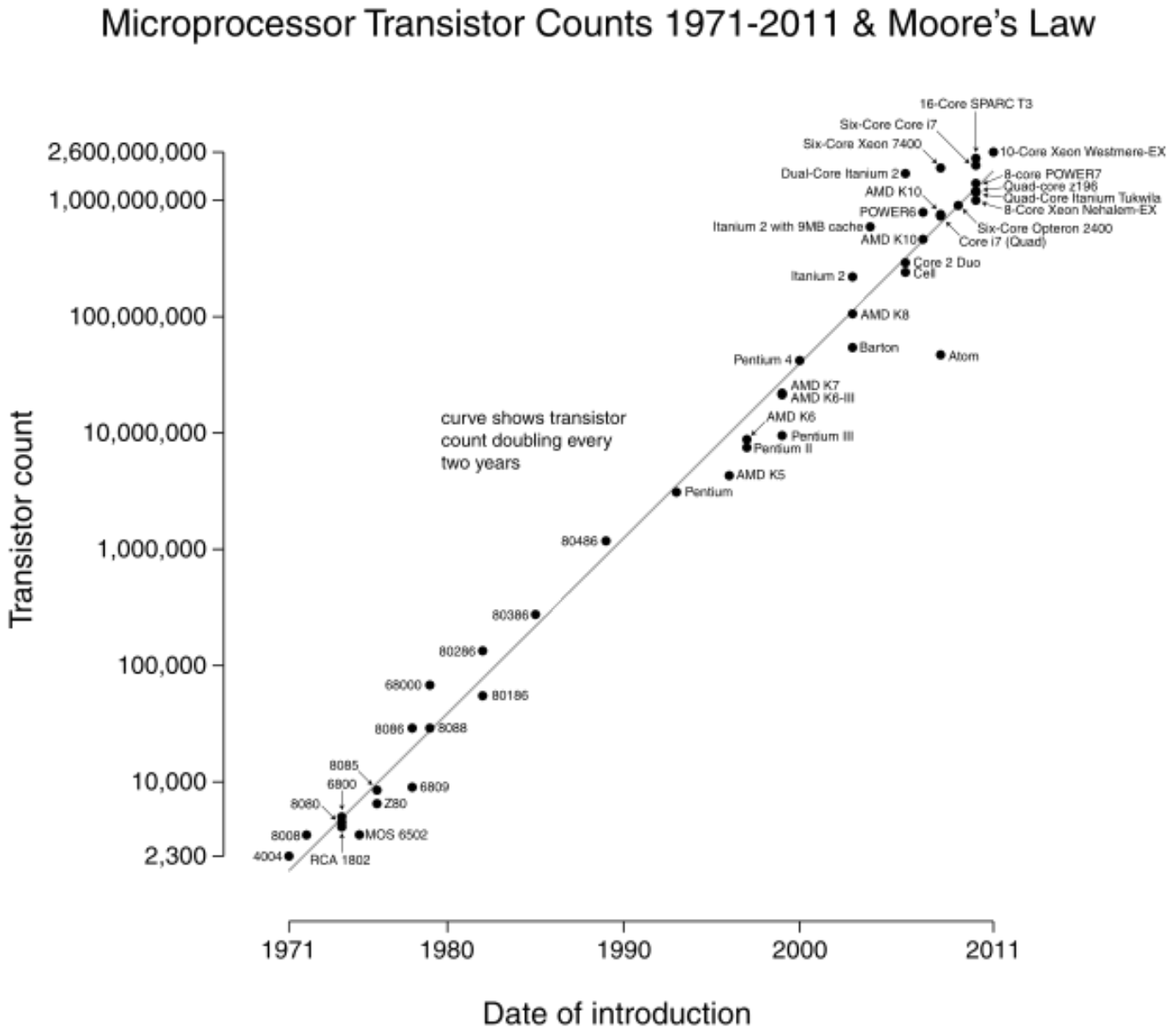
- The compute resources can include:
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem usually demonstrates characteristics such as the ability to:
  - Be broken apart into discrete pieces 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.

Why Use Parallel Computing?

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever faster serial computers:
  - Transmission speeds - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.
  - Limits to miniaturization - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.
  - Economic limitations - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

Moore’s law


Microprocessor Transistor Counts 1971-2011 & Moore’s Law
Transistor count
2,600,000,000
1,000,000,000
100,000,000
10,000,000
1,000,000
100,000
10,000
2,300
1971
1980
1990
2000
2011
Date of introduction
curve shows transistor count doubling every two years
16-Core SPARC T3
Six-Core Core i7
Six-Core Xeon 7400
10-Core Xeon Westmere-EX
Dual-Core Itanium 2
8-core POWER7
Quad-core z196
AMD K10
Quad-Core Itanium Tukwila
POWER6
8-Core Xeon Nehalem-EX
Itanium 2 with 9MB cache
Six-Core Opteron 2400
AMD K10
Core i7 (Quad)
Core 2 Duo
Cell
Itanium 2
AMD K8
Barton
Atom
Pentium 4
AMD K7
AMD K6-III
AMD K6
Pentium III
Pentium II
AMD K5
Pentium
80486
80386
80286
68000
80186
8086
8088
8085
6800
6809
8080
Z80
8008
MOS 6502
4004
RCA 1802

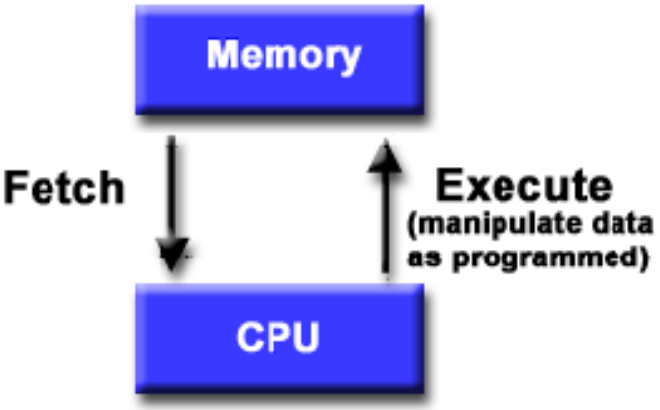
- The future: during the past 15 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) suggest that ***parallelism is the future of computing***.

von Neumann Architecture

- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

von Neumann Architecture (2)

- Basic design:
  - Memory is used to store both program and data instructions
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.



Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. 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*** and ***Data***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

Flynn's Classical Taxonomy (2)

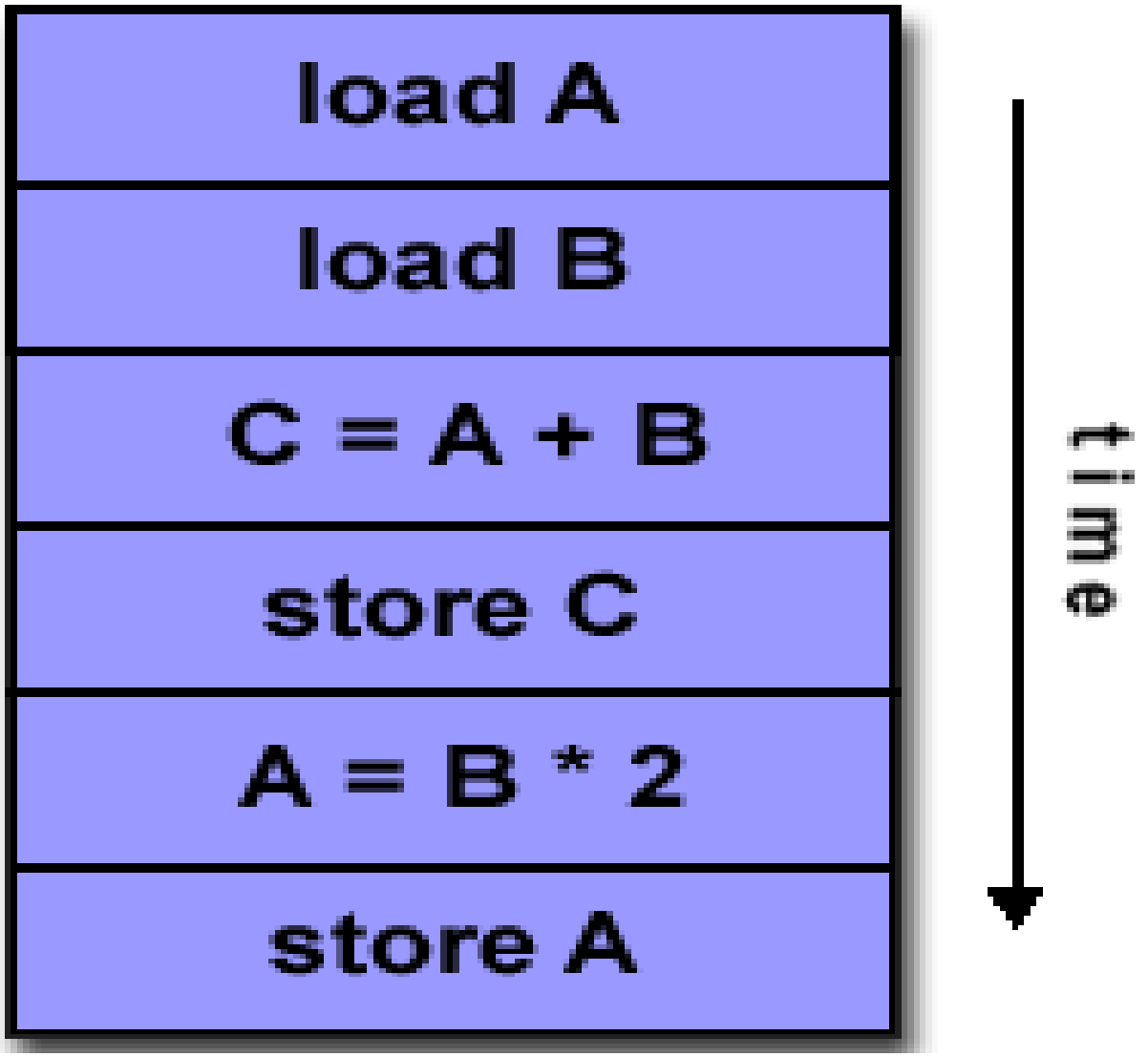
- The matrix below defines the 4 possible classifications according to Flynn.

S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

Single Instruction, Single Data

- 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 and until recently, the most prevalent form of computer

Single Instruction, Single Data (2)



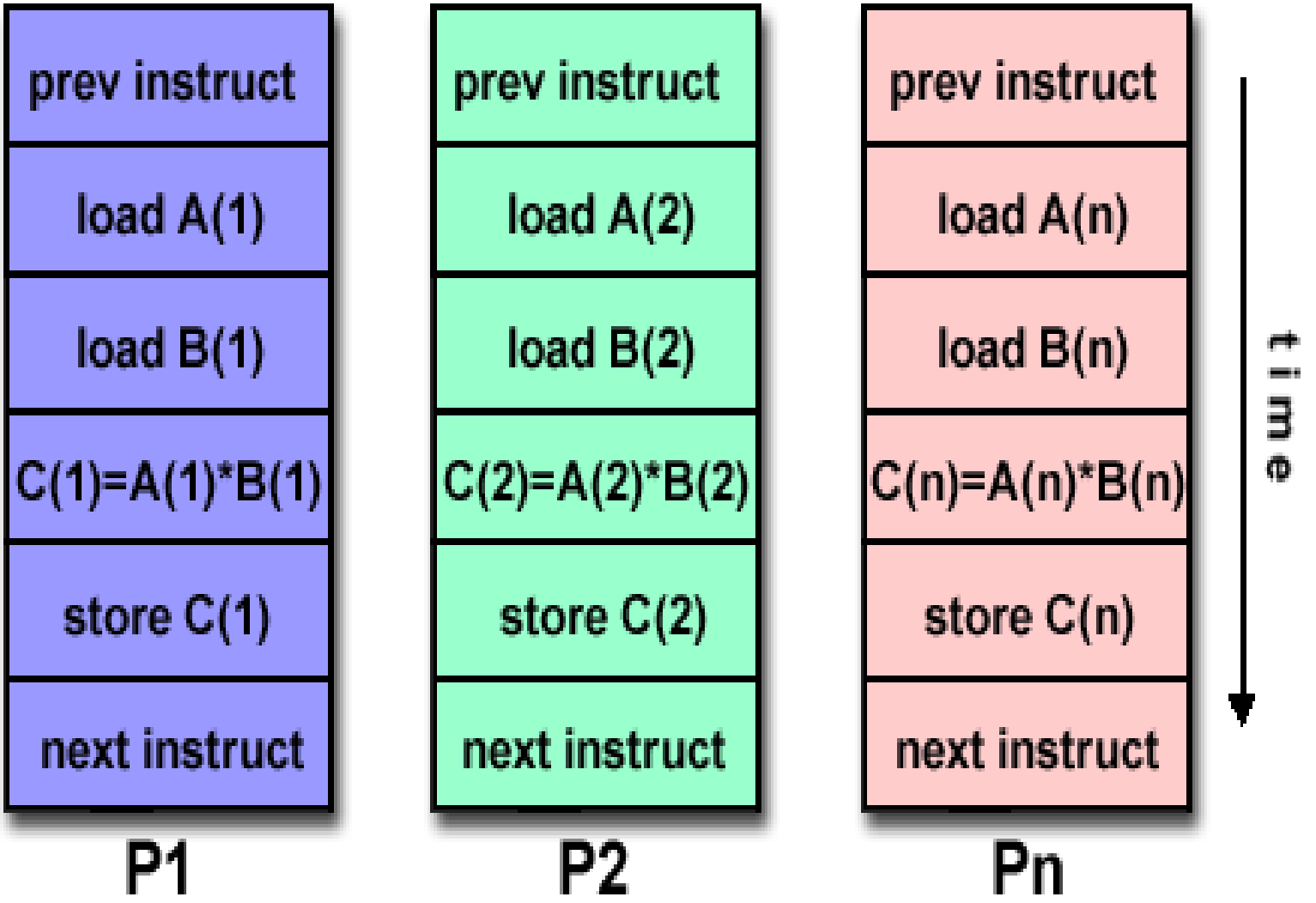
Single Instruction, Multiple Data

- 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
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.

Single Instruction, Multiple Data (2)

- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Examples:
  - Vector Processors: Thinking Machine CM-2, Cray Y-MP etc.
  - MMX and SSE extensions to the x86 architecture.
  - GPU's multiprocessors (groups of usually 32 cores)

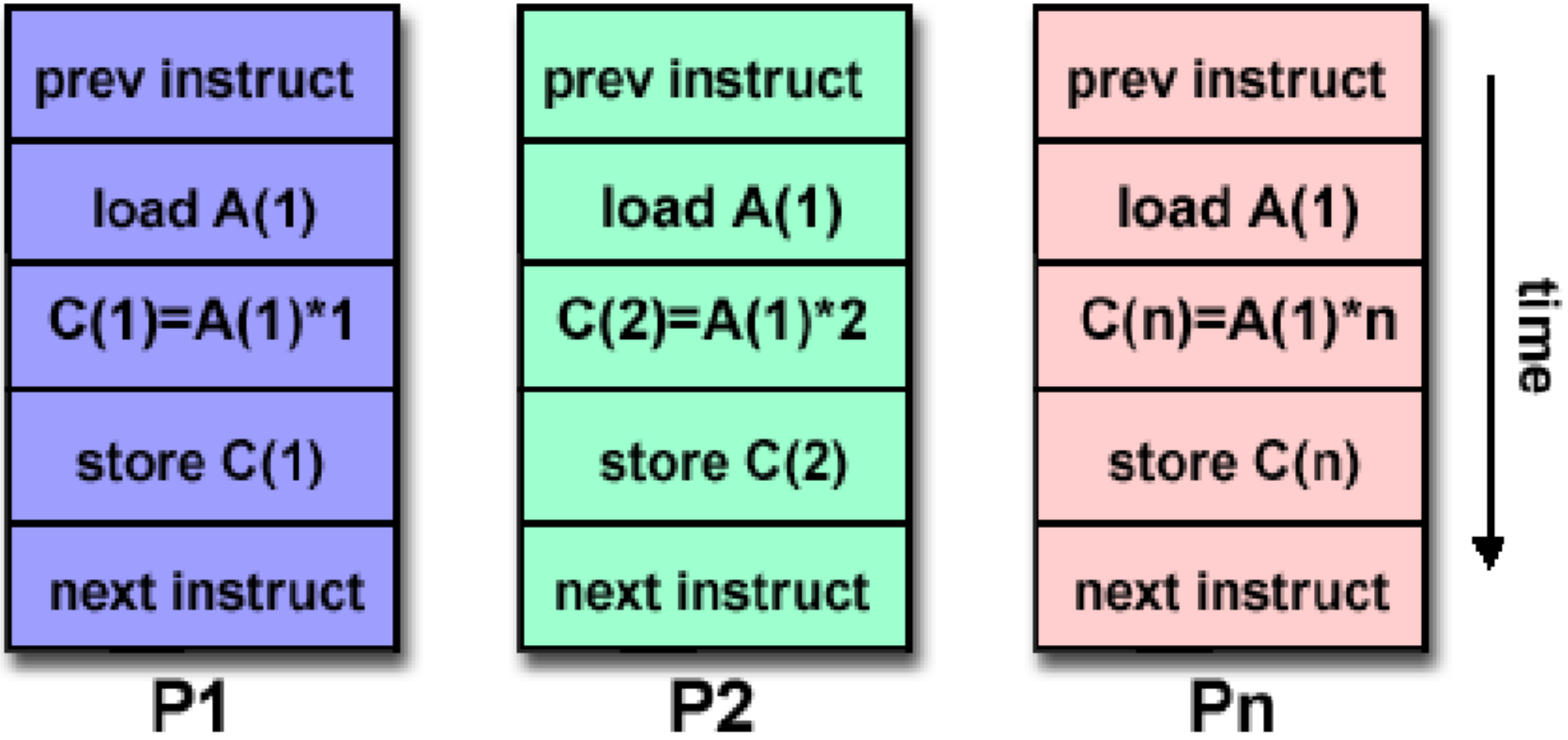
Single Instruction, Multiple Data (3)



Multiple Instruction, Single Data

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few 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.

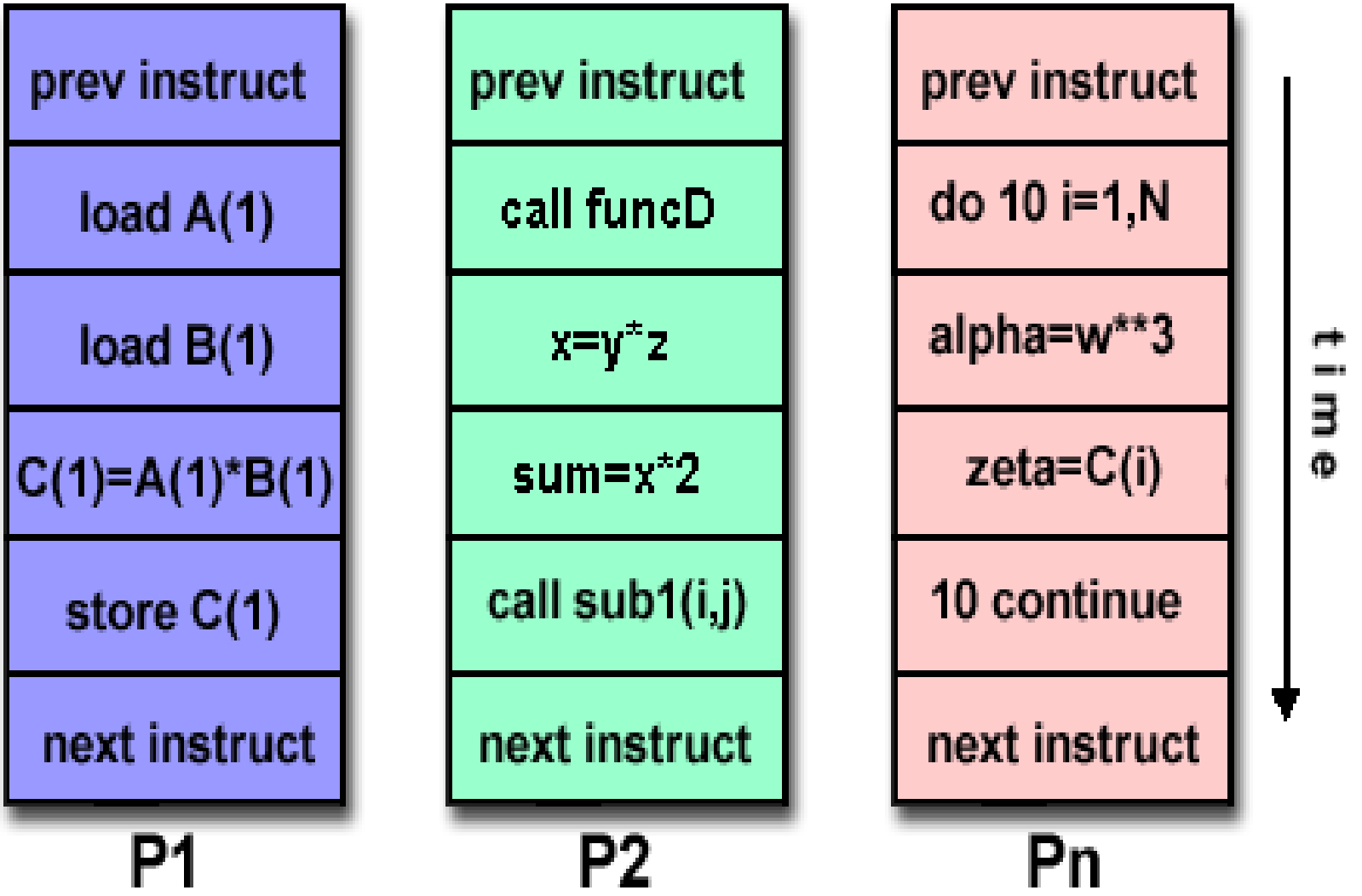
Multiple Instruction, Single Data (2)



Multiple Instruction, Multiple Data

- Currently, the most common type of parallel computer. Most modern computers fall into this category.
- 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
- Examples: most current supercomputers, networked parallel computer clusters and multi-processor SMP computers; most of the current PCs.

Multiple Instruction, Multiple Data (2)

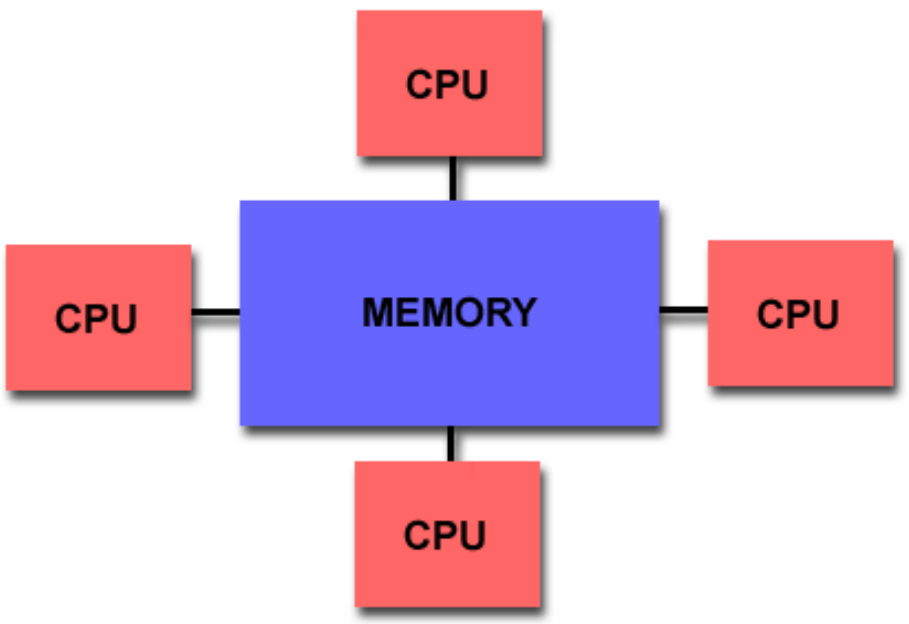


Parallel Computer Memory Architectures

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.



Shared Memory (2)

- 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.
- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.

Shared Memory (3)

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.

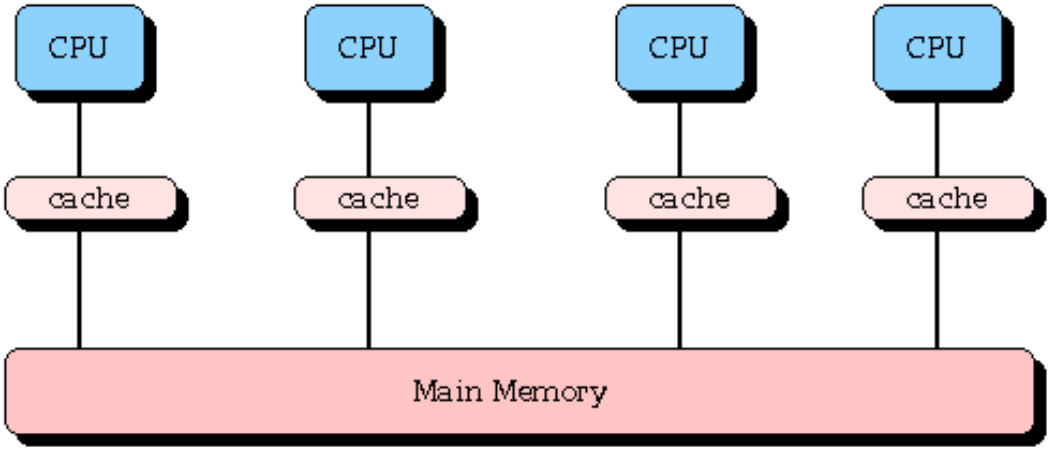
(Nearly all CPU architectures use a small amount of very fast non-shared memory known as *cache* to exploit locality of reference in memory accesses).

Shared Memory (4)

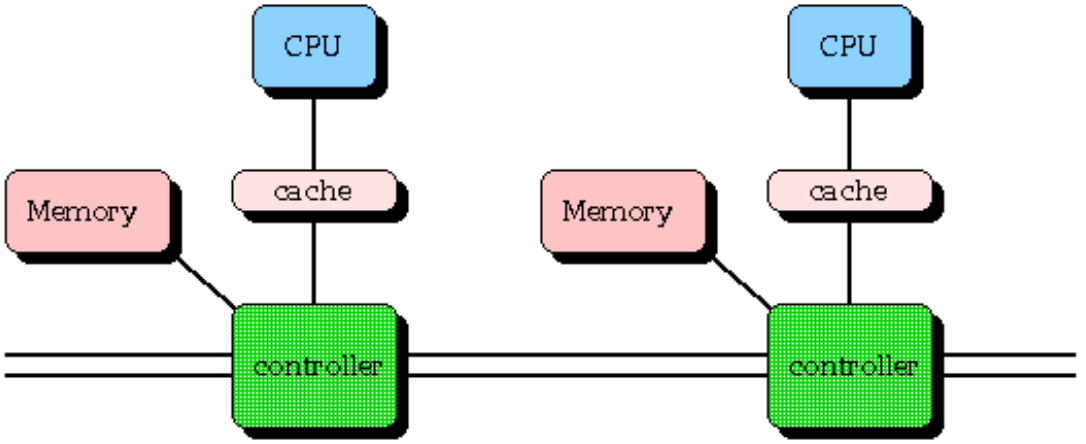
Non-Uniform Memory Access (NUMA):

- 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

Shared Memory (5)



UMA



NUMA

Shared Memory (6)

- With NUMA, the cache coherence is typically accomplished by using inter-processor communication between cache controllers to keep a consistent memory image when more than one cache stores the same memory location.
- For this reason, CC-NUMA performs poorly when multiple processors attempt to access the same memory area in rapid succession.
- Operating-system support for NUMA attempts to reduce the frequency of this kind of access by allocating processors and memory in NUMA-friendly ways and by avoiding scheduling and locking algorithms that make NUMA-unfriendly accesses necessary.

Shared Memory (7)

Advantages:

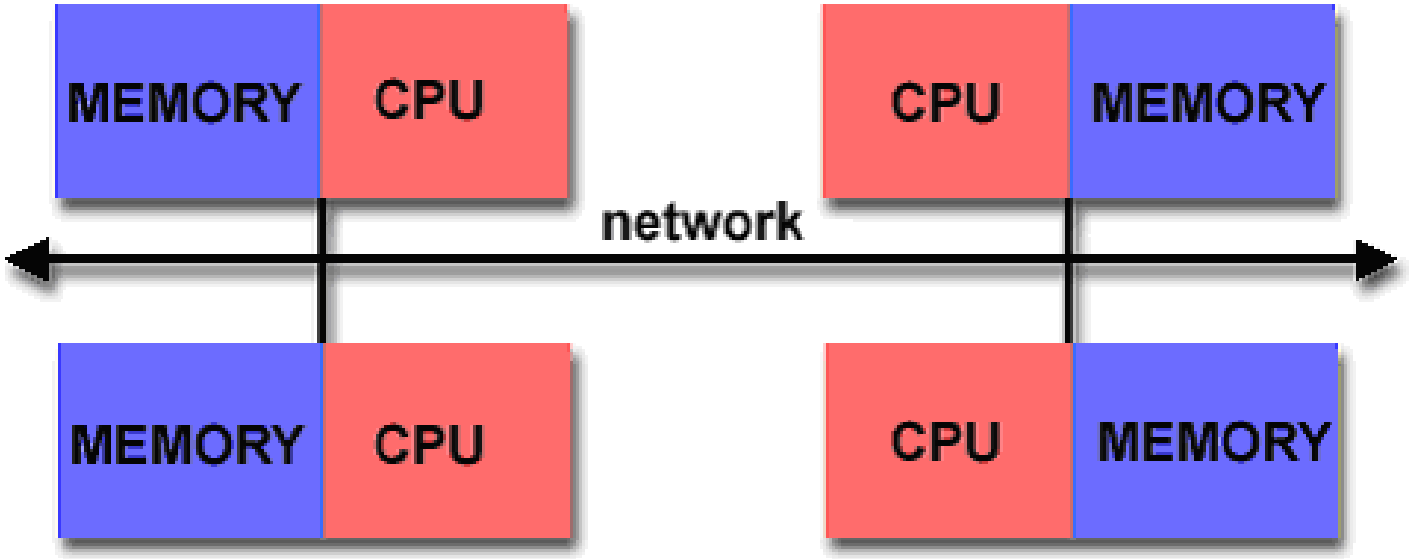
- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

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 insure "correct" access of global memory.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Distributed Memory

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.



Distributed Memory (2)

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- When a processor needs to access data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can can be as simple as Ethernet.

Distributed Memory (3)

Advantages:

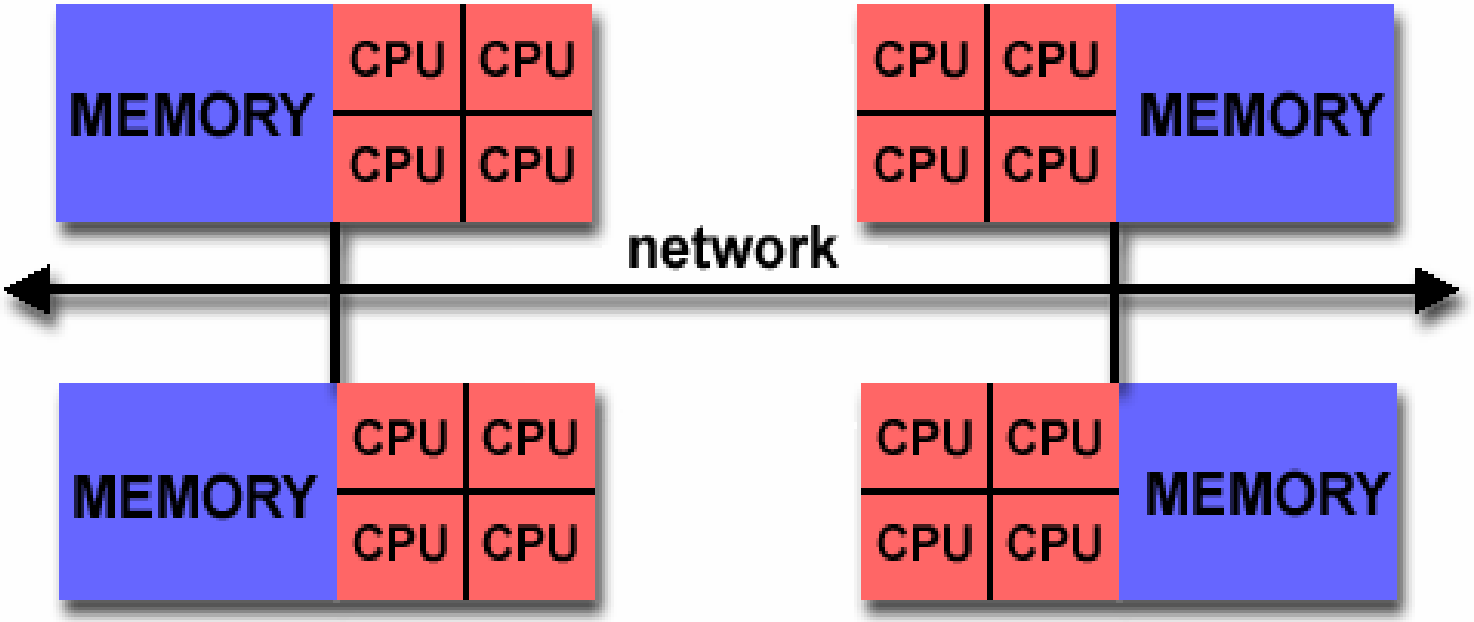
- Memory is scalable with number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

Hybrid Distributed-Shared Memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



Hybrid Distributed-Shared Memory (2)

- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

Parallel Programming Models

Overview

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.

Overview (2)

- Which model to use 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 slides will cover each of the models mentioned above, and will also discuss some of their actual implementations.

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- 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. 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.

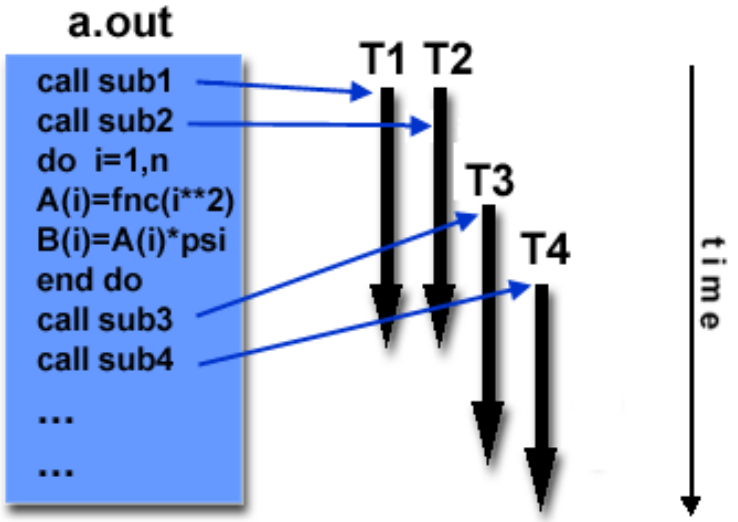
Shared Memory Model (2)

Implementations:

- On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.
- No common distributed memory platform implementations currently exist.

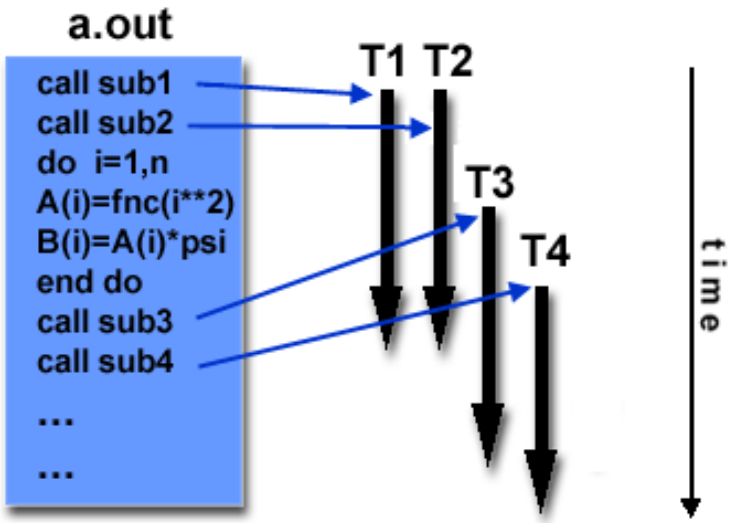
Threads Model

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
  - 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.



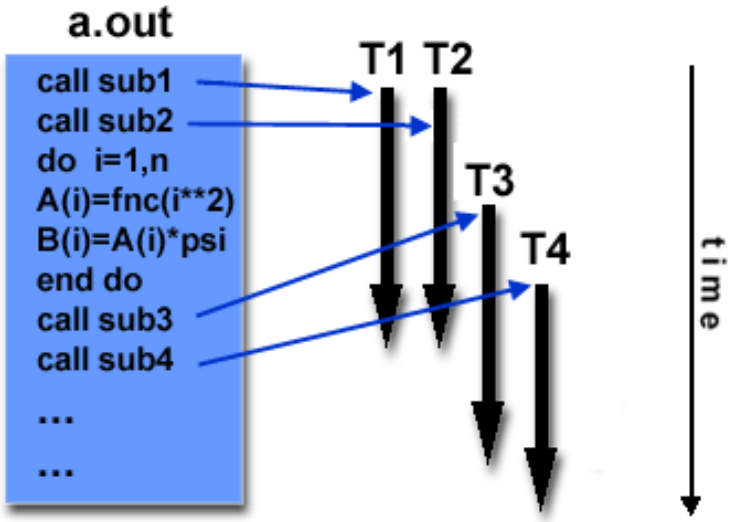
Threads Model (2)

- 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. 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 Model (3)

  - Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to insure 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.
- Threads are commonly associated with shared memory architectures and operating systems.



Threads: 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 embedded in either serial or parallel source code
- In both cases, the programmer is responsible for determining all parallelism.
- 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.

Threads: implementations (2)

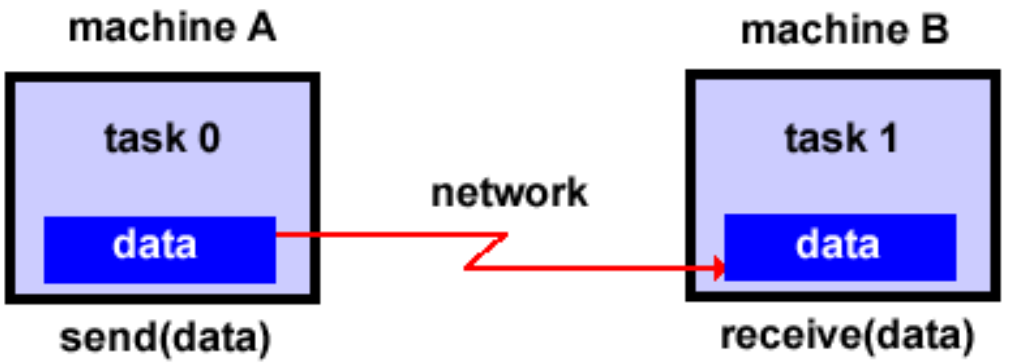
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***.
- **POSIX Threads**
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995).
  - C Language only
  - Commonly referred to as Pthreads.
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
  - Very explicit parallelism; requires significant programmer attention to detail.

Threads: implementations (3)

- **OpenMP**
  - Compiler directive based; can use serial code
  - Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
  - Portable / multi-platform, including Unix and Windows NT platforms
  - Available in C/C++ and Fortran implementations
  - Can be very easy and simple to use - provides for "incremental parallelism"
- Microsoft has its own implementation for threads, which is not related to the UNIX POSIX standard or OpenMP.

Message Passing Model

- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



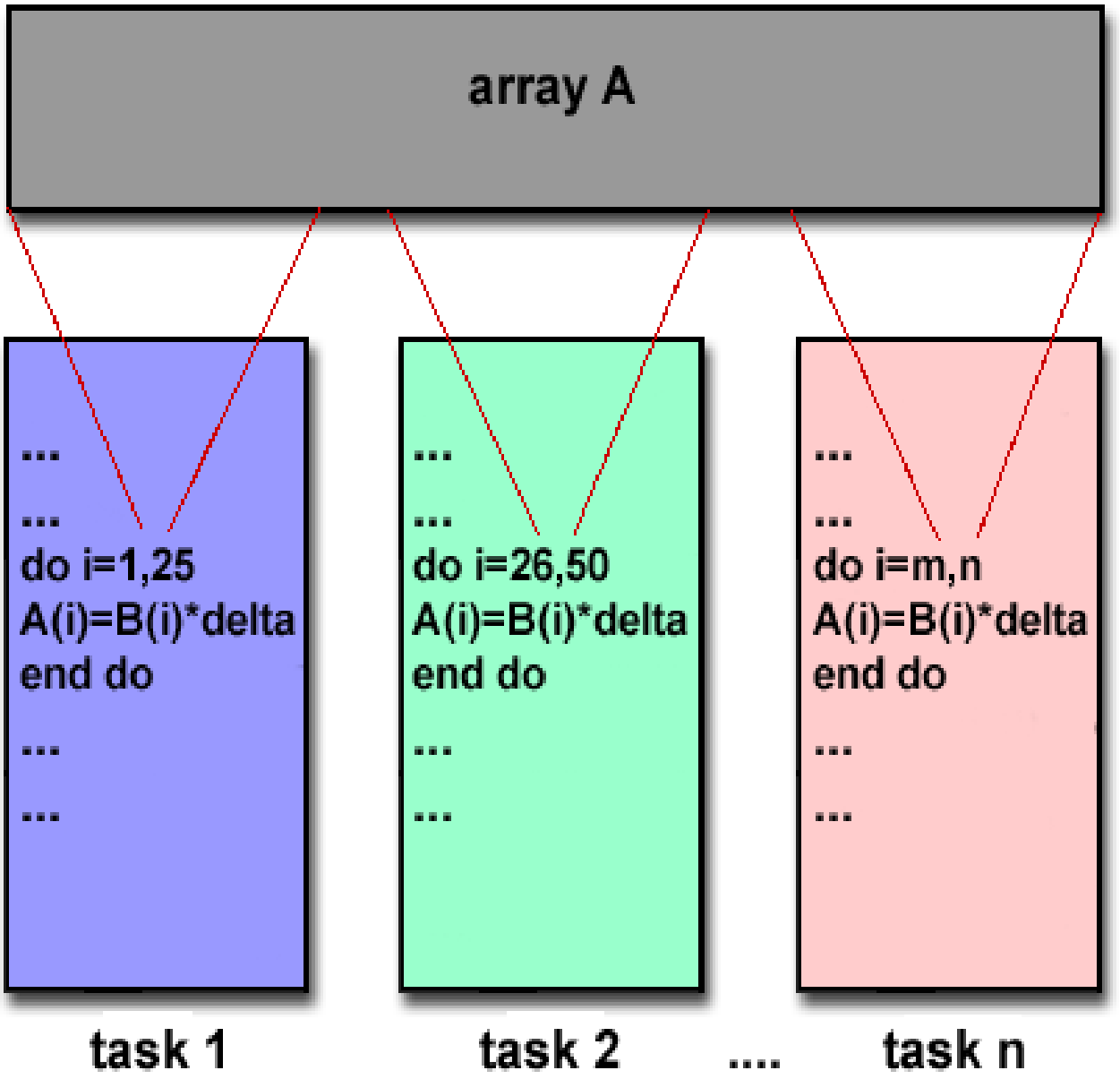
MPI: implementations

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are embedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

MPI: implementations (2)

- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. Part 3 (not covered in this course) was released in 2012. All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. SHARCNET is using Open MPI library (open source), which has a full MPI-2 standards conformance (MPI-3 is incomplete).
- For shared memory architectures, MPI implementations usually don't use a network for task communications. Instead, they use shared memory (memory copies) for performance reasons.

Data Parallel Model



Data Parallel Model (2)

- The data parallel model demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.

Implementations

- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or compiler directives recognized by a data parallel compiler.
- **High Performance Fortran (HPF):** Extensions to Fortran 90 to support data parallel programming.
  - Contains everything in Fortran 90
  - Directives to tell compiler how to distribute data added
  - Assertions that can improve optimization of generated code added
  - Data parallel constructs added (now part of Fortran 95)
  - Implementations are available for most common parallel platforms

Implementations (2)

Coarray Fortran:

- An extension of Fortran 95/2003 for parallel processing created by Robert Numrich and John Reid.
- A Coarray Fortran program is interpreted as if it were replicated a number of times and all copies were executed asynchronously. Each copy has its own set of data objects and is termed an *image*. The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to provide a concise representation of references to data that is spread across images.
- The Fortran 2008 standard now includes coarrays, as decided at the May 2005 meeting of the ISO Fortran Committee.

Implementations (3)

UPC (Unified Parallel C):

- an extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (e.g. clusters).
- The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor.

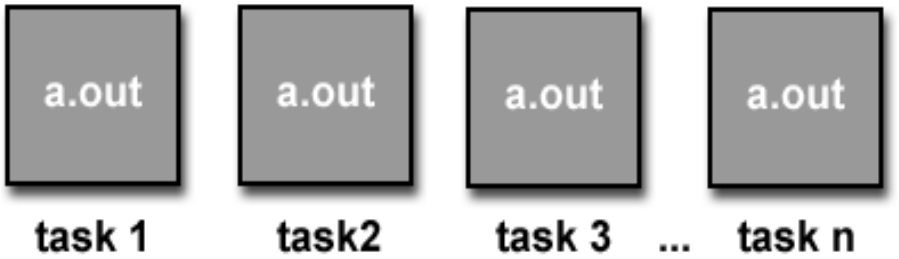
Implementations (4)

- In order to express parallelism, UPC extends ISO C 99 with the following constructs:
  - An explicitly parallel execution model
  - A shared address space
  - Synchronization primitives and a memory consistency model
  - Memory management primitives

Hybrid

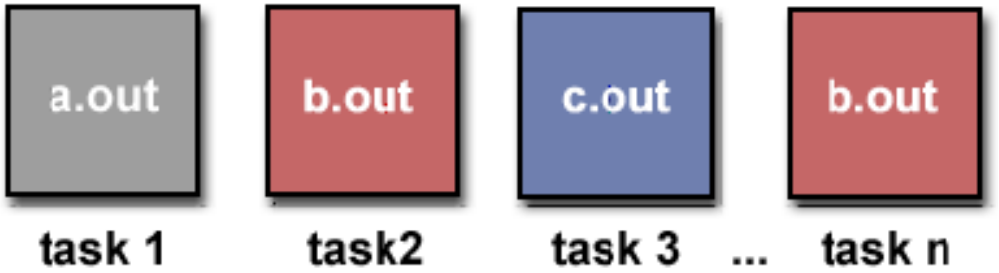
- In this model, any two or more parallel programming models are combined.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP). This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- CUDA programming model (for GPU programming) is another example. It combines elements of the data parallel model (at the lower, block level), shared memory model (at the higher, inter-block level), and distributed memory model (between CPU and GPUs).

Single Program Multiple Data (SPMD):



- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data

Multiple Program Multiple Data (MPMD)



- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data.