Multi-core Architecture and Programming

Yang Quansheng(杨全胜)
http://www.njyangqs.com/

School of Computer Science & Engineering
Southeast University

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

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.

What is parallel computing

- *Parallel computing* is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs or 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 or COREs

What is parallel computing

- The compute resources can include:
  - A single processor with multiple core
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of all.

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

Flynn Classical 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*.

SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

Flynn Classical Taxonomy

- 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

Flynn Classical Taxonomy

- 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 image processing.
  - Synchronous (lockstep) and deterministic execution
  - Two varieties: Processor Arrays and Vector Pipelines

Flynn Classical Taxonomy

- 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.

Flynn Classical Taxonomy

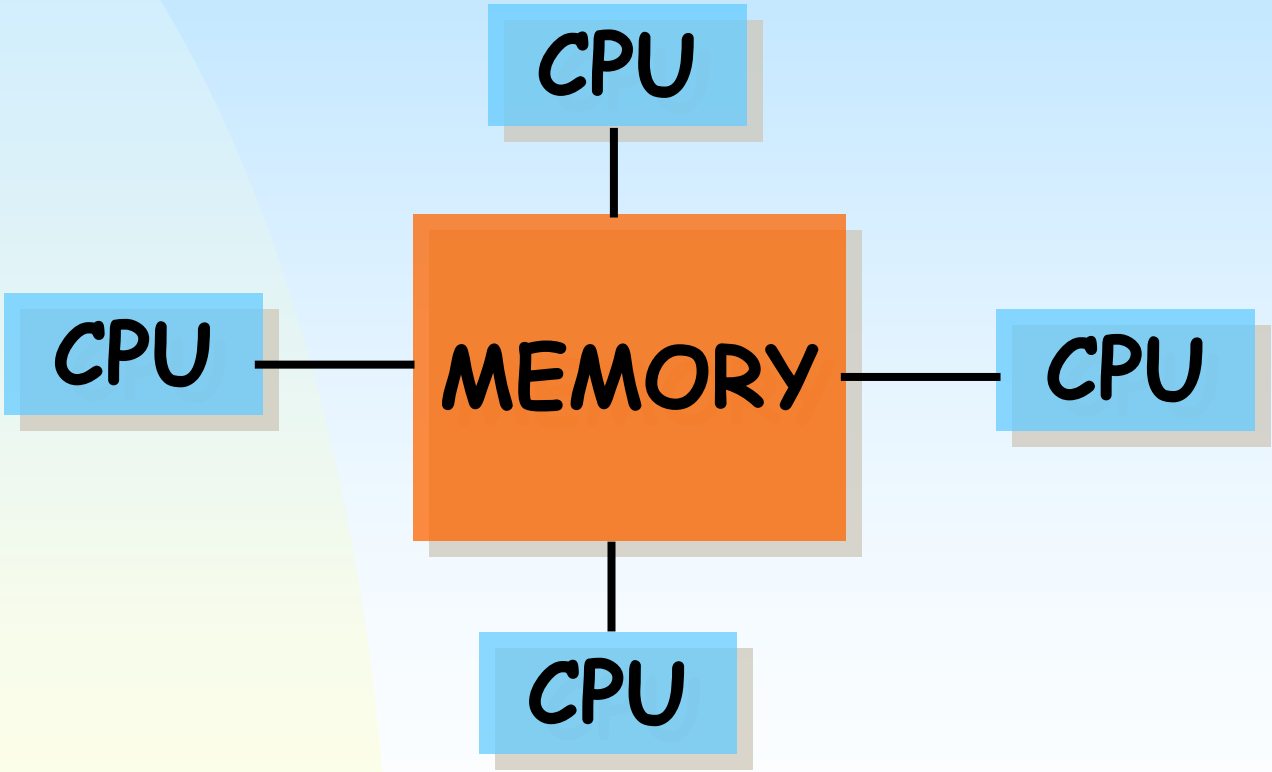
- Multiple Instruction, Multiple Data (MIMD)
  - Currently, the most common type of parallel computer. Most modern computers fall into this category.
  - Multiple Instruction: every processor may be executing a different instruction stream
  - Multiple Data: every processor may be working with a different 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.

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

Parallel Computer Memory Architectures

- Share Memory



Parallel Computer Memory Architectures

- Share 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.

Parallel Computer Memory Architectures

- Share Memory
  - Uniform Memory Access (UMA):
    - Most commonly represented today by Symmetric Multiprocessor (SMP) machines
    - 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.

Parallel Computer Memory Architectures

- Share Memory
  - 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

Parallel Computer Memory Architectures

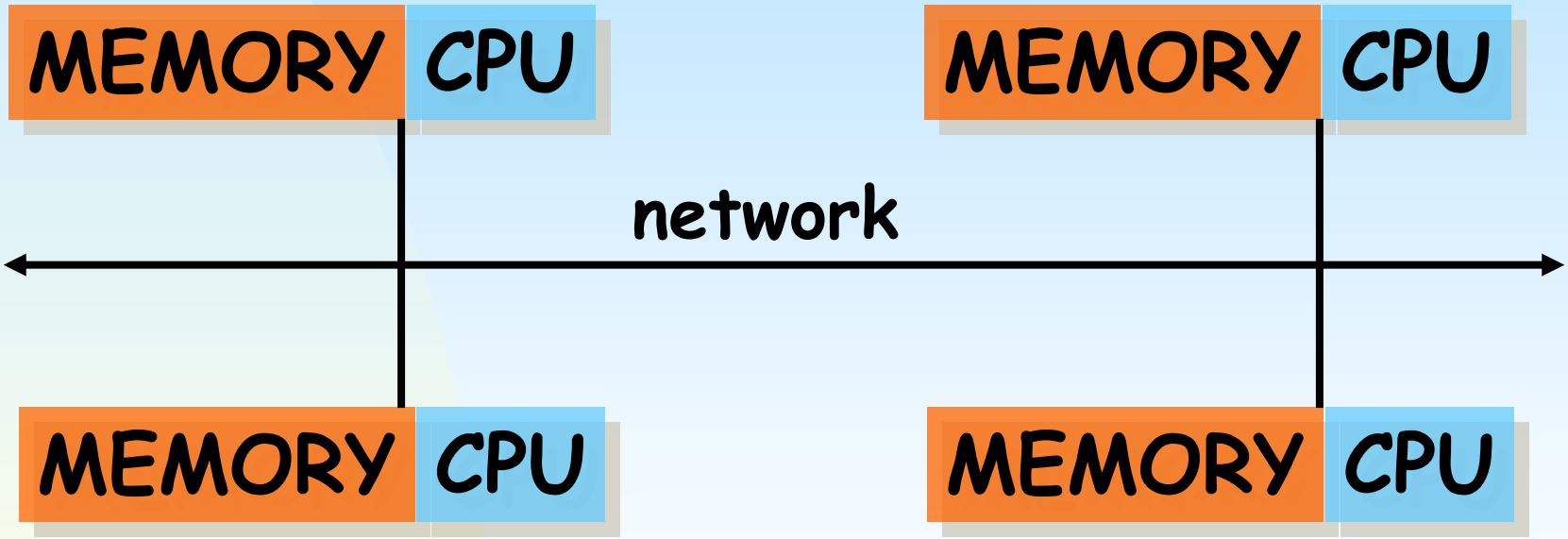
- Share Memory
  - Advantages
    - Global address space provides a user-friendly programming perspective to memory
    - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Parallel Computer Memory Architectures

- Share Memory
  - Disadvantages
    - Adding more CPUs can geometrically increases traffic on the shared memory-CPU path
    - For cache coherent systems, adding more CPUs can geometrically increase traffic associated with cache/memory management
    - Programmer responsibility for synchronization constructs that insure "correct" access of global memory.

Parallel Computer Memory Architectures

- Distributed Memory



Parallel Computer Memory Architectures

- Distributed 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. 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.

Parallel Computer Memory Architectures

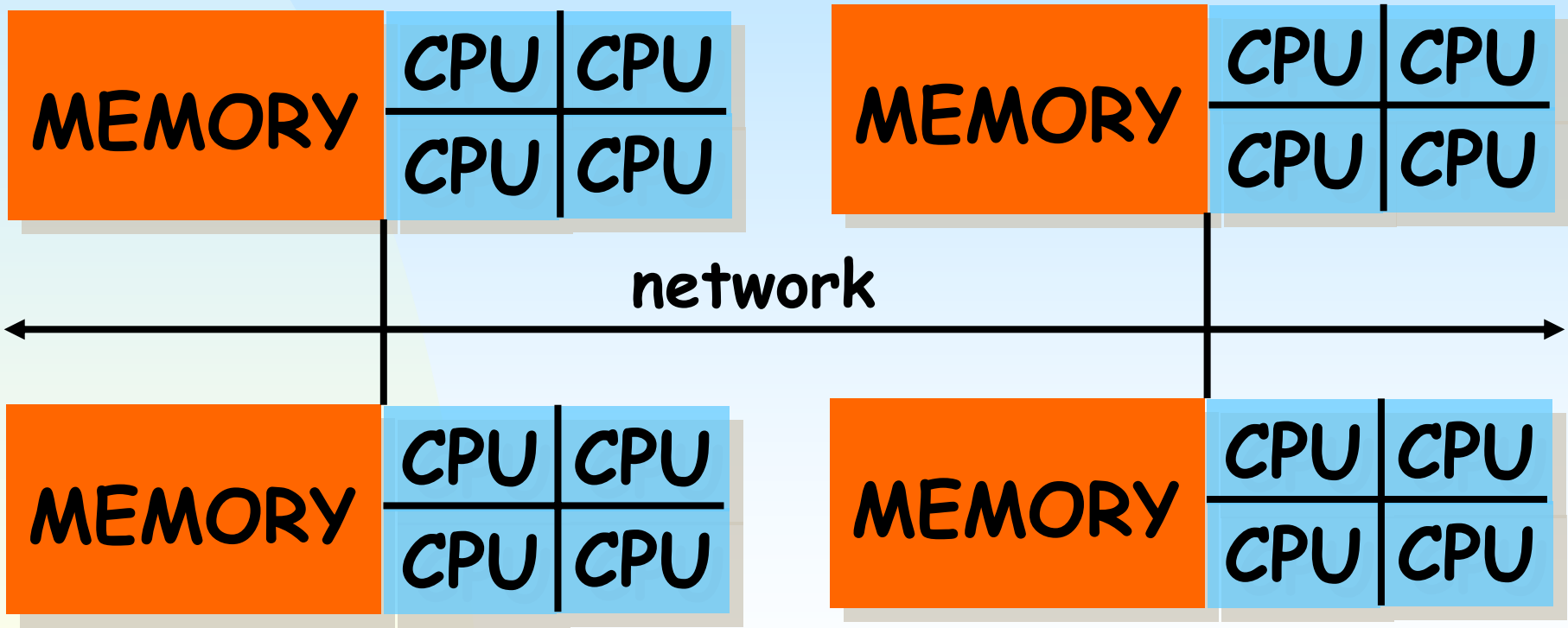
- Distributed Memory
  - Advantages
    - Increase the number of processors and the size of memory increases proportionately.
    - Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain cache coherency.
    - Cost effectiveness: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

Parallel Computer Memory Architectures

- Distributed Memory
  - 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

Parallel Computer Memory Architectures

- Hybrid Distributed-Shared Memory



Parallel Computer Memory Architectures

- Hybrid Distributed-Shared Memory
  - 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.

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

Parallel Programming Models

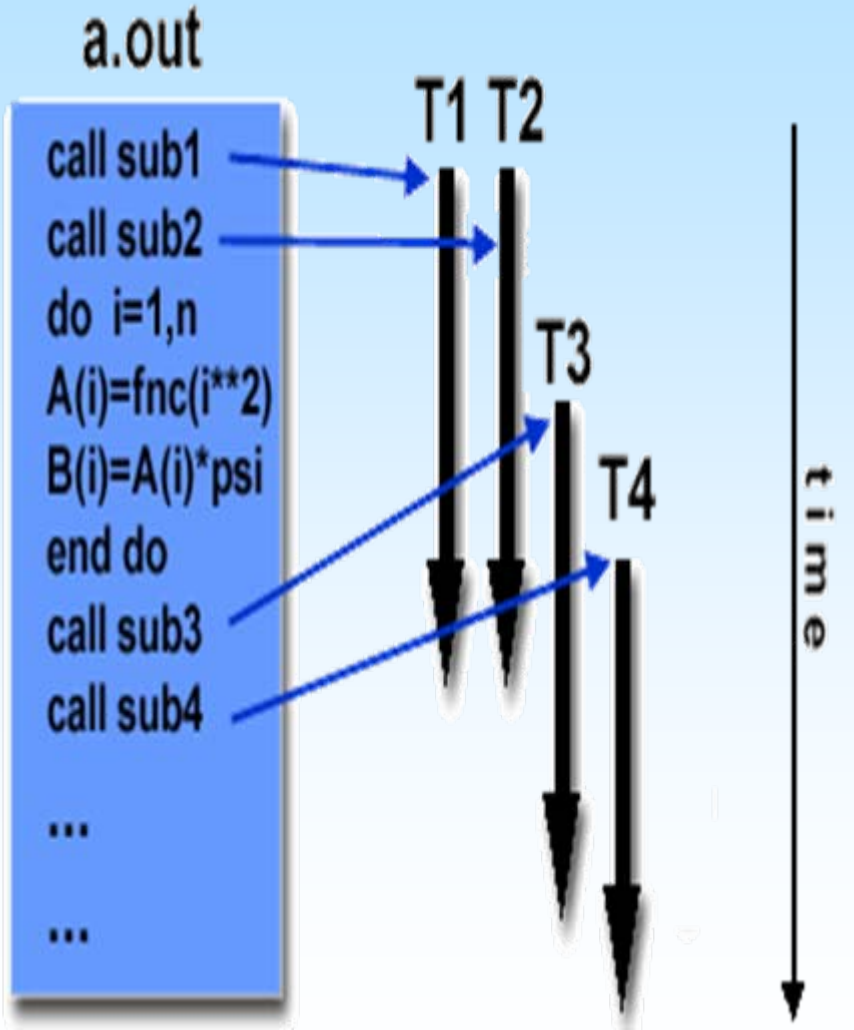
- Shared Memory Model
  - In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
  - Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
  - An advantage of this model from the programmer's point 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.

Parallel Programming Models

- 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, the KSR ALLCACHE approach provided a shared memory view of data even though the physical memory of the machine was distributed.

Parallel Programming Models

- Threads Model
  - In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
  - Threads are commonly associated with shared memory architectures and operating systems.



Parallel Programming Models

- 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.
    - There are two very different implementations of threads: *POSIX Threads* and *OpenMP*.

Parallel Programming Models

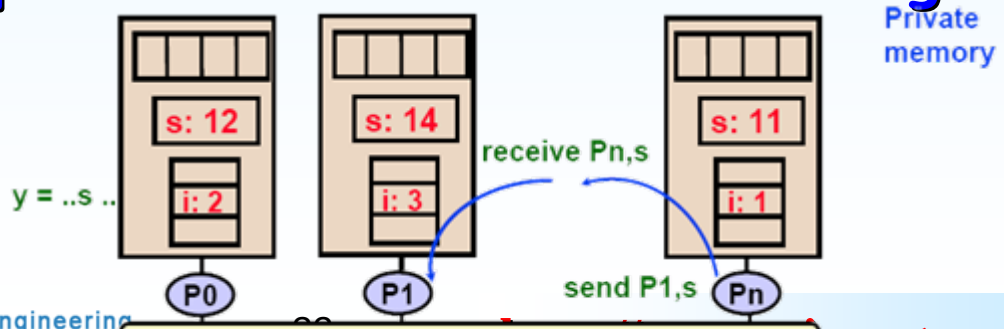
- Threads Model
  - POSIX Threads
    - Library based; requires parallel coding
    - Specified by the IEEE POSIX 1003.1c standard (1995).
    - C Language only
    - Commonly referred to as Pthreads.
    - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
    - Very explicit parallelism; requires significant programmer attention to detail.

Parallel Programming Models

- Threads Model
  - 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"

Parallel Programming Models

- Message Passing Model
  - 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.



Parallel Programming Models

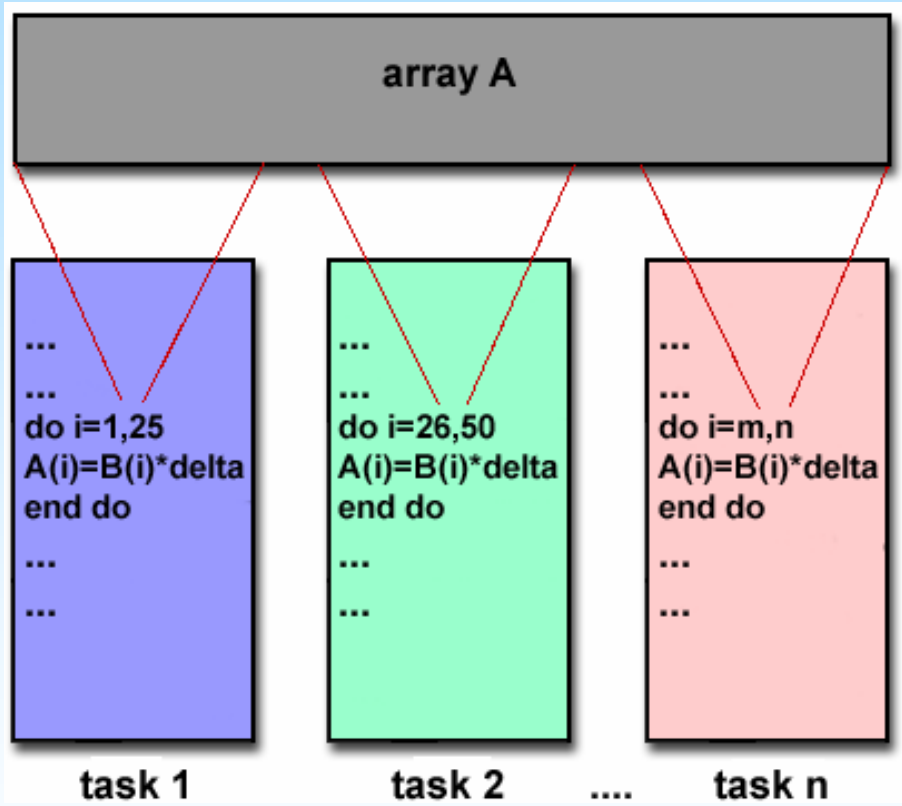
- Message Passing Model
  - Implementations:
    - In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
    - MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations 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.

Parallel Programming Models

- Data Parallel Model
  - 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".

Parallel Programming Models

- Data Parallel Model
  - 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.



Parallel Programming Models

- 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)
    - 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.

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

Designing Parallel Programs

- Manual Parallelization
  - The programmer is typically responsible for both identifying and actually implementing parallelism.
  - Very often, manually developing parallel codes is a time consuming, complex, error-prone and *iterative* process.

Designing Parallel Programs

- Automatic Parallelization
  - Fully Automatic
    - The compiler analyzes the source code and identifies opportunities for parallelism.
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - Programmer Directed
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.

Designing Parallel Programs

- Automatic Parallelization
  - Problem
    - Wrong results may be produced
    - Performance may actually degrade
    - Much less flexible than manual parallelization
    - Limited to a subset (mostly loops) of code
    - May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex
    - Most automatic parallelization tools are for Fortran

Designing Parallel Programs

- The step in developing parallel software
  1. First, understand the problem that you wish to solve in parallel.
  2. Determine whether or not the problem is one that can actually be parallelized.
     - Calculate the potential energy for each of several thousand independent conformations of a molecule. (can be parallelized)
     - Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: F(k + 2) = F(k + 1) + F(k) (can not be parallllelize

Designing Parallel Programs

- The step in developing parallel software
  3. Identify the program's *hotspots*:
     - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
     - Profilers and performance analysis tools can help
     - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

Designing Parallel Programs

- The step in developing parallel software
  4. Identify *bottlenecks* in the program
     - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred?
     - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
  5. Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*.
  6. Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Designing Parallel Programs

- Partitioning
  - break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
  - There are two basic ways to partition computational work among parallel tasks
    - Domain Decomposition: the data associated with a problem is decomposed. Each parallel task then works on a portion of the data.
    - Functional Decomposition: The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

Designing Parallel Programs

- Types of Synchronization:
  - Barrier
    - Usually implies that all tasks are involved
    - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
    - When the last task reaches the barrier, all tasks are synchronized.
    - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

Designing Parallel Programs

- Types of Synchronization:
  - Lock / semaphore
    - Can involve any number of tasks
    - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
    - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
    - Other tasks can also acquire the lock but must wait until the task that owns the lock releases it.
    - Can be blocking or non-blocking

Designing Parallel Programs

- Types of Synchronization:
  - Synchronous communication operations
    - Involves only those tasks executing a communication operation
    - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

Designing Parallel Programs

- Data Dependencies
  - A *dependence* exists between program statements when the order of statement execution affects the results of the program.
  - A *data dependence* results from multiple use of the same location(s) in storage by different tasks.
  - Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

Designing Parallel Programs

- Data Dependencies
  - Loop carried data dependence

```
for (i = 1;i<100;i++)
  a[i] = a[i-1]*32
```

  - Loop independent data dependence

```
 task 1      task 2
 ------      ------
 X = 2       X = 4
   .           .
   .           .
 Y = X*2     Y = X*3
```

Designing Parallel Programs

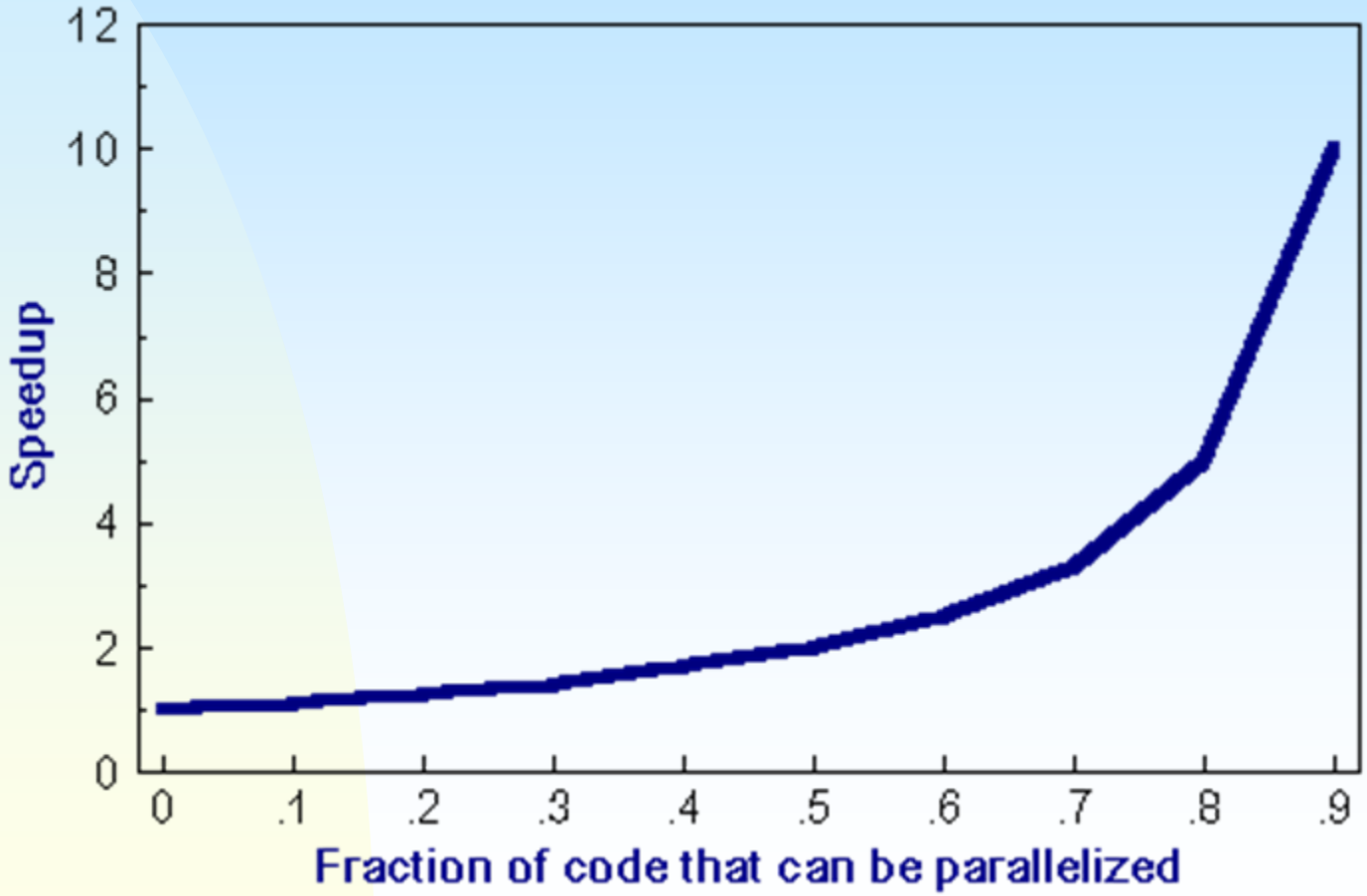
- Data Dependencies
  - How to Handle Data Dependencies:
    - Distributed memory architectures - communicate required data at synchronization points.
    - Shared memory architectures -synchronize read/write operations between tasks.

Basics of parallel computing

- Content
  - What is parallel computing
  - Flynn Classical Taxonomy
  - Parallel Computer Memory Architectures
  - Parallel Programming Models
  - Designing Parallel Programs
  - Performance valuation

Performance valuation

- Amdahl's Law



Performance valuation

- Amdahl's Law
  - Potential program speedup is defined by the fraction of code (P) that can be parallelized speedup = 1/(1-P)
  - If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup). If all of the code is parallelized, P = 1 and the speedup is infinite (in theory). If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

Performance valuation

- Amdahl's Law
  - Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$speedup = \frac{1}{\frac{P}{N} + S}$$

where P = parallel fraction, N = number of processors and S = serial fraction.

Performance valuation

- Optimize the performance of program
  - For the serial program
    - Call the high performance library function, just like BLAS, FFTW, etc.
    - Using the proper option for optimize in compiler.
    - The dimension of array must be proper
    - The order of nested loop is important to the locality of data accessing.
    - Expand the looping

Performance valuation

- Optimize the performance of program
  - For the parallel program
    - Reduce the traffic, enlarge the communication granularity
    - Using the high performance arithmetic for the global communications
    - Advance the degree of parallelism, let CPU always busy.
    - Load balance
    - Overlap the communication and computing.
    - Reduce the traffic via repeat the computing