Introduction to Parallel Computing

Fundamentals and Terminology

Dr. Ugur GUVEN

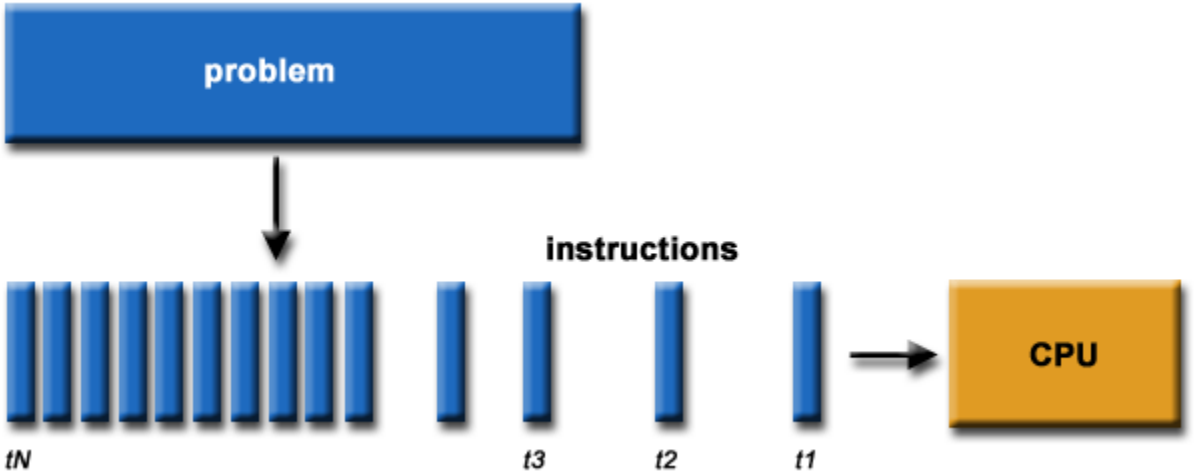
Abstract – What is the Scope of Parallel Computing

- This presentation covers the basics of parallel computing. Beginning with a brief overview and some concepts and terminology associated with parallel computing, the topics of parallel memory architectures and programming models are then explored. These topics are followed by a discussion on a number of issues related to designing parallel programs. The last portion of the presentation is spent examining how to parallelize several different types of serial programs.

- Level/Prerequisites: None

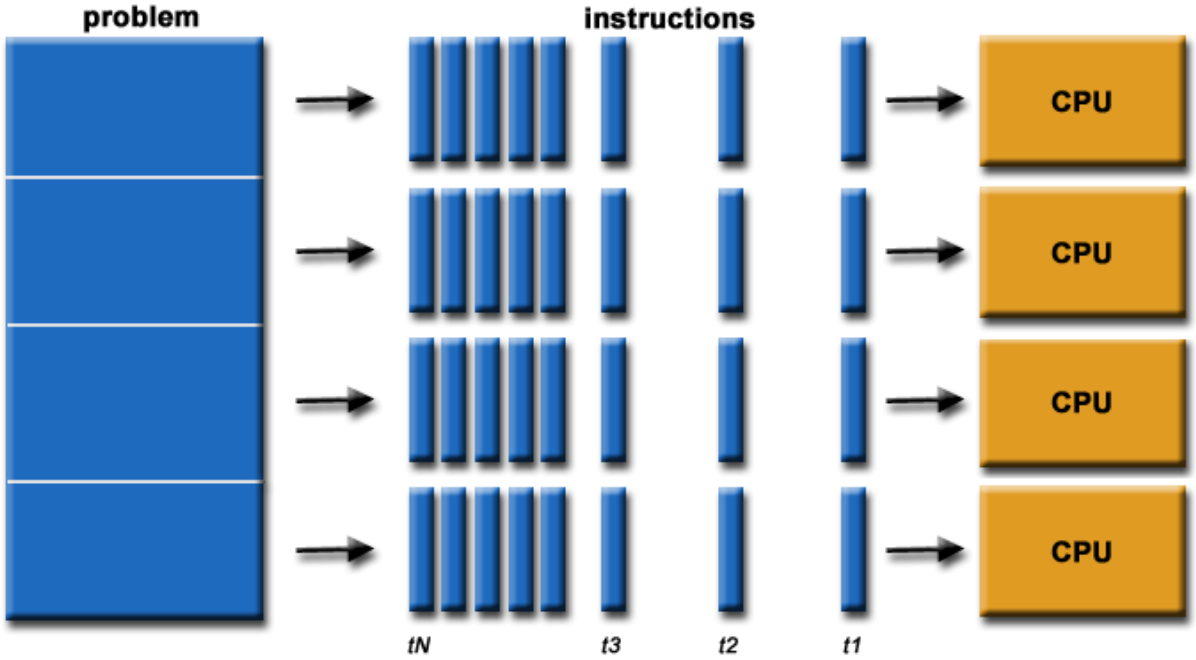
What is Parallel Computing? (1)

- 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 ins
  - Ins er.
  - Or ioment in



What is Parallel Computing? (2)

- 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
  - 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



Parallel Computing: Resources

- The compute resources can include:
  - A single computer with multiple processors;
  - A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA …)
  - An arbitrary number of computers connected by a network;
  - A combination of both.

Parallel Computing: The Computational Problem

- 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;
  - Solved in less time with multiple compute resources than with a single compute resource.

Parallel Computing: What For? (1)

- Parallel computing is an evolution of serial computing that attempts to emulate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence.
- Some examples:
  - Planetary and galactic orbits
  - Weather and ocean patterns
  - Tectonic plate drift
  - Rush hour traffic in Paris
  - Automobile assembly line
  - Daily operations within a business
  - Building a shopping mall
  - Ordering a hamburger at the drive through.

Parallel Computing: what for? (2)

- Traditionally, parallel computing has been considered to be "the high end of computing" and has been motivated by numerical simulations of complex systems and "Grand Challenge Problems" such as:
  - weather and climate
  - chemical and nuclear reactions
  - biological, human genome
  - geological, seismic activity
  - mechanical devices - from prosthetics to spacecraft
  - electronic circuits
  - manufacturing processes
  - CFD Problems

Parallel Computing: what for? (3)

- Today, commercial applications are providing an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Example applications include:
  - parallel databases, data mining
  - oil exploration
  - web search engines, web based business services
  - computer-aided diagnosis in medicine
  - management of national and multi-national corporations
  - advanced graphics and virtual reality, particularly in the entertainment industry
  - networked video and multi-media technologies
  - collaborative work environments
- Ultimately, parallel computing is an attempt to maximize the infinite but seemingly scarce commodity called time.

Why Parallel Computing? (1)

- This is a legitime question! Parallel computing is complex on any aspect!

- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

Why Parallel Computing? (2)

- 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.

Limitations of Serial Computing

- 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.

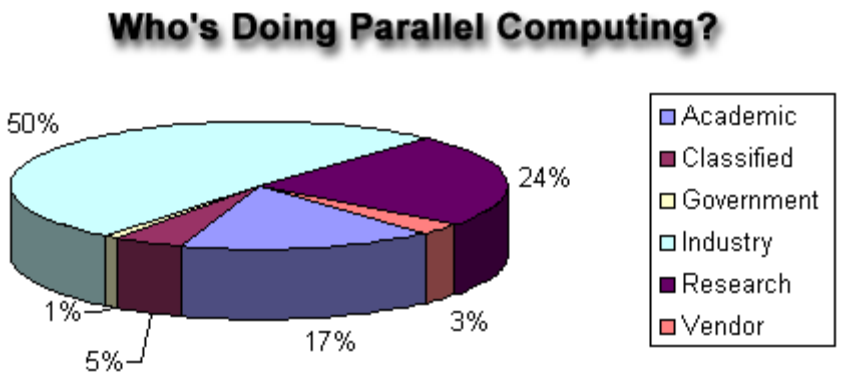
The Future

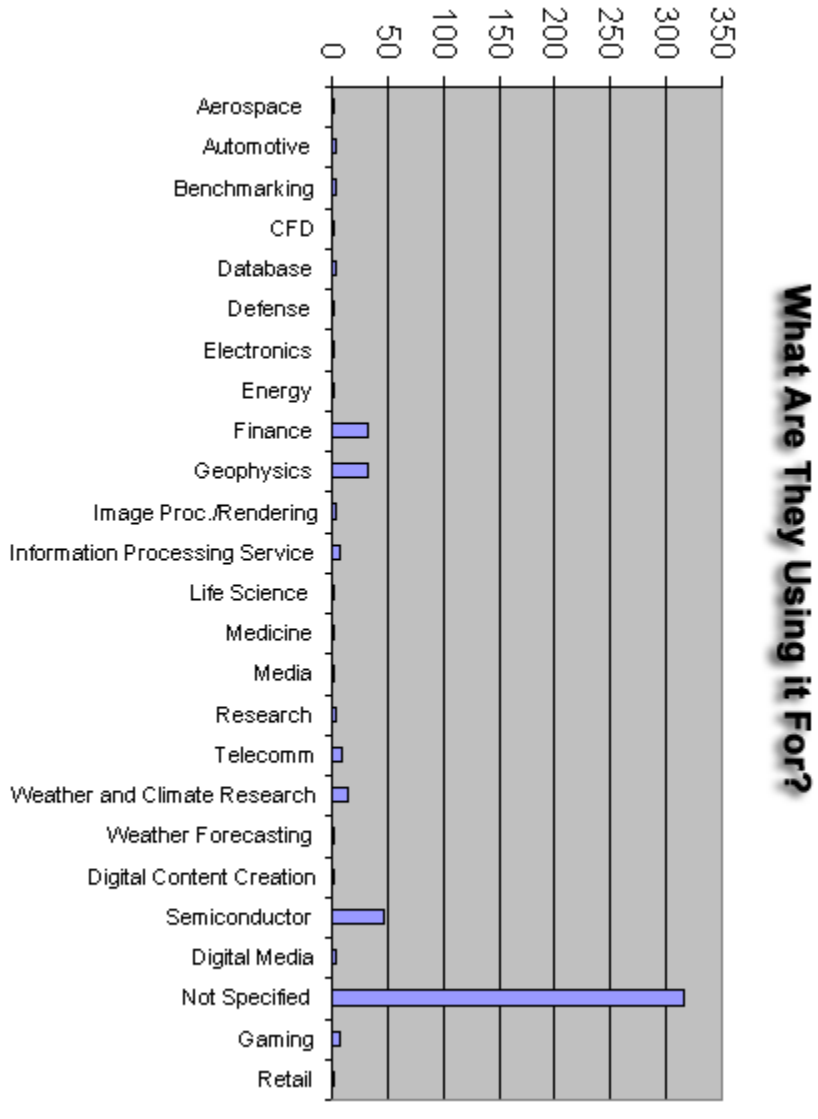
- During the past 10 years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing***.
- It will be multi-forms, mixing general purpose solutions (your PC…) and very speciliazed solutions as IBM Cells, ClearSpeed, GPGPU from NVidia …

Who and What? (1)

- Top500.org provides statistics on parallel computing users - the charts below are just a sample. Some things to note:
  - Sectors may overlap - for example, research may be classified research. Respondents have to choose between the two.
- "Not Specified" is by far the largest application - probably means multiple applications.

Who and


Who's Doing Parallel Computing?
50%
24%
1%
5%
17%
3%
Academic
Classified
Government
Industry
Research
Vendor


What Are They Using it For?
0
50
100
150
200
250
300
350
Aerospace
Automotive
Benchmarking
CFD
Database
Defense
Electronics
Energy
Finance
Geophysics
Image Proc./Rendering
Information Processing Service
Life Science
Medicine
Media
Research
Telecomm
Weather and Climate Research
Weather Forecasting
Digital Content Creation
Semiconductor
Digital Media
Not Specified
Gaming
Retail
Data obtained from top500.org, June 2006

Concepts and Terminology

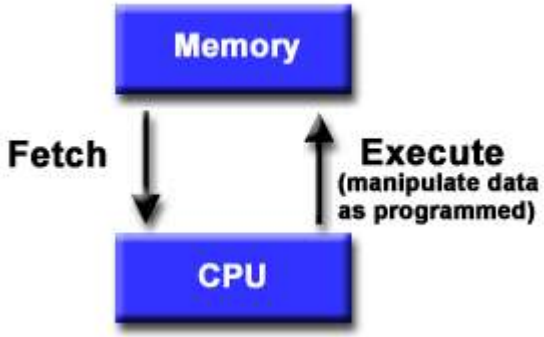
Von Neumann Architecture

- 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.

Basic Design

- 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***.

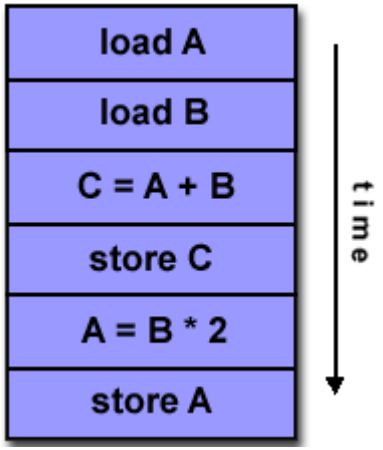
Flynn Matrix

- 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**

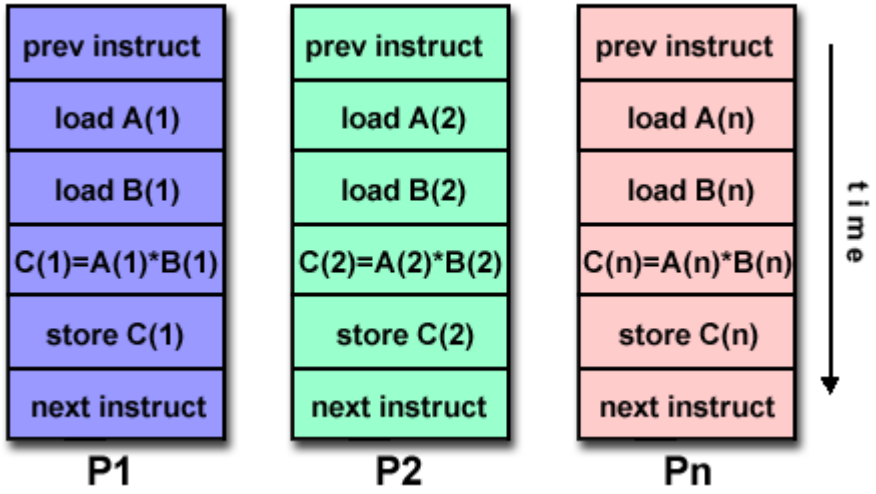
Single Instruction, Single Data (SISD)

- 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 and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes



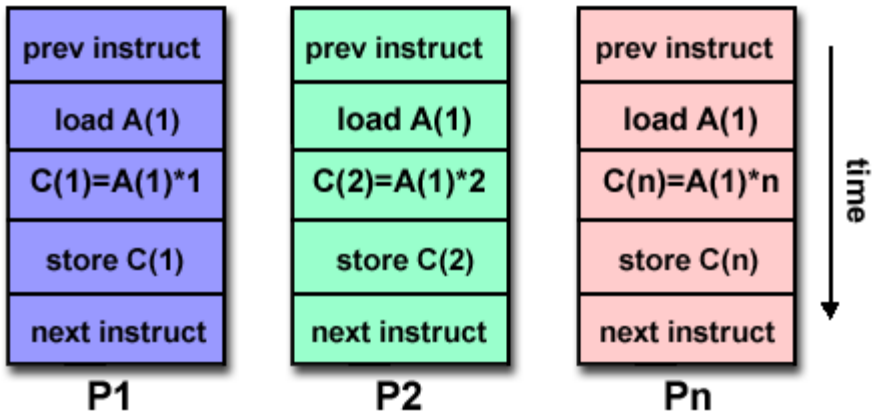
Single Instruction, Multiple Data (SIMD)

- 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
- 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.
- Best suited for specialized problems characterized by a high degree of regularity,such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820



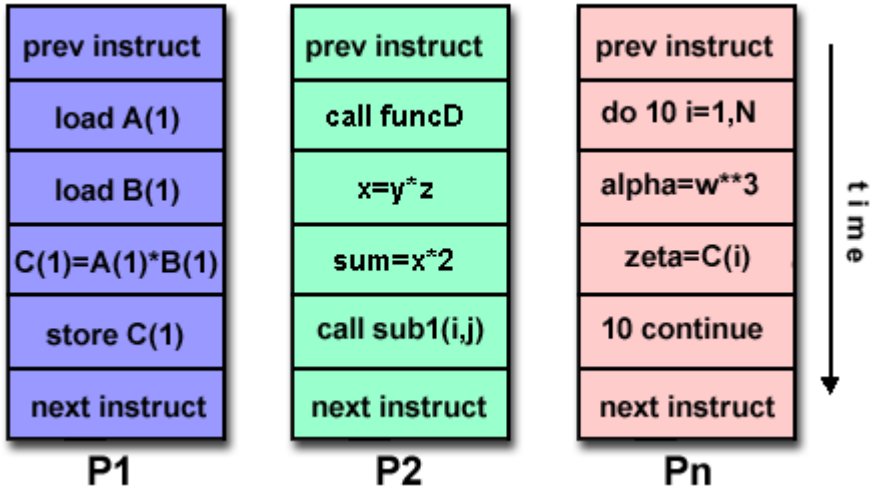
Multiple Instruction, Single Data (MISD)

- 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. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- 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)

- 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 "grids" and multi-processor SMP computers - including some types of PCs.



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.

- **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.
- **Parallel Task**
  - A task that can be executed by multiple processors safely (yields correct results)
- **Serial Execution**
  - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

Some General Parallel Terminology

- **Parallel Execution**
  - Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
- **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.
- **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.

Some General Parallel Terminology

- **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.

Some General Parallel Terminology

- **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
- **Observed Speedup**
  - Observed speedup of a code which has been parallelized, defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

  - One of the simplest and most widely used indicators for a parallel program's performance.

Some General Parallel Terminology

- **Parallel Overhead**
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
    - Task start-up time
    - Synchronizations
    - Data communications
    - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
    - Task termination time
- **Massively Parallel**
  - Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but currently BG/L pushes this number to 6 digits.

Some General Parallel Terminology

- **Scalability**
  - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communications
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application and coding

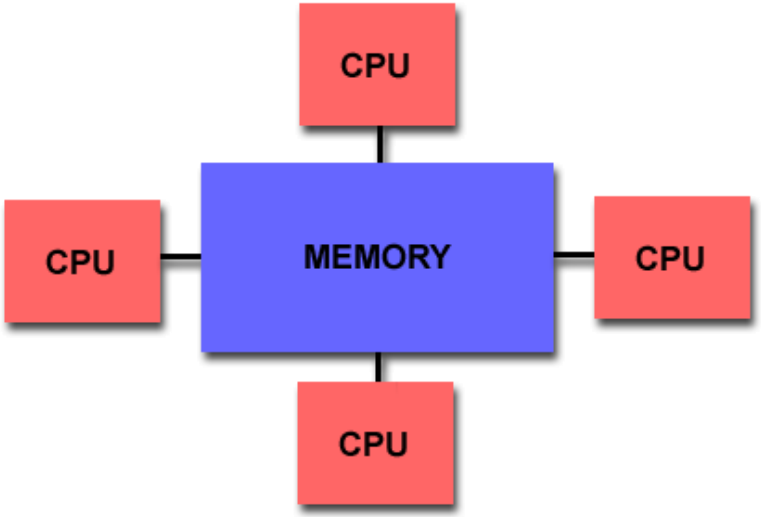
Parallel Computer Memory Architectures

Memory Architectures

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

Shared Memory

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

Shared Memory : UMA vs. NUMA

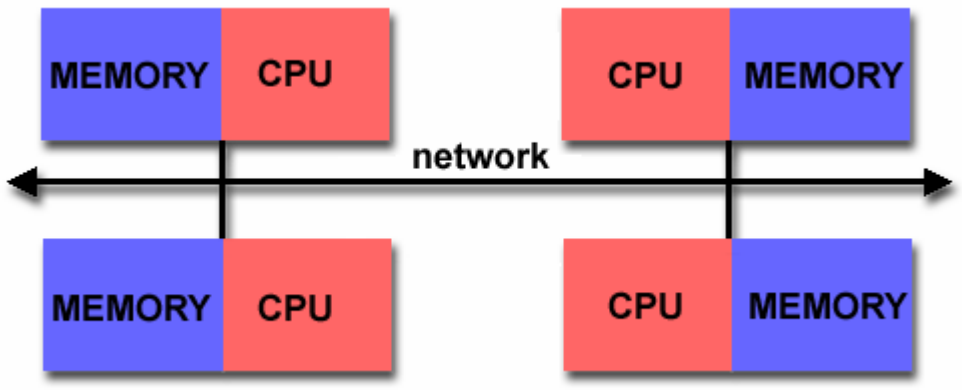
- Uniform Memory Access (UMA):
  - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
  - Identical processors
  - Equal access and access times to memory
  - Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory,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

- 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.
- 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 access to 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: Pro and Con

- 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

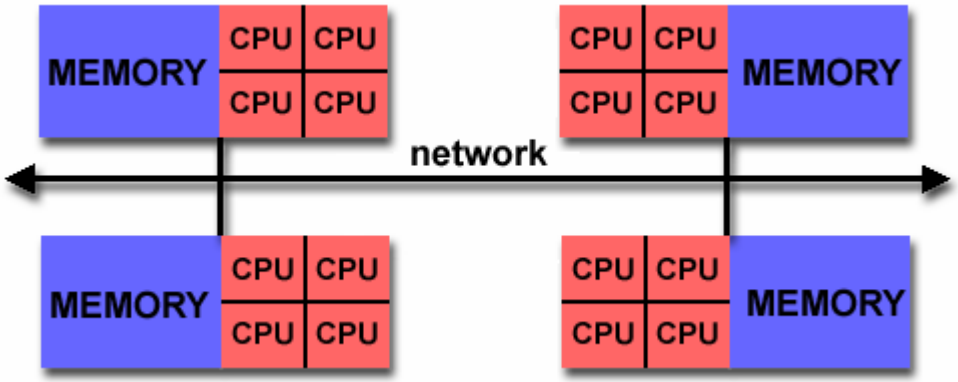
Hybrid Distributed-Shared Memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Hybrid Distributed-Shared Memory

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



- 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.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.

Parallel Programming Models

Parallel Programming Models

- Overview
- Shared Memory Model
- Threads Model
- Message Passing Model
- Data Parallel Model
- Other 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

- 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.
- Shared memory model on a distributed memory machine: Kendall Square Research (KSR) ALLCACHE approach.
  - Machine memory was physically distributed, but appeared to the user as a single shared memory (global address space). Generically, this approach is referred to as "virtual shared memory".
  - Note: although KSR is no longer in business, there is no reason to suggest that a similar implementation will not be made available by another vendor in the future.
  - Message passing model on a shared memory machine: MPI on SGI Origin.
- The SGI Origin employed the CC-NUMA type of shared memory architecture, where every task has direct access to global memory. However, the ability to send and receive messages with MPI, as is commonly done over a network of distributed memory machines, is not only implemented but is very commonly used.

Overview

- 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 sections describe each of the models mentioned above, and also discuss some of their actual implementations.

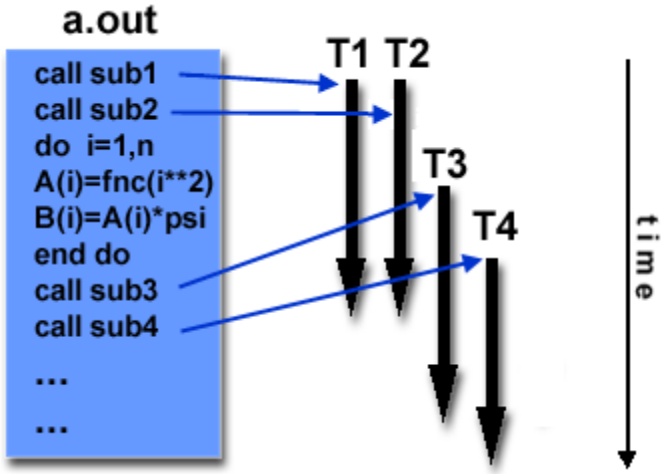
Shared Memory Model

- 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: 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. However, as mentioned previously in the Overview section, the KSR ALLCACHE approach provided a shared memory view of data even though the physical memory of the machine was distributed.

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.
  - 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 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 Model 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 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.
- 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 Model: OpenMP

- **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

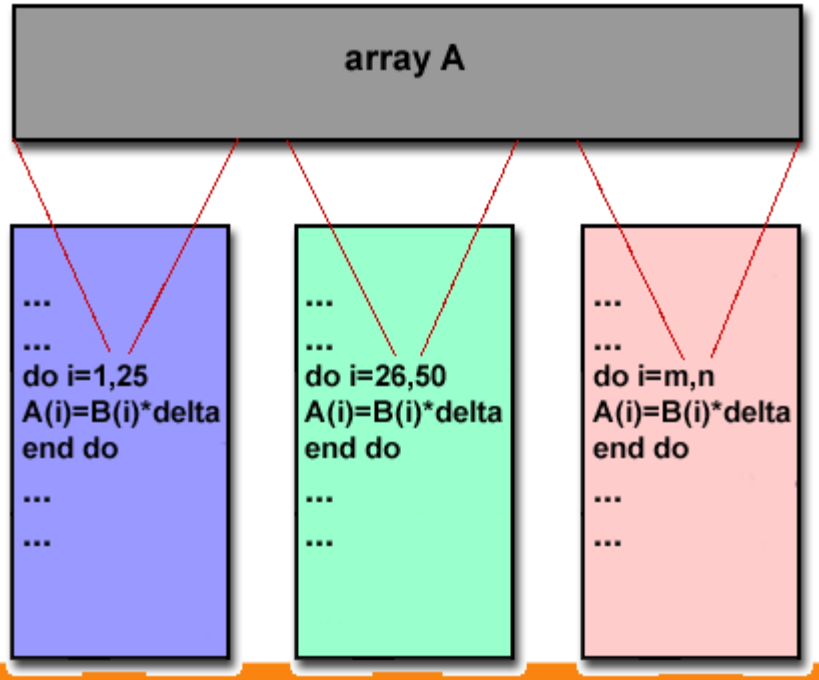
- The message passing model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
  - Tasks exchange data through communications by sending 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.

Message Passing Model Implementations: MPI

- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded 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.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996. Both MPI specifications are available on the web at www.mcs.anl.gov/Projects/mpi/standard.html.

Message Passing Model Implementations: MPI

- 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. A few offer a full implementation of MPI-2.
- 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

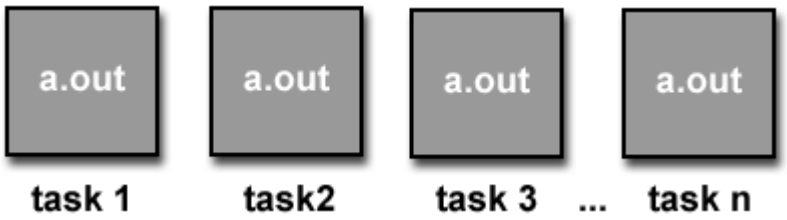
- The data parallel model demonstrates the following characteristics:
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, 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.

Data Parallel Model 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.
- **Fortran 90 and 95 (F90, F95):** ISO/ANSI standard extensions to Fortran 77.
  - Contains everything that is in Fortran 77
  - New source code format; additions to character set
  - Additions to program structure and commands
  - Variable additions - methods and arguments
  - Pointers and dynamic memory allocation added
  - Array processing (arrays treated as objects) added
  - Recursive and new intrinsic functions added
  - Many other new features
- Implementations are available for most common parallel platforms.

Data Parallel Model Implementations

- **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.
- **Compiler Directives:** Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms.
- Distributed memory implementations of this model usually have the compiler convert the program into standard code with calls to a message passing library (MPI usually) to distribute the data to all the processes. All message passing is done invisibly to the programmer.



Other Models

- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software.
- Only three of the more common ones are mentioned here.
  - Hybrid
  - Single Program Multiple Data
  - Multiple Program Multiple Data

Hybryd

- 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.
- Another common example of a hybrid model is combining data parallel with message passing. As mentioned in the data parallel model section previously, data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

Single Program Multiple Data (SPMD)

- 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)

- 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