

CHAPTER 1

Introduction Parallel Computing

What is parallel computing ?

- **Parallel computing** is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel")

What is parallel computing ?

There are several different forms of parallel computing:

1. bit-level
2. instruction level
3. task parallelism

Background of Parallel Computing

- Traditionally, computer software has been written for serial computation.
- To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions.
- These instructions are executed on a central processing unit on one computer.
- Only one instruction may execute at a time—after that instruction is finished, the next is executed.

Background of Parallel Computing

- Parallel computing, uses multiple processing elements simultaneously to solve a problem.
- This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others.
- The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.

Amdhal's dan Gustafson's

- The potential speed-up of an algorithm on a parallel computing platform is given by Amdahl's law, originally formulated by Gene Amdahl in the 1960s.
- It states that a small portion of the program which cannot be parallelized will limit the overall speed-up available from parallelization.

- A program solving a large mathematical or engineering problem will typically consist of several parallelizable parts and sequential parts. If α is the fraction of running time a program spends on non-parallelizable parts, then:

$$\lim_{P \rightarrow \infty} \frac{1}{\frac{1-\alpha}{P} + \alpha} = \frac{1}{\alpha}$$

is the maximum speed-up with parallelization of the program. If the sequential portion of a program accounts for 10% of the runtime (), we can get no more than a 10 \times speed-up, regardless of how many processors are added. This puts an upper limit on the usefulness of adding more parallel execution units.

- Gustafson's law is another law in computing, closely related to Amdahl's law. It states that the speedup with processors is

$$S(P) = P - \alpha(P - 1) = \alpha + P(1 - \alpha).$$

- Both Amdahl's law and Gustafson's law assume that the running time of the sequential portion of the program is independent of the number of processors.
- Amdahl's law assumes that the entire problem is of fixed size so that the total amount of work to be done in parallel is also *independent of the number of processors*, whereas Gustafson's law assumes that the total amount of work to be done in parallel *varies linearly with the number of processors*.

Race conditions, mutual exclusion, synchronization, and parallel slowdown

- Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. For example, consider the following program

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A: Write back to variable V	3B: Write back to variable V

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A: Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

- One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again.
- This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

- Locking multiple variables using non-atomic locks introduces the possibility of program deadlock.
- An atomic lock locks multiple variables all at once.
- If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

- Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier.
- Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

- Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other.
- Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

Fine-grained, coarse-grained, and embarrassing parallelism

- Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second; it exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate. Embarrassingly parallel applications are considered the easiest to parallelize.

Consistency models

- Parallel programming languages and parallel computers must have a consistency model (also known as a memory model). The consistency model defines rules for how operations on computer memory occur and how results are produced.

- One of the first consistency models was Leslie Lamport's sequential consistency model. Sequential consistency is the property of a parallel program that its parallel execution produces the same results as a sequential program. Specifically, a program is sequentially consistent if "... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program".

- Software transactional memory is a common type of consistency model. Software transactional memory borrows from database theory the concept of atomic transactions and applies them to memory accesses.

Flynn's taxonomy

- Michael J. Flynn created one of the earliest classification systems for parallel (and sequential) computers and programs, now known as Flynn's taxonomy.
- Flynn classified programs and computers by whether they were operating using a single set or multiple sets of instructions, and whether or not those instructions were using a single set or multiple sets of data.

Flynn's taxonomy

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

- The single-instruction-single-data (SISD) classification is equivalent to an entirely sequential program. The single-instruction-multiple-data (SIMD) classification is analogous to doing the same operation repeatedly over a large data set.

- This is commonly done in signal processing applications. Multiple-instruction-single-data (MISD) is a rarely used classification. While computer architectures to deal with this were devised (such as systolic arrays), few applications that fit this class materialized. Multiple-instruction-multiple-data (MIMD) programs are by far the most common type of parallel programs.

Types of parallelism

Bit-level parallelism

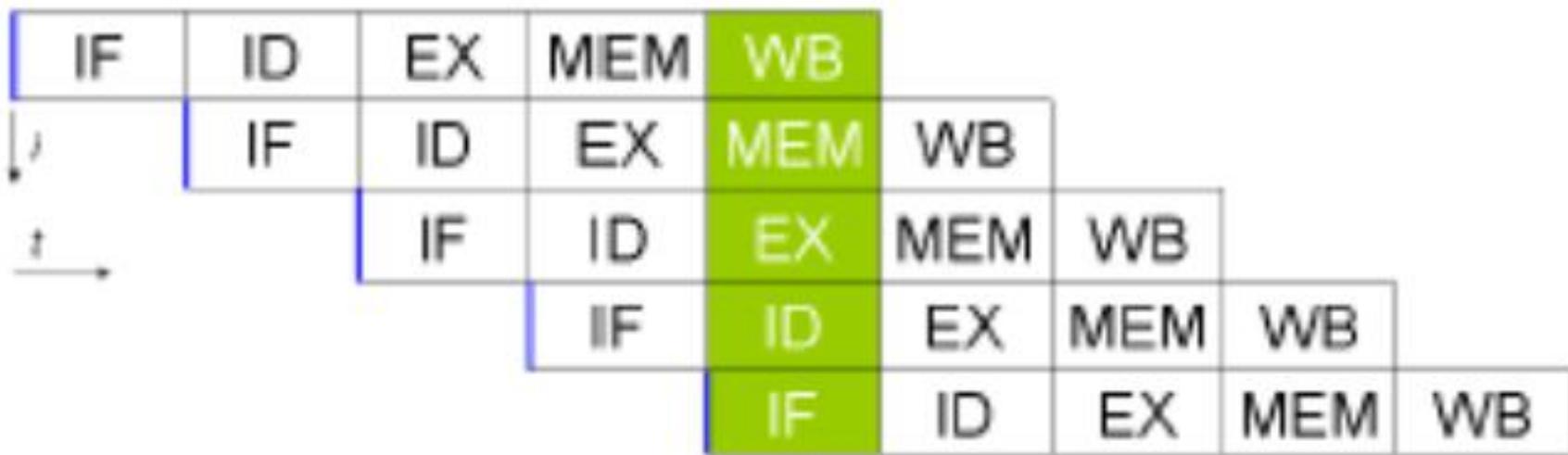
- From the advent of very-large-scale integration (VLSI) computer-chip fabrication technology in the 1970s until about 1986, speed-up in computer architecture was driven by doubling computer word size—the amount of information the processor can manipulate per cycle. Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose sizes are greater than the length of the word.

- For example, where an 8-bit processor must add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction, then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition; thus, an 8-bit processor requires two instructions to complete a single operation, where a 16-bit processor would be able to complete the operation with a single instruction.

- Historically, 4-bit microprocessors were replaced with 8-bit, then 16-bit, then 32-bit microprocessors. This trend generally came to an end with the introduction of 32-bit processors, which has been a standard in general-purpose computing for two decades. Not until recently (c. 2003–2004), with the advent of x86-64 architectures, have 64-bit processors become commonplace.

Types of parallelism

Instruction-level parallelism



A canonical five-stage pipeline in a RISC machine (IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back)

- A computer program, is in essence, a stream of instructions executed by a processor. These instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Advances in instruction-level parallelism dominated computer architecture from the mid-1980s until the mid-1990s.

- Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action the processor performs on that instruction in that stage; a processor with an N -stage pipeline can have up to N different instructions at different stages of completion. The canonical example of a pipelined processor is a RISC processor, with five stages: instruction fetch, decode, execute, memory access, and write back. The Pentium 4 processor had a 35-stage pipeline.

	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
i		IF	ID	EX	MEM	WB
t		IF	ID	EX	MEM	WB
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	
	IF	ID	EX	MEM	WB	

A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle. It can have two instructions in each stage of the pipeline, for a total of up to 10 instructions (shown in green) being simultaneously executed.

- In addition to instruction-level parallelism from pipelining, some processors can issue more than one instruction at a time. These are known as superscalar processors. Instructions can be grouped together only if there is no data dependency between them. Scoreboarding and the Tomasulo algorithm (which is similar to scoreboardding but makes use of register renaming) are two of the most common techniques for implementing out-of-order execution and instruction-level parallelism.

Types of parallelism

Task parallelism

- Task parallelism is the characteristic of a parallel program that "entirely different calculations can be performed on either the same or different sets of data".
- This contrasts with data parallelism, where the same calculation is performed on the same or different sets of data. Task parallelism does not usually scale with the size of a problem.

INTRODUCTION to PARALLEL COMPUTING

Sequential Processes vs Parallel Processes

- ***Sequential processes*** are those that occur in strict order, where it is not possible to the next step until the current one is completed.
- ***Parallel processes*** are those in which many events can happen simultaneously or exhibit concurrencies.

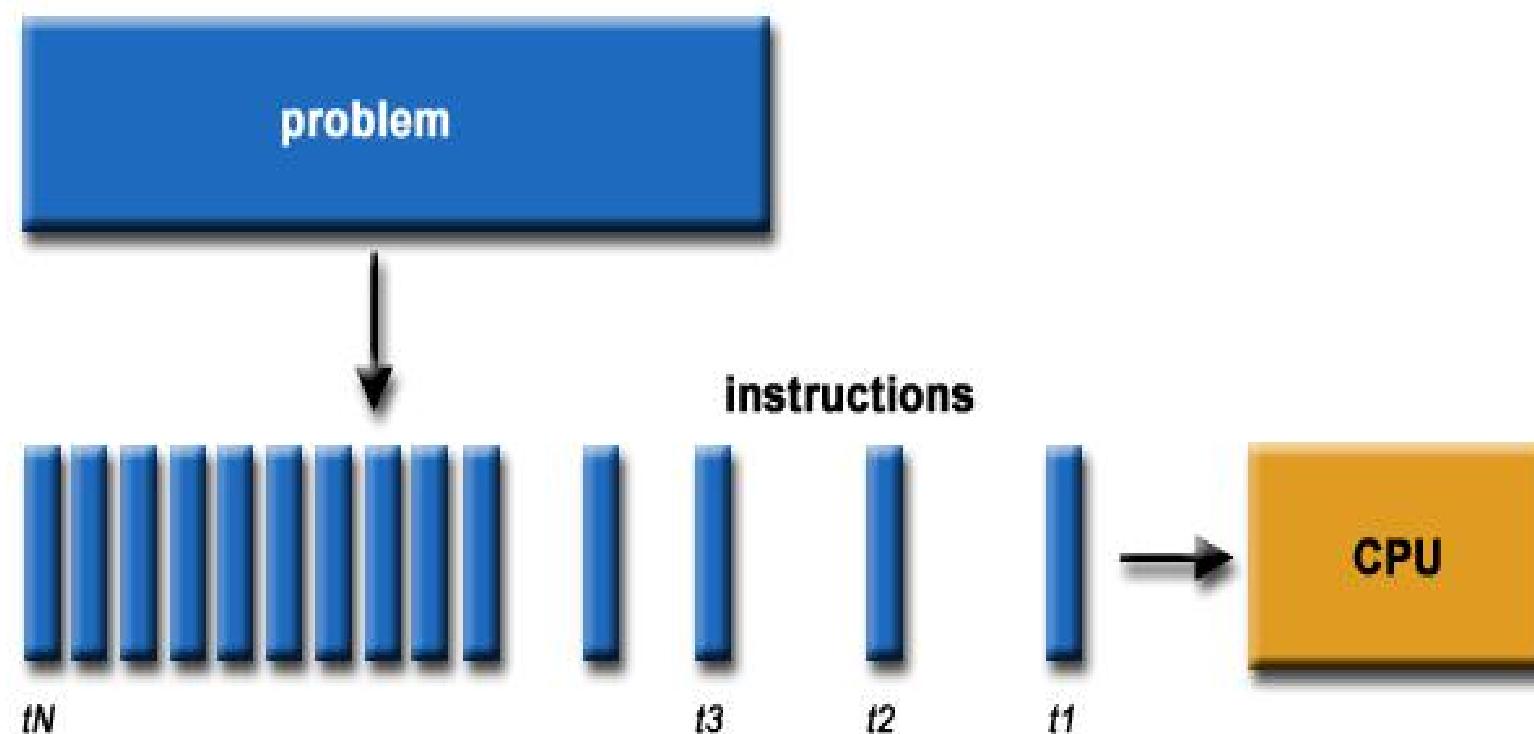
Parallel Computing

- A parallel computer is a “Collection of processing elements that communicate and co-operate to solve large problems fast”.

Conventional 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.

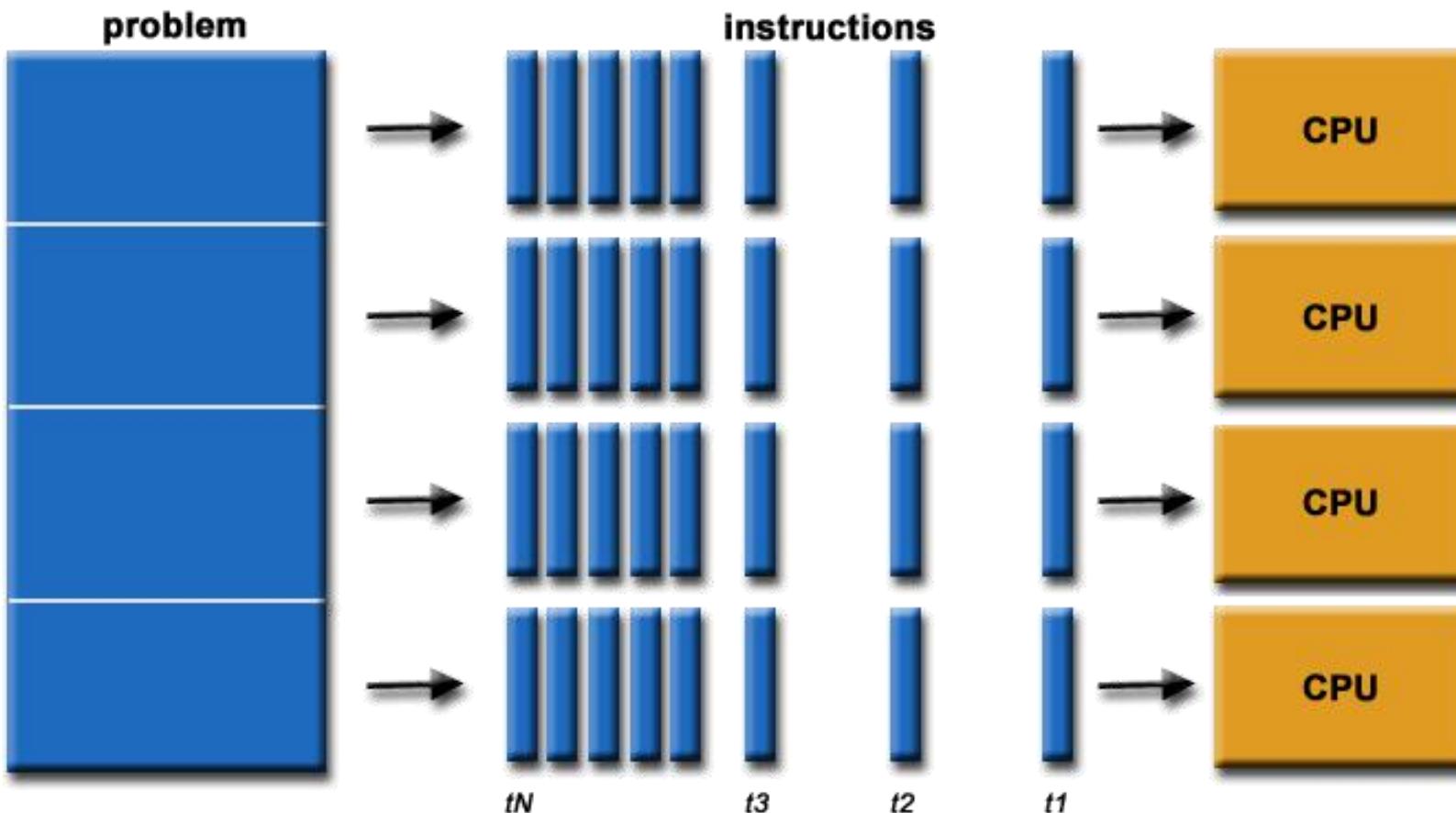
Conventional Computing



Parallel Computing

- 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



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
 - 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.

Parallel Computing: what for? (2)

- 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? (3)

- 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

Parallel Computing: what for? (4)

- 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 legitimate 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.

Von Neumann Architecture

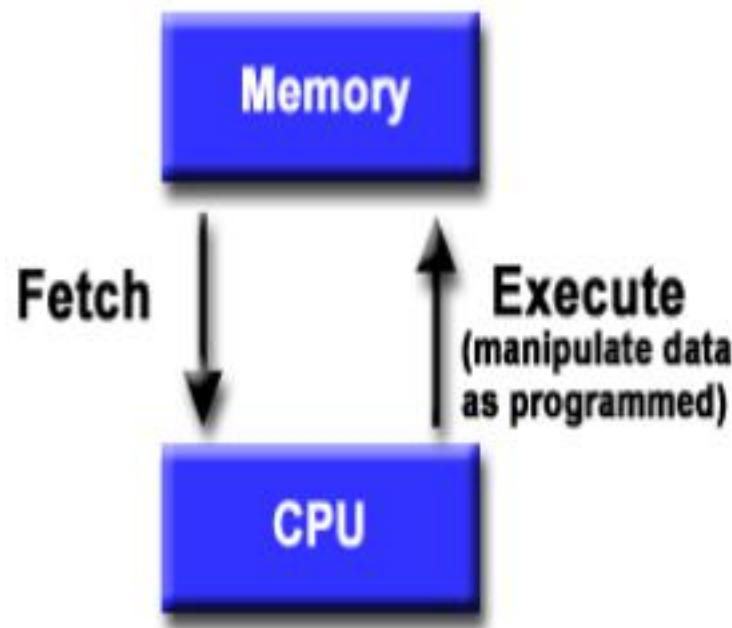
- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.
- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read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

Basic Design

- 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***.

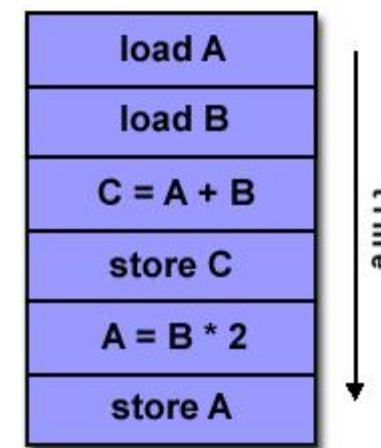
Flynn Matrix

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

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

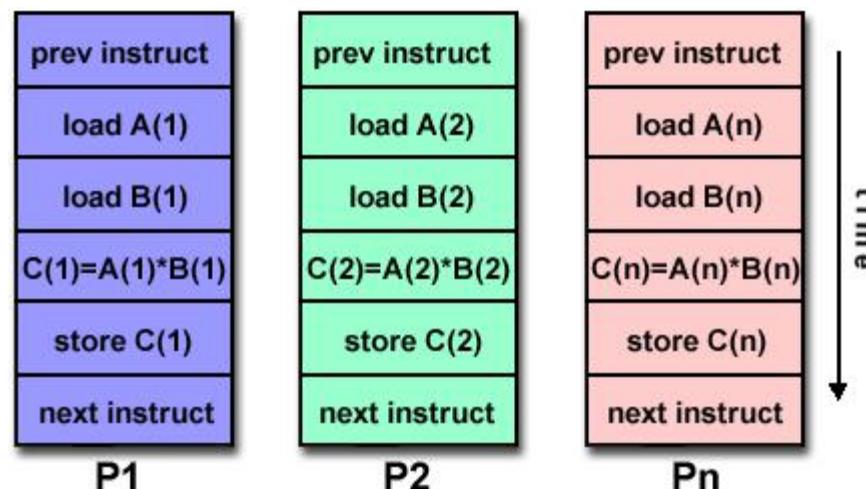


Single Instruction, Multiple Data (SIMD)

- 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.

Single Instruction, Multiple Data (SIMD)

- 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

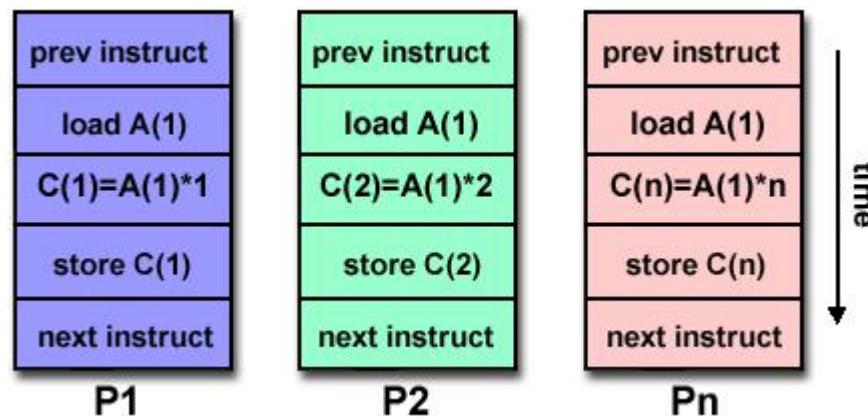


Multiple Instruction, Single Data (MISD)

- 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).

Multiple Instruction, Single Data (MISD)

- Some conceivable uses might be:
 - multiple frequency filters operating on a single signal stream
 - multiple cryptography algorithms attempting to crack a single coded message.

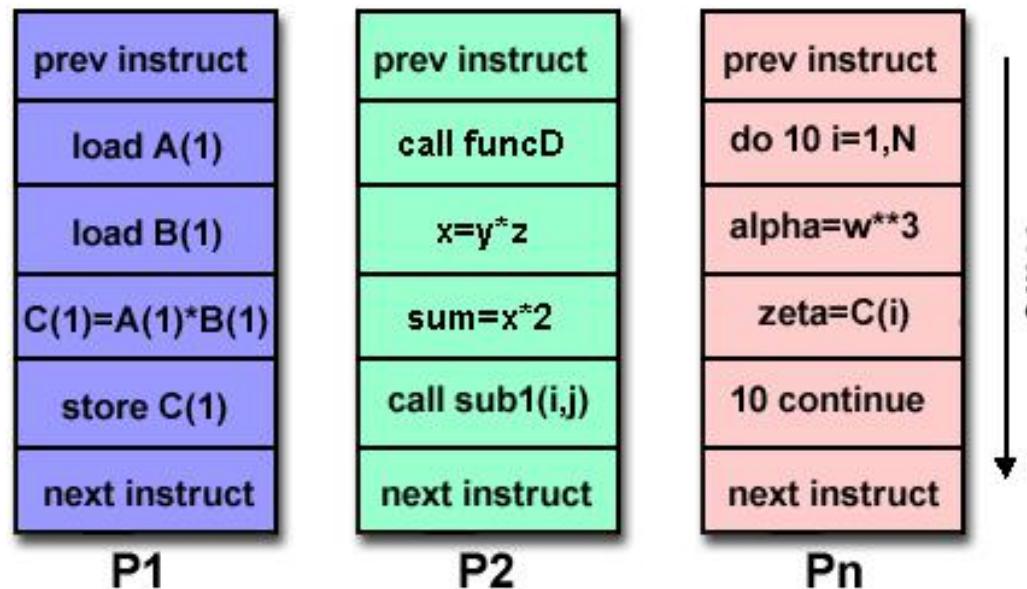


Multiple Instruction, Multiple Data (MIMD)

- 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

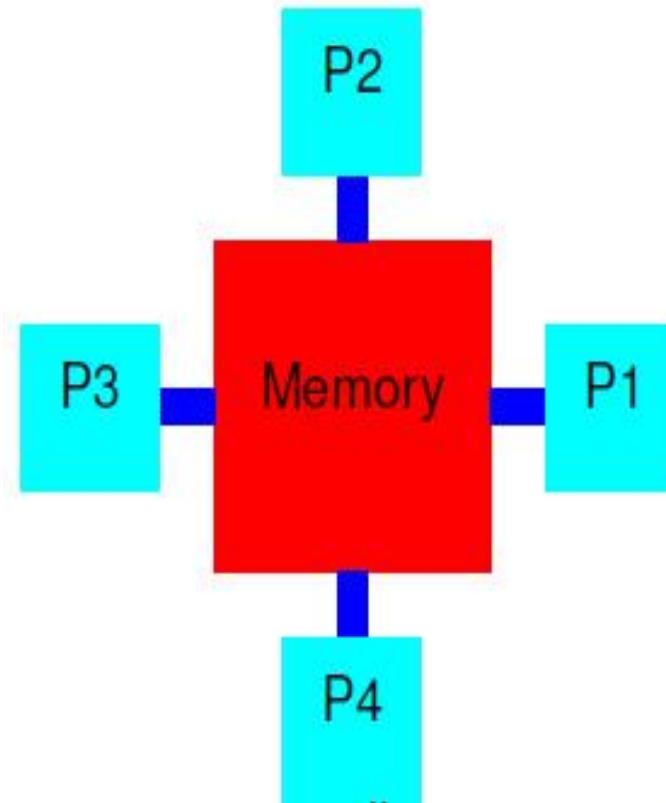
Multiple Instruction, Multiple Data (MIMD)

- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.



Shared Memory Processors

- All processors access all memory as global address space.
- Processors operate independently but share memory resources.

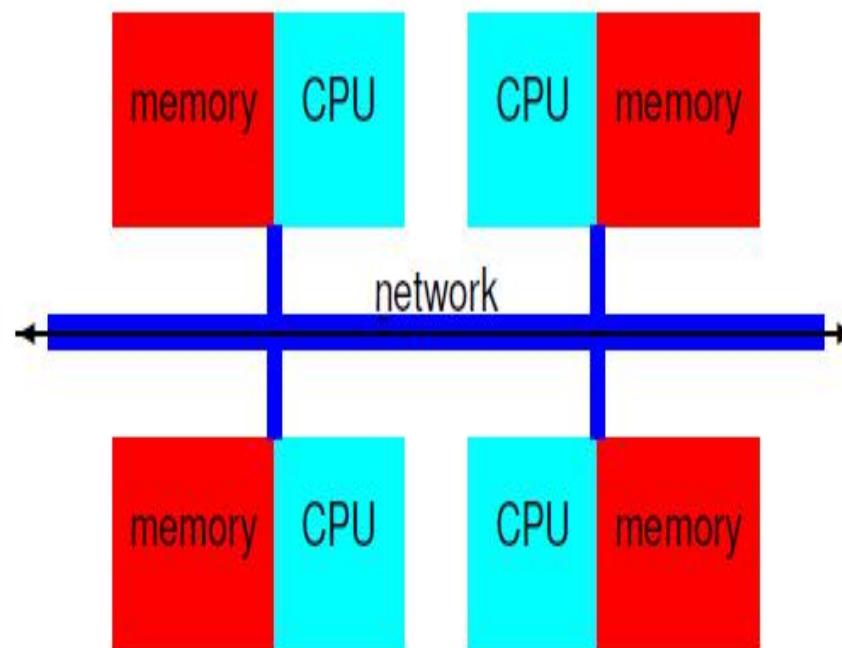


Shared Memory Processors: General Characteristics

- **Advantages:**
 - a. Global address space simplified programming.
 - b. Allow incremental parallelization.
 - c. Data sharing between CPUs fast and uniform.
- **Disadvantages:**
 - a. Lack of memory scalability between memory and CPU.
 - b. Increasing CPUs increase memory traffic geometrically on shared memory-CPU paths.
 - c. Programmer responsible for synchronization of memory accesses.
 - d. Soaring expense of internal network.

Distributed Memory

- Each processor has its own private memory.
- No global address space.
- Network access to communicate between processors.

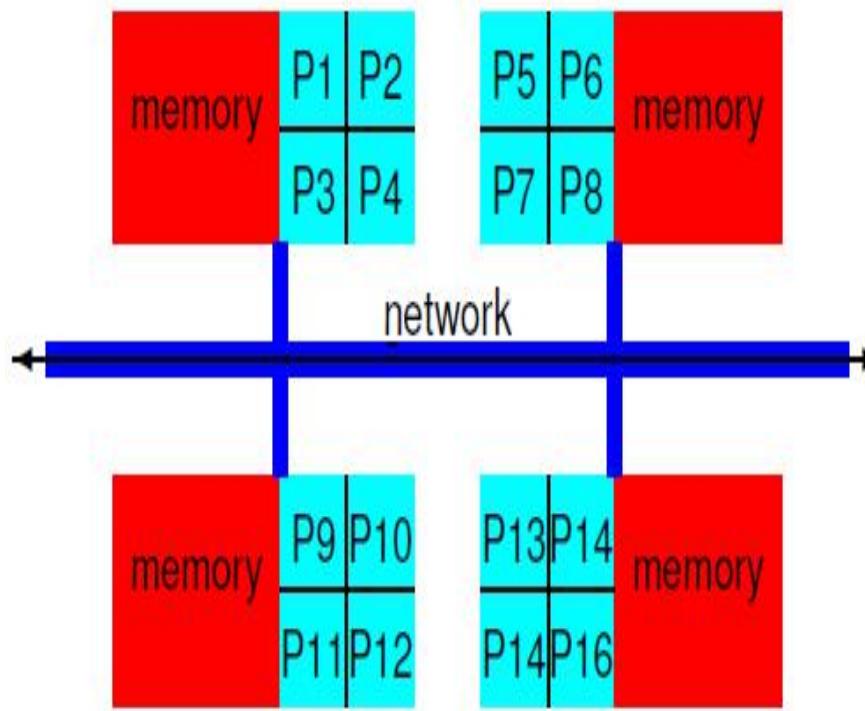


Distributed Memory Processors: General Characteristics

- **Advantages:**
 - a. Memory size scales with CPUs.
 - b. Fast local memory access no network interference.
 - c. Cost effective (commodity components).
- **Disadvantages:**
 - a. Programmer responsible for communication details.
 - b. Difficult to map existing data structure, based on global memory, to this memory organization.
 - c. Non-uniform memory access time.
 - d. All or nothing parallelization.

Hybrid Distributed-Shared Memory

- Most common type of current parallel computers.
- Shared memory component is a CC-UMA SMP.
- Local global address space within each SMP.

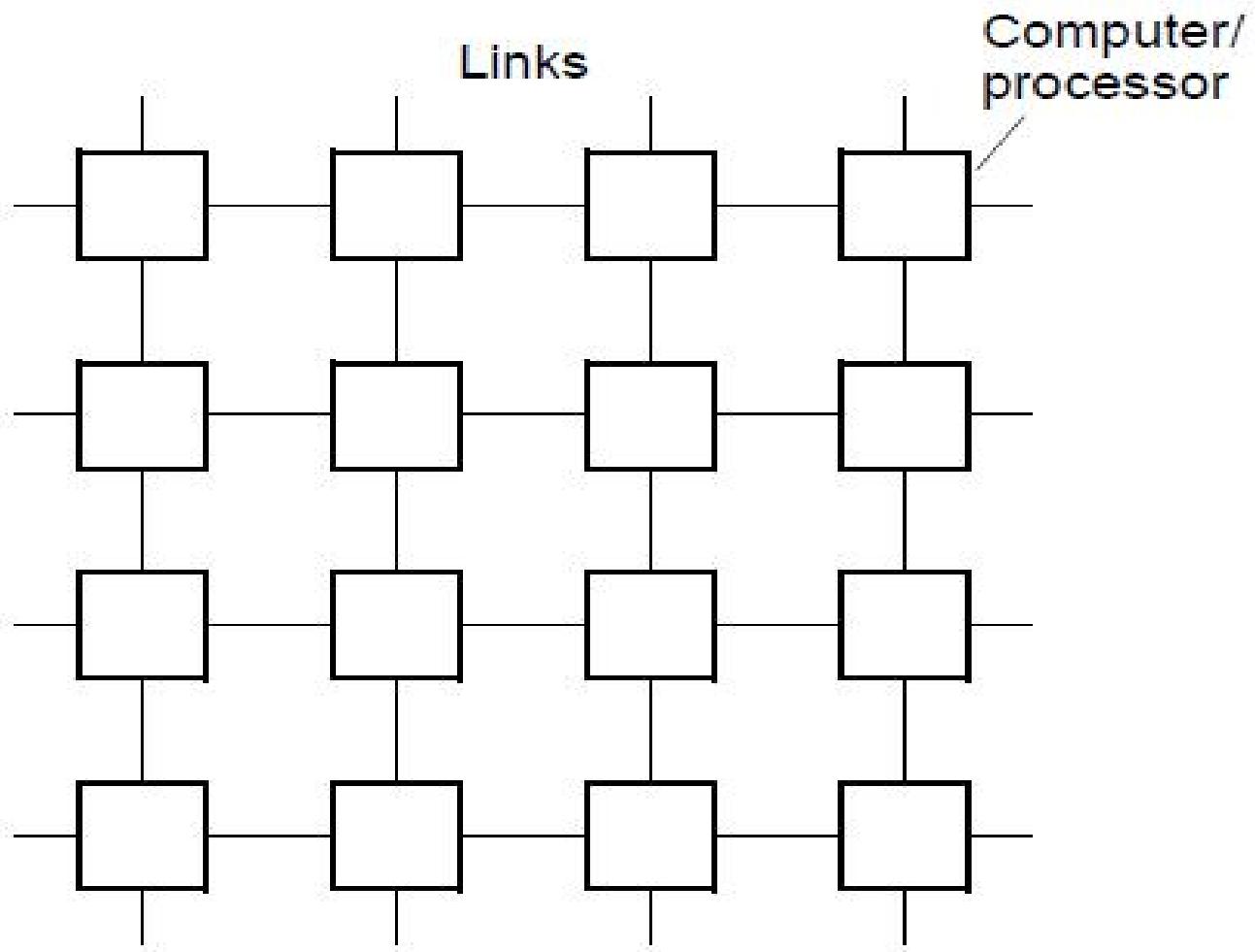


Interconnection Networks

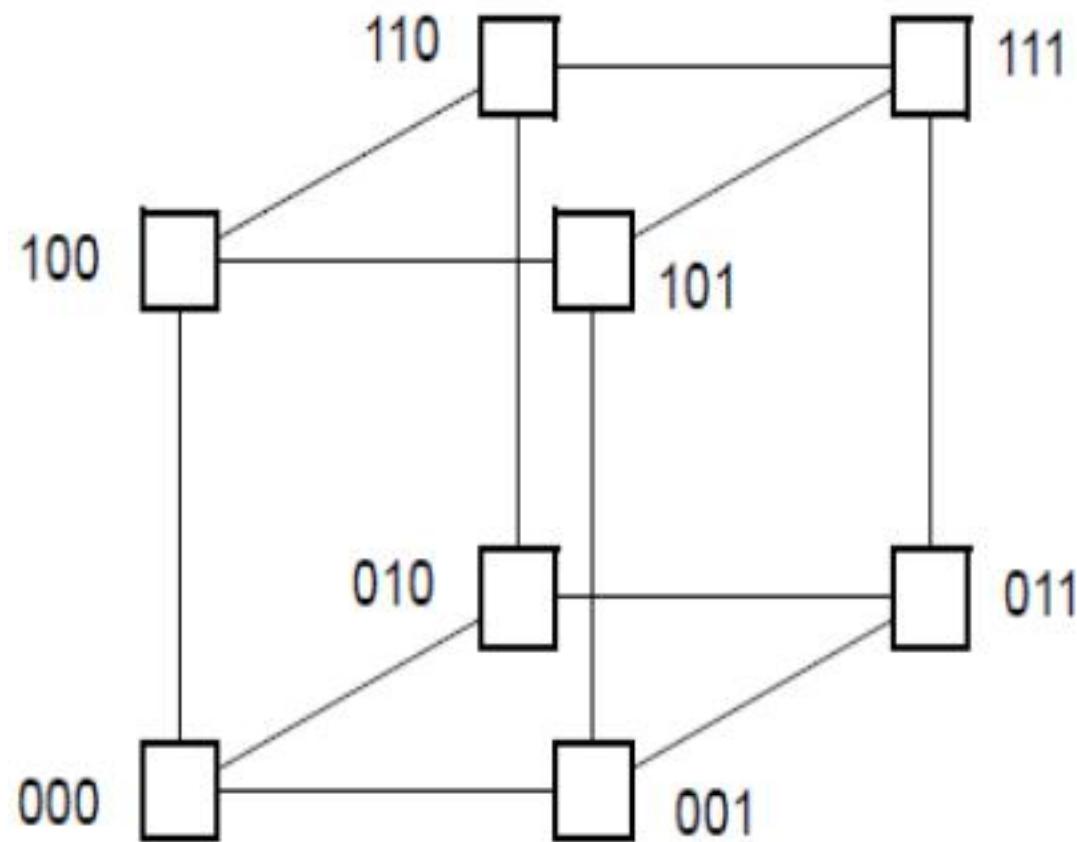
Various:

- Ring
- Tree
- 2-D Array
- 3-D Array
- Hypercube

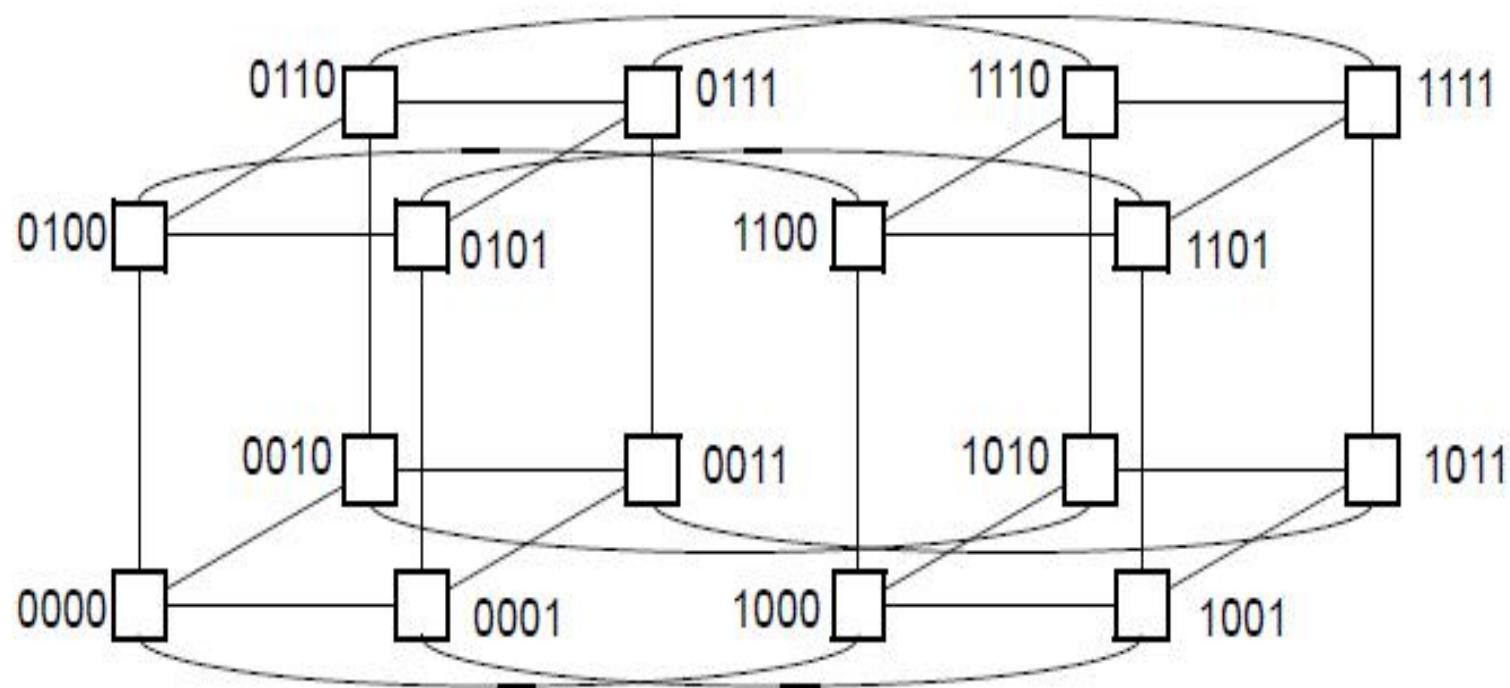
Two-Dimensional Array (mesh)



Three-Dimensional Hypercube



Four-Dimensional Hypercube



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

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.

Limitations of Serial Computing

- 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 specialized solutions as IBM Cells, ClearSpeed, GPGPU from NVidia ...

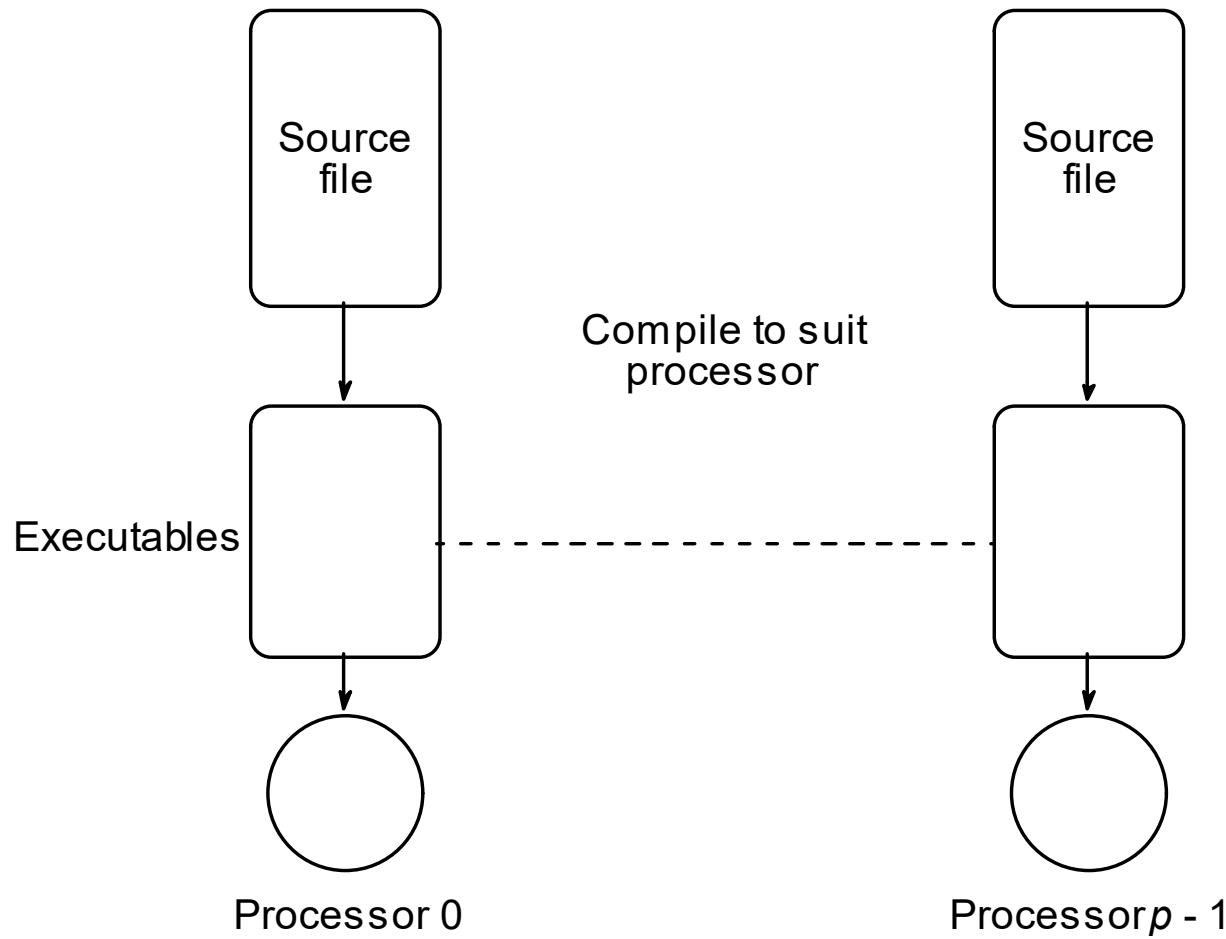
MESSAGE PASSING COMPUTING

Message-Passing Programming using User-level Message Passing Libraries

Two primary mechanisms needed:

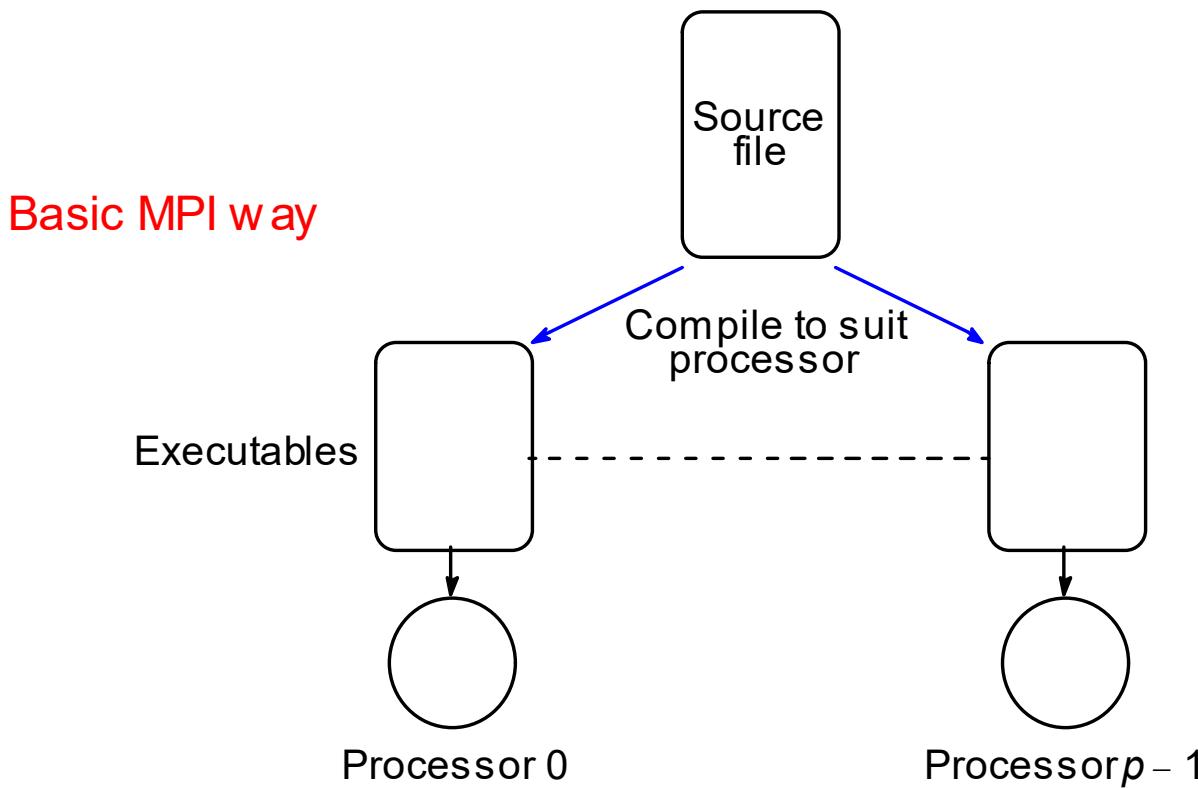
1. A method of creating separate processes for execution on different computers
2. A method of sending and receiving messages

Multiple program, multiple data (MPMD) model



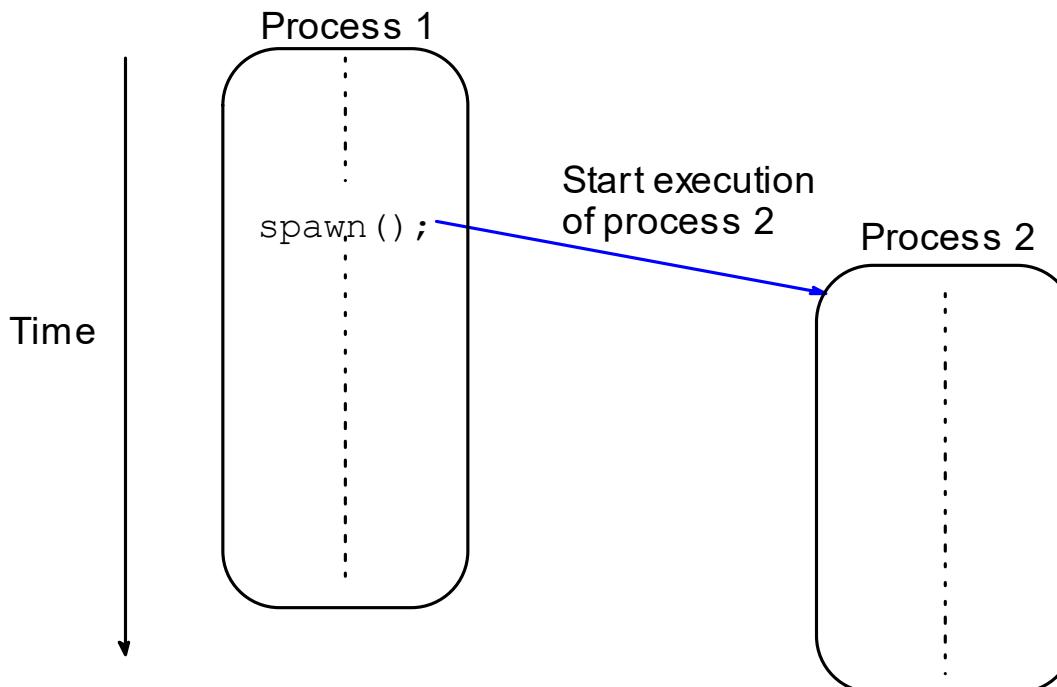
Single Program Multiple Data (SPMD) Model

Different processes merged into one program. Control statements select different parts for each processor to execute. All executables started together - *static process creation*



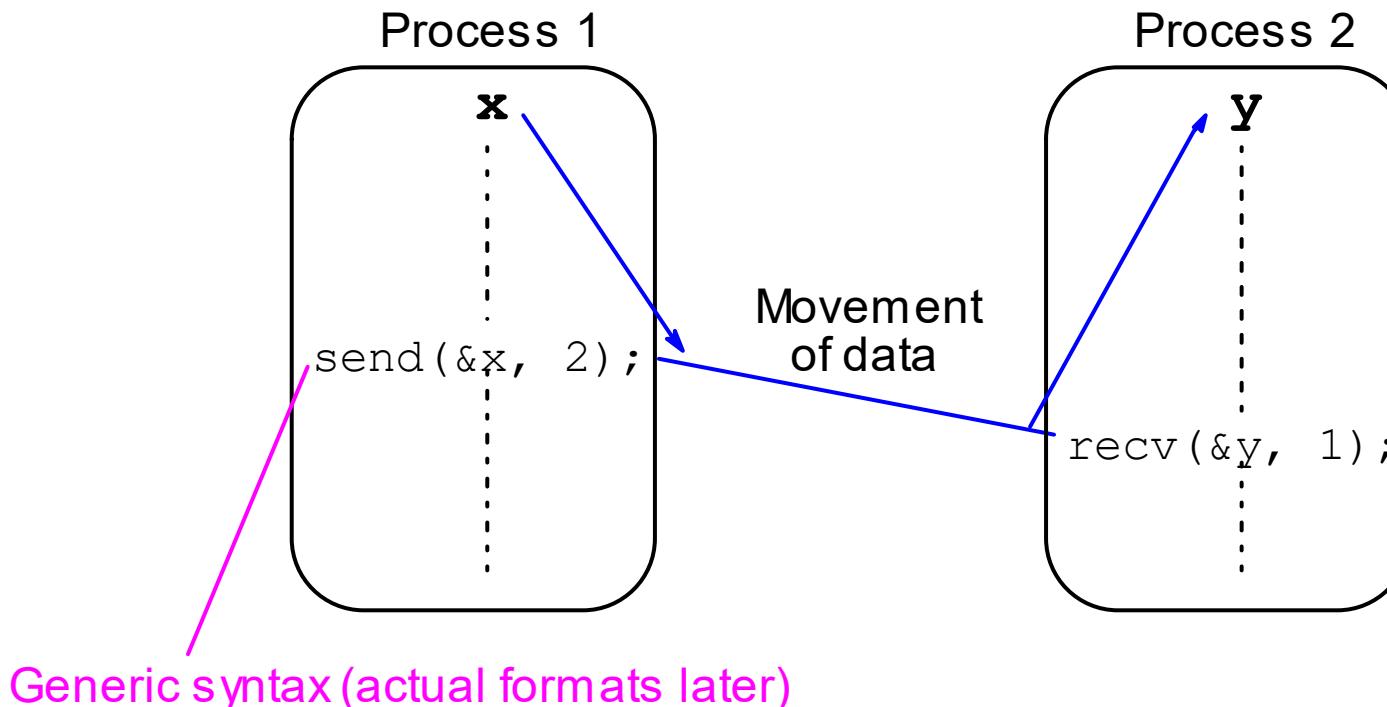
Multiple Program Multiple Data (MPMD) Model

Separate programs for each processor. One processor executes master process. Other processes started from within master process - *dynamic process creation*.



Basic “point-to-point” Send and Receive Routines

Passing a message between processes using send()
and recv() library calls:



Synchronous Message Passing

Routines that actually return when message transfer completed.

Synchronous send routine

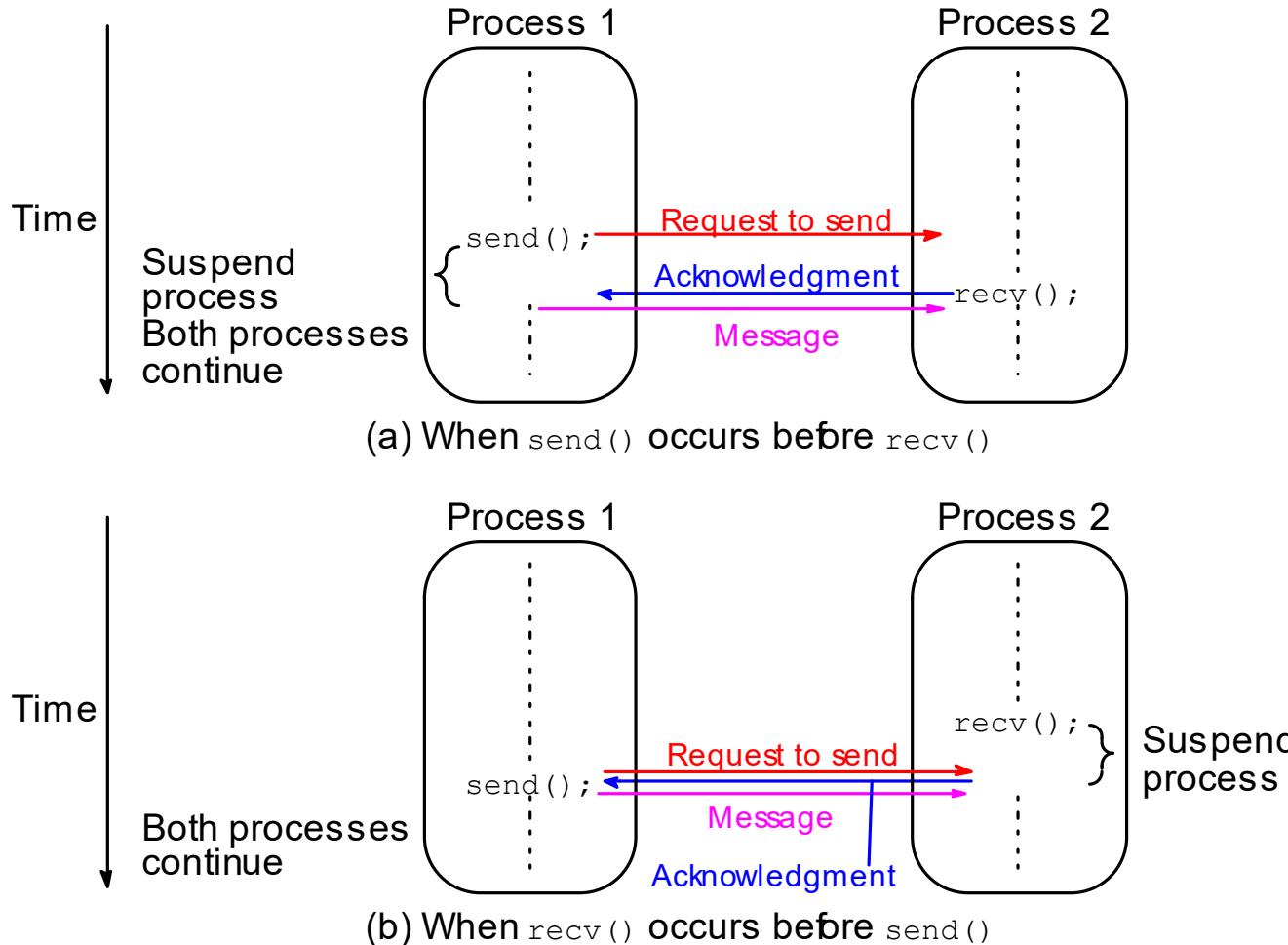
- Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

- Waits until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

Synchronous send() and recv() using 3-way protocol



Asynchronous Message Passing

- Routines that do not wait for actions to complete before returning. Usually require local storage for messages.
- More than one version depending upon the actual semantics for returning.
- In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

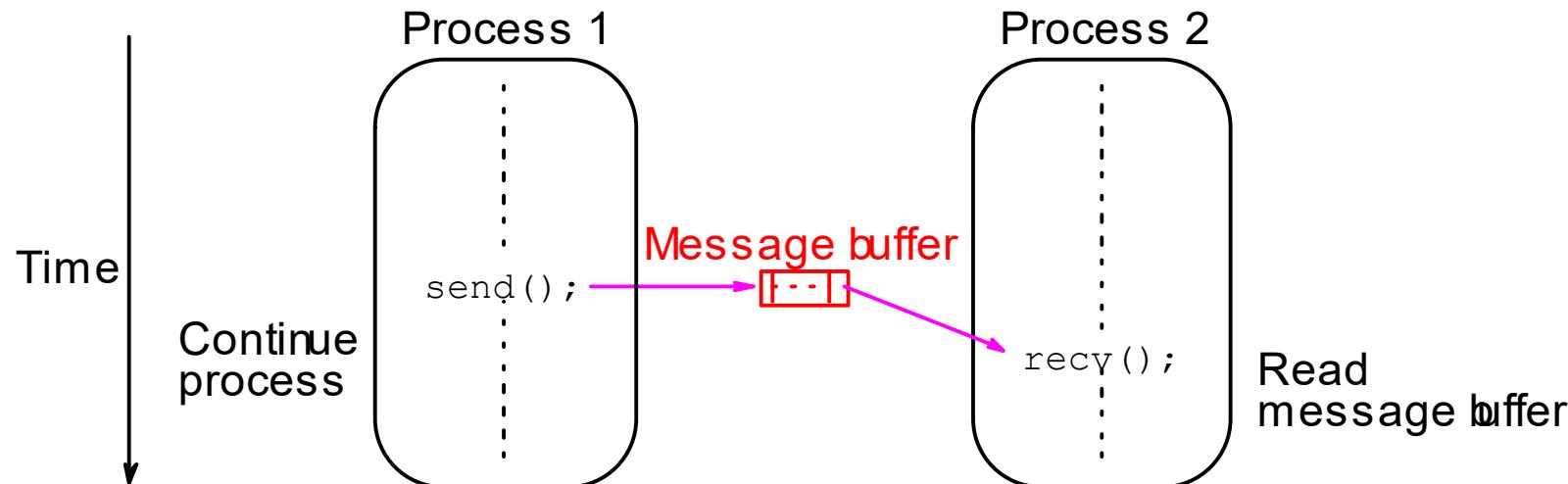
MPI Definitions of Blocking and Non-Blocking

- ***Blocking*** - return after their local actions complete, though the message transfer may not have been completed.
- ***Non-blocking*** - return immediately.
Assumes that data storage used for transfer not modified by subsequent statements prior to being used for transfer, and it is left to the programmer to ensure this.

These terms may have different interpretations in other systems.

How message-passing routines return before message transfer completed

Message buffer needed between source and destination to hold message:



Asynchronous (blocking) routines changing to synchronous routines

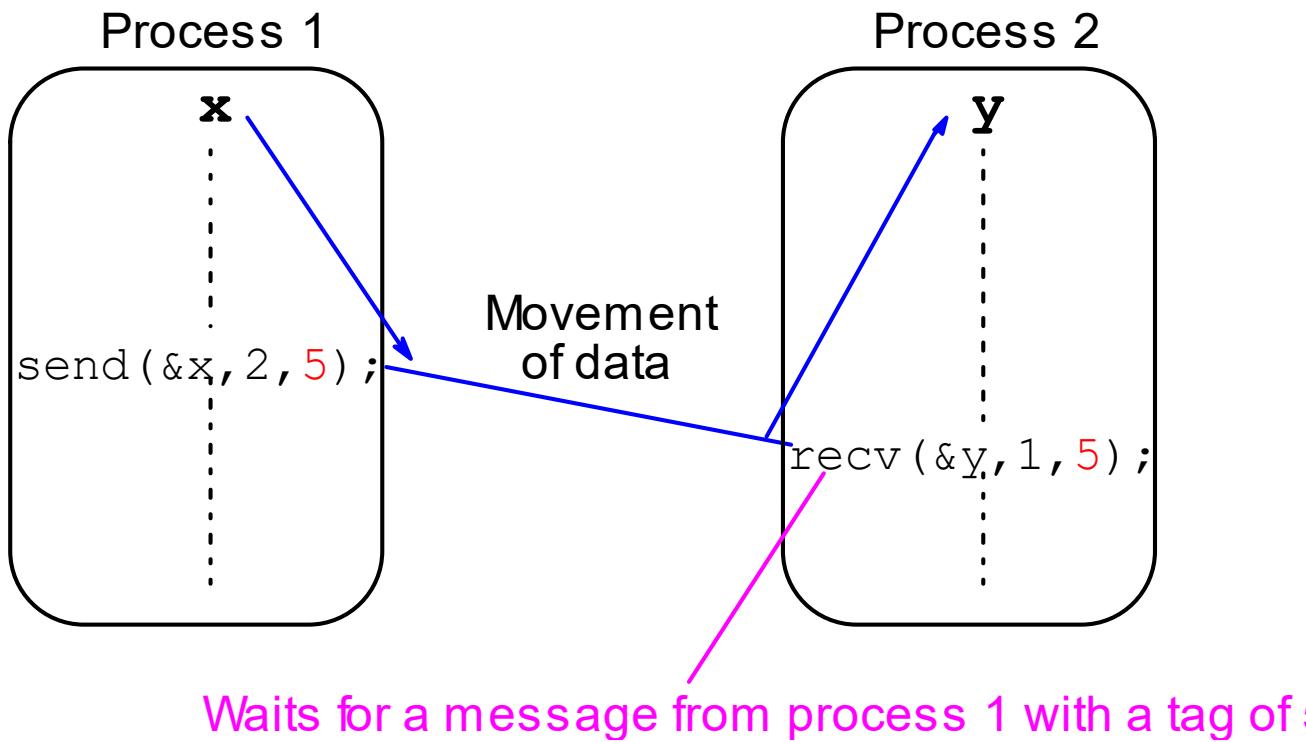
- Once local actions completed and message is safely on its way, sending process can continue with subsequent work.
- Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.
- Then, send routine will wait until storage becomes re-available - i.e then routine behaves as a synchronous routine.

Message Tag

- Used to differentiate between different types of messages being sent.
- Message tag is carried within message.
- If special type matching is not required, a wild card message tag is used, so that the recv() will match with any send().

Message Tag Example

To send a message, x, with message tag 5 from a source process, 1, to a destination process, 2, and assign to y:



“Group” message passing routines

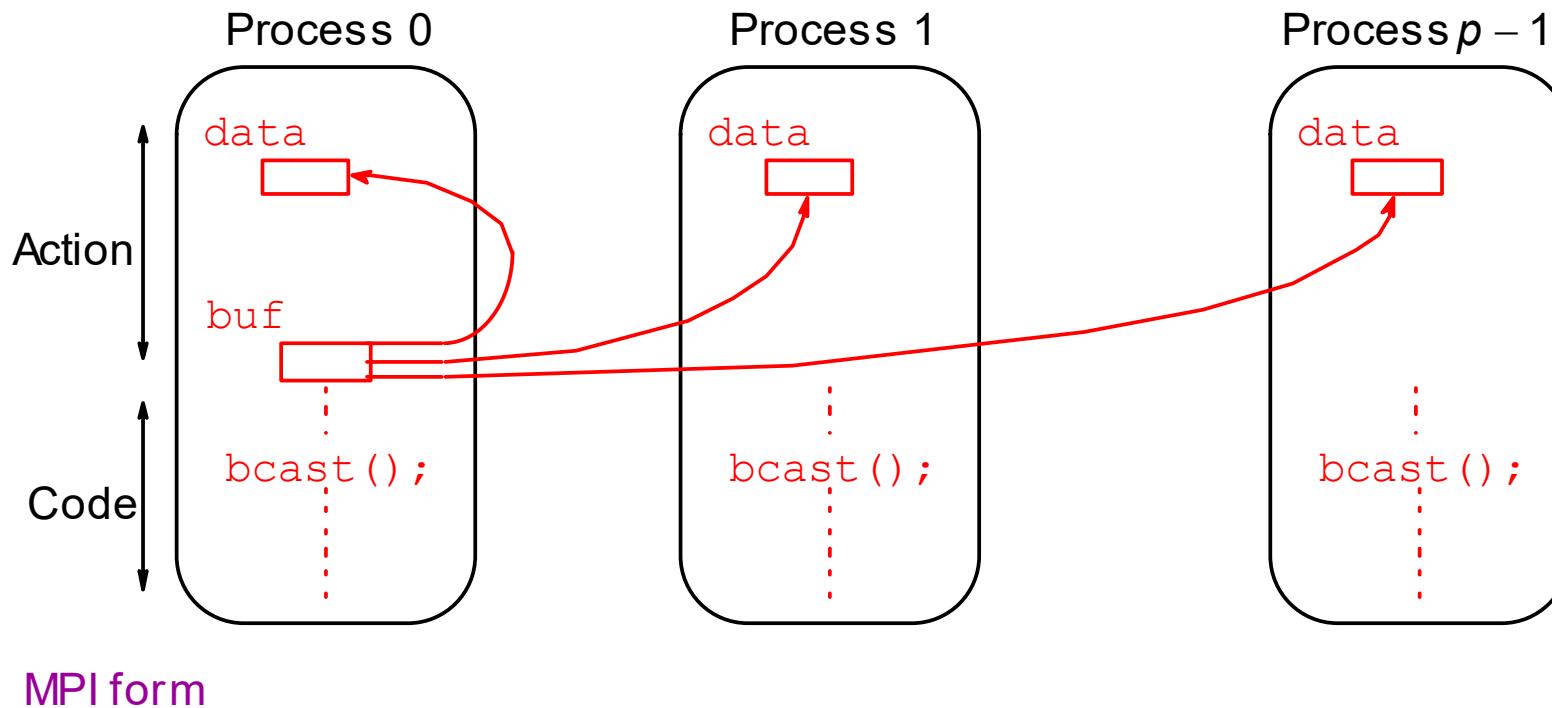
Have routines that send message(s) to a group of processes or receive message(s) from a group of processes

Higher efficiency than separate point-to-point routines although not absolutely necessary.

Broadcast

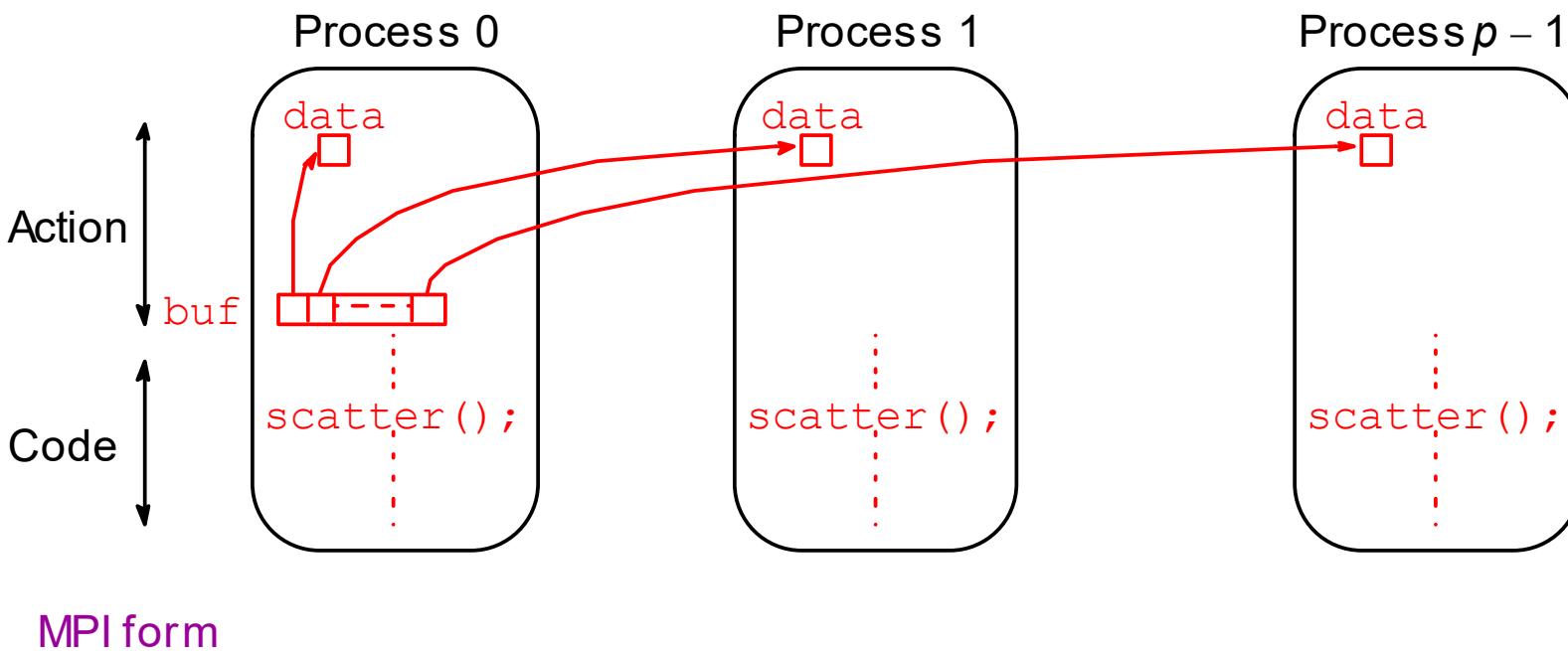
Sending same message to all processes concerned with problem.

Multicast - sending same message to defined group of processes.



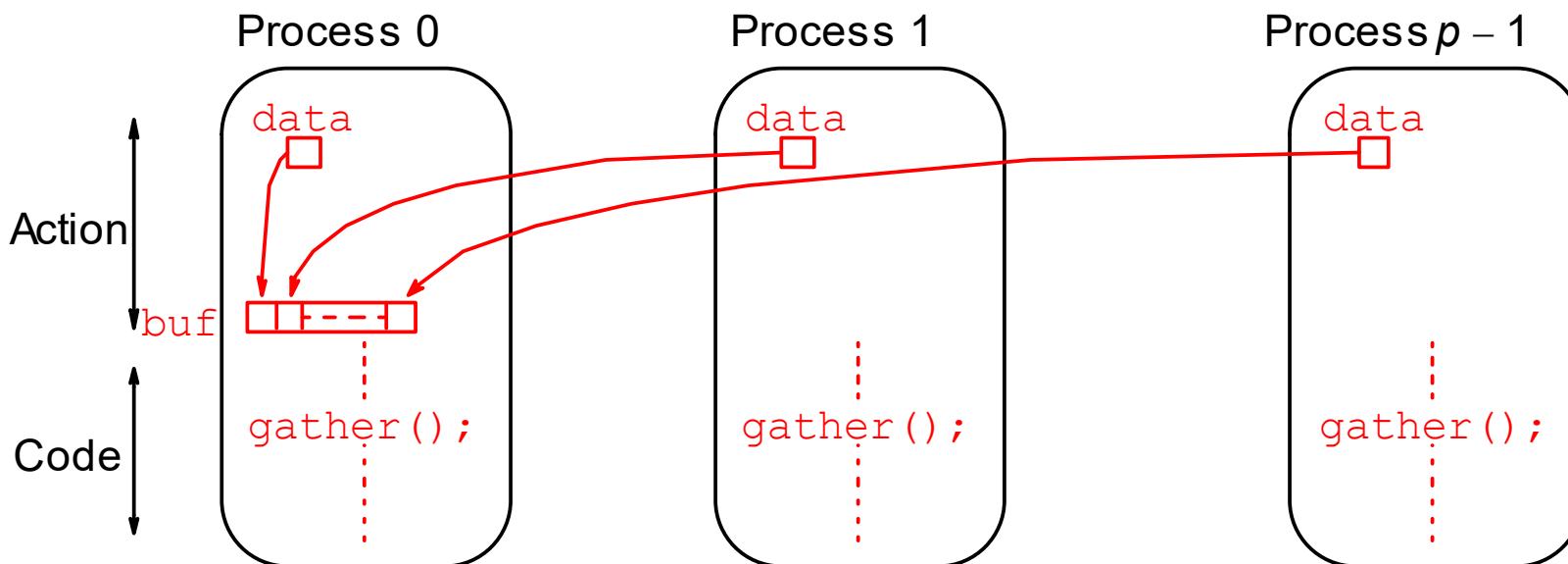
Scatter

Sending each element of an array in root process to a separate process. Contents of i th location of array sent to i th process.



Gather

Having one process collect individual values from set of processes.

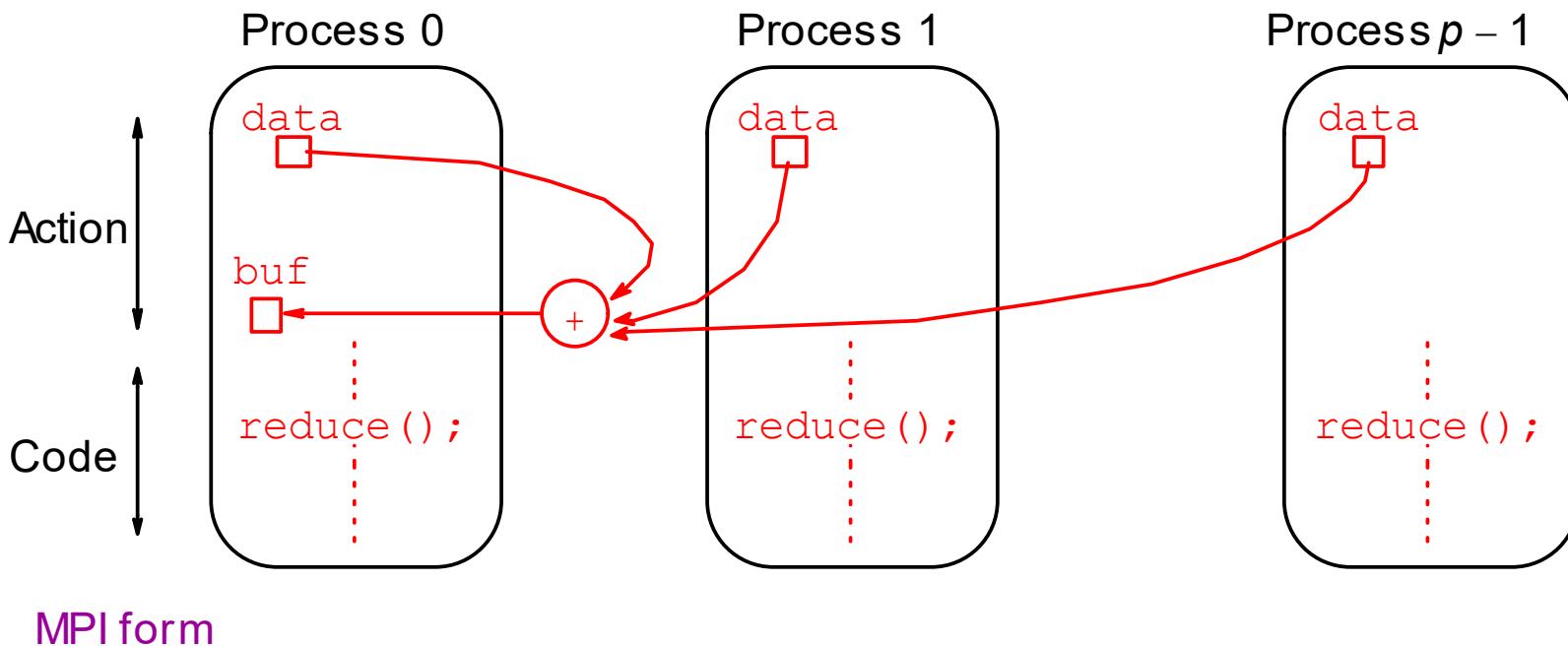


MPI form

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example: Values could be gathered and then added together by root:



PVM (Parallel Virtual Machine)

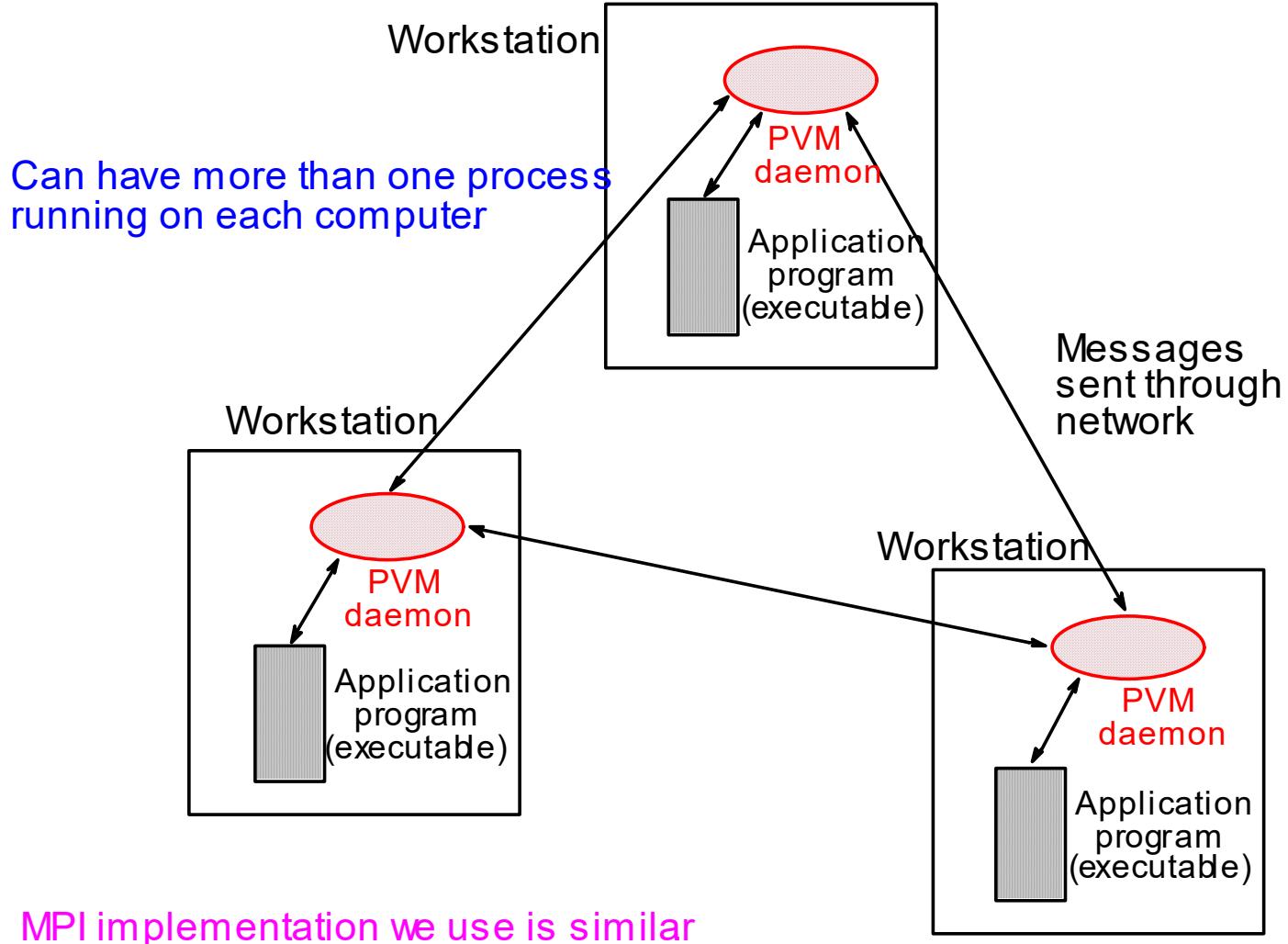
Perhaps first widely adopted attempt at using a workstation cluster as a multicomputer platform, developed by Oak Ridge National Laboratories. Available at no charge.

Programmer decomposes problem into separate programs (usually master and group of identical slave programs).

Programs compiled to execute on specific types of computers.

Set of computers used on a problem first must be defined prior to executing the programs (in a hostfile).

Message routing between computers done by PVM daemon processes installed by PVM on computers that form the virtual machine.



MPI

(Message Passing Interface)

- Message passing library standard developed by group of academics and industrial partners to foster more widespread use and portability.
- Defines routines, not implementation.
- Several free implementations exist.

MPI

Process Creation and Execution

- Purposely not defined - Will depend upon implementation.
- Only static process creation supported in MPI version 1. All processes must be defined prior to execution and started together.
- Originally SPMD model of computation.
- MPMD also possible with static creation - each program to be started together specified.

Communicators

- Defines scope of a communication operation.
- Processes have ranks associated with communicator.
- Initially, all processes enrolled in a “universe” called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $p - 1$, with p processes.
- Other communicators can be established for groups of processes.

Using SPMD Computational Model

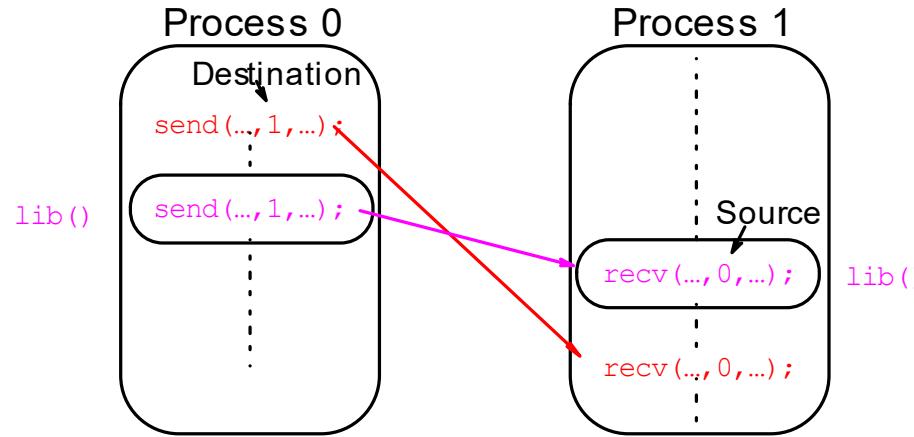
```
main (int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    .
    .
    .
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /*find process rank */
    if (myrank == 0)
        master();
    else
        slave();
    .
    .
    .
    MPI_Finalize();
}
```

where `master()` and `slave()` are to be executed by master process and slave process, respectively.

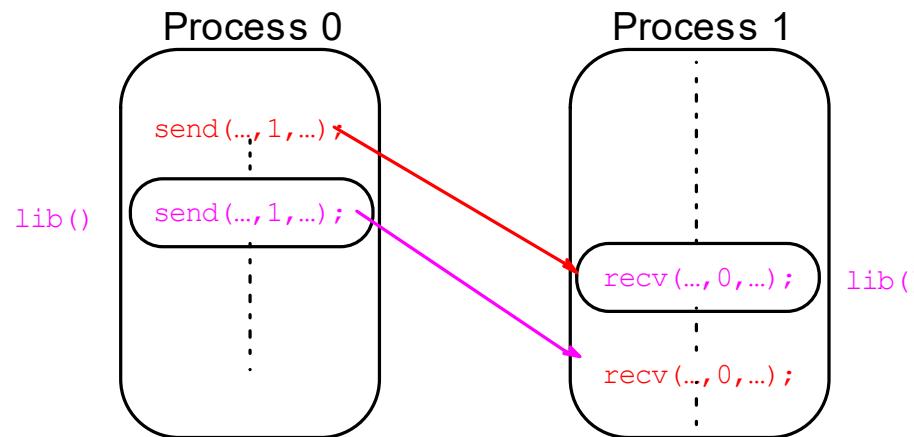
Unsafe message passing

Example

(a) Intended behavior



(b) Possible behavior



MPI Solution

“Communicators”

- Defines a communication domain - a set of processes that are allowed to communicate between themselves.
- Communication domains of libraries can be separated from that of a user program.
- Used in all point-to-point and collective MPI message-passing communications.

Default Communicator

`MPI_COMM_WORLD`

- Exists as first communicator for all processes existing in the application.
- A set of MPI routines exists for forming communicators.
- Processes have a “rank” in a communicator.

MPI Point-to-Point Communication

- Uses send and receive routines with message tags (and communicator).
- Wild card message tags available

MPI Blocking Routines

- Return when “locally complete” - when location used to hold message can be used again or altered without affecting message being sent.
- Blocking send will send message and return - does not mean that message has been received, just that process free to move on without adversely affecting message.

Parameters of blocking send

```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

Address of send buffer | Datatype of each item | Message tag

Number of items to send | Rank of destination process | Communicator

Parameters of blocking receive

```
MPI_Recv(buf, count, datatype, src, tag, comm, status)
```

Address of receive buffer
Maximum number of items to receive
Datatype of each item
Rank of source process
Message tag
Communicator
Status after operation

Example

To send an integer x from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */  
  
if (myrank == 0) {  
    int x;  
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);  
} else if (myrank == 1) {  
    int x;  
    MPI_Recv(&x, 1, MPI_INT,  
             0, msgtag, MPI_COMM_WORLD, status);  
}
```

MPI Nonblocking Routines

- **Nonblocking send** - `MPI_Isend()` - will return “immediately” even before source location is safe to be altered.
- **Nonblocking receive** - `MPI_Irecv()` - will return even if no message to accept.

Nonblocking Routine Formats

`MPI_Isend(buf, count, datatype, dest, tag, comm, request)`

`MPI_Irecv(buf, count, datatype, source, tag, comm, request)`

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait()` waits until operation completed and returns then.

`MPI_Test()` returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.

Determined by accessing `request` parameter.

Example

To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank); /* find rank */
if (myrank == 0) {
    int x;
    MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
    compute();
    MPI_Wait(req1, status);
} else if (myrank == 1) {
    int x;
    MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Four Send Communication Modes

- ***Standard Mode Send*** - Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering provided, send could complete before receive reached.
- ***Buffered Mode*** - Send may start and return before a matching receive. Necessary to specify buffer space via routine `MPI_Buffer_attach()`.
- ***Synchronous Mode*** - Send and receive can start before each other but can only complete together.
- ***Ready Mode*** - Send can only start if matching receive already reached, otherwise error. Use with care.

- Each of the four modes can be applied to both blocking and non blocking send routines.
- Only the standard mode is available for the blocking and non blocking receive routines.
- Any type of send routine can be used with any type of receive routine.

Collective Communication

Involves set of processes, defined by an intra-communicator.
Message tags not present. Principal collective operations:

- **MPI_Bcast()** - Broadcast from root to all other processes
- **MPI_Gather()** - Gather values for group of processes
- **MPI_Scatter()** - Scatters buffer in parts to group of processes
- **MPI_Alltoall()** - Sends data from all processes to all processes
- **MPI_Reduce()** - Combine values on all processes to single value
- **MPI_Reduce_scatter()** - Combine values and scatter results
- **MPI_Scan()** - Compute prefix reductions of data on processes

Example

To gather items from group of processes into process 0,
using dynamically allocated memory in root process:

```
int data[10];                      /*data to be gathered from processes*/
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);    /* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
    buf = (int *)malloc(grp_size*10*sizeof (int)); /*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORL
D};
```

MPI_Gather() gathers from all processes, including root.

Barrier

- As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

Sample MPI program

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>

#define MAXSIZE 1000

void main(int argc, char *argv)
{
    int myid, numprocs;

    int data[MAXSIZE], i, x, low, high, myresult, result;

    char fn[255];

    char *fp;

    MPI_Init(&argc,&argv);

    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);

    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    if (myid == 0) /* Open input file and initialize data */

        strcpy(fn,getenv("HOME"));

        strcat(fn,"/MPI/rand_data.txt");

        if ((fp = fopen(fn,"r")) == NULL) {

            printf("Can't open the input file: %s\n", fn);

            exit(1);

        }

        for(i = 0; i < MAXSIZE; i++) fscanf(fp,"%d", &data[i]);

    }

    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD); /* broadcast data */

    x = n/proc; /* Add my portion of data */

    low = myid * x;

    high = low + x;

    for(i = low; i < high; i++)

        myresult += data[i];

    printf("I got %d from %d\n", myresult, myid); /* Compute global sum */

    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0) printf("The sum is %d.\n", result);

    MPI_Finalize();

}
```

Evaluating Parallel Programs

Sequential execution time, t_s : Estimate by counting computational steps of best sequential algorithm.

Parallel execution time, t_p : In addition to number of computational steps, t_{comp} , need to estimate communication overhead, t_{comm} :

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

Computational Time

Count number of computational steps. When more than one process executed simultaneously, count computational steps of most complex process. Generally, function of n and p , i.e.

$$t_{\text{comp}} = f(n, p)$$

Often break down computation time into parts. Then

$$t_{\text{comp}} = t_{\text{comp1}} + t_{\text{comp2}} + t_{\text{comp3}} + \dots$$

Analysis usually done assuming that all processors are same and operating at same speed.

Communication Time

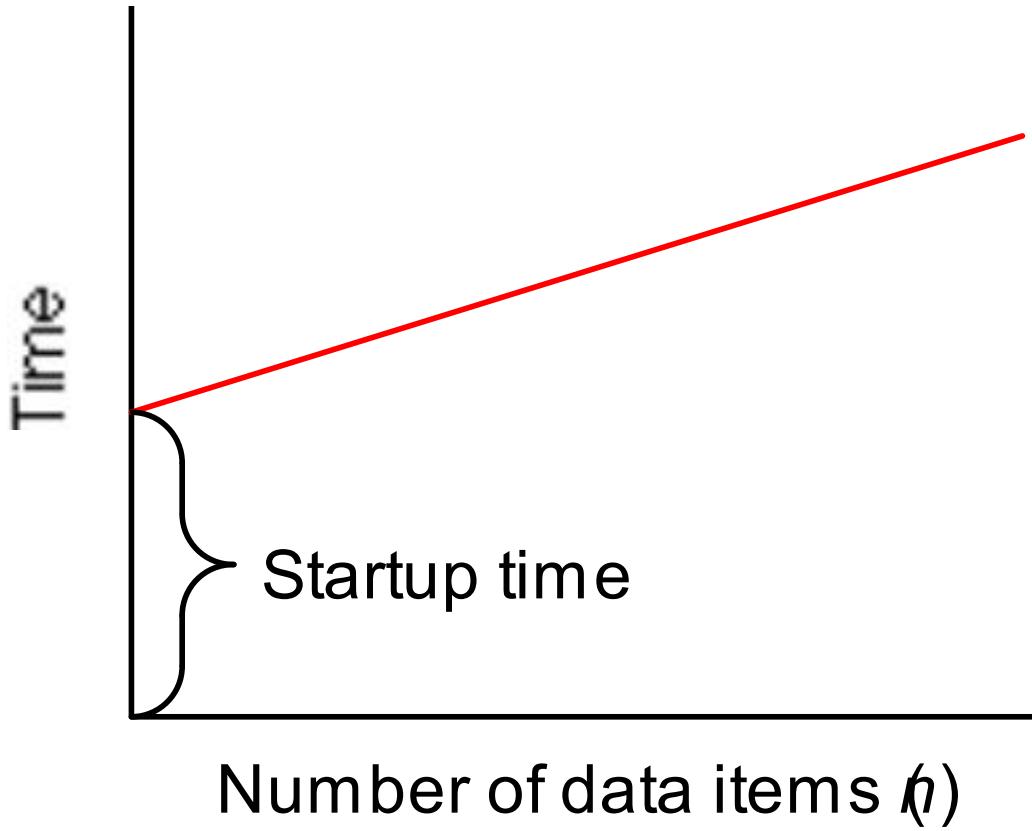
Many factors, including network structure and network contention. As a first approximation, use

$$t_{\text{comm}} = t_{\text{startup}} + nt_{\text{data}}$$

t_{startup} is startup time, essentially time to send a message with no data. Assumed to be constant.

t_{data} is transmission time to send one data word, also assumed constant, and there are n data words.

Idealized Communication Time



Final communication time, t_{comm} , the summation of communication times of all sequential messages from a process, i.e.

$$t_{\text{comm}} = t_{\text{comm1}} + t_{\text{comm2}} + t_{\text{comm3}} + \dots$$

Typically, communication patterns of all processes same and assumed to take place together so that only one process need be considered.

Both startup and data transmission times, t_{startup} and t_{data} , measured in units of one computational step, so that can add t_{comp} and t_{comm} together to obtain parallel execution time, t_p .

Benchmark Factors

With t_s , t_{comp} , and t_{comm} , can establish speedup factor and computation/communication ratio for a particular algorithm/implementation:

$$\text{Speedup factor} = \frac{t_s}{t_p} = \frac{t_s}{t_{\text{comp}} + t_{\text{comm}}}$$

$$\text{Computation/communication ratio} = \frac{t_{\text{comp}}}{t_{\text{comm}}}$$

Both functions of number of processors, p , and number of data elements, n .

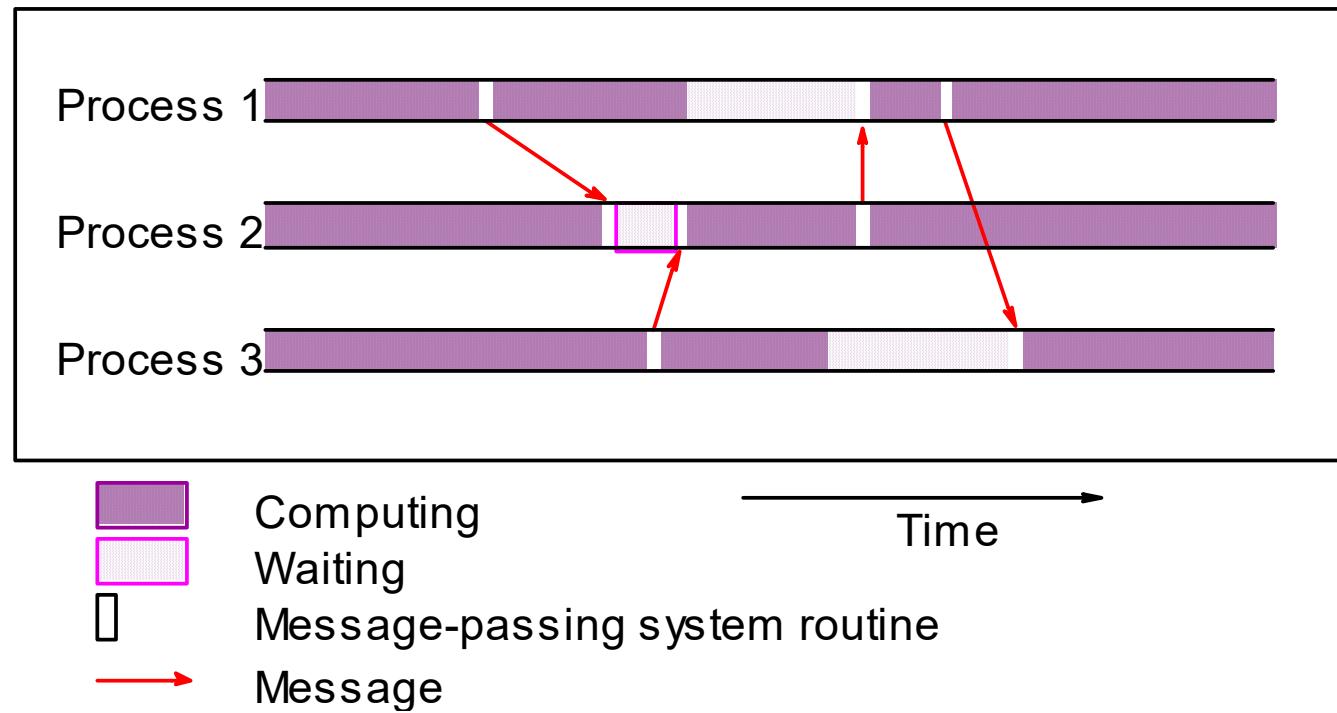
Will give indication of scalability of parallel solution with increasing number of processors and problem size.

Computation/communication ratio will highlight effect of communication with increasing problem size and system size.

Debugging and Evaluating Parallel Programs Empirically

Visualization Tools

Programs can be watched as they are executed in a space-time diagram (or process-time diagram):



Implementations of visualization tools are available for MPI.

An example is the Upshot program visualization system.

Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

```
L1: time(&t1);          /* start timer */  
.  
. .  
.  
L2: time(&t2);          /* stop timer */  
.  
. .  
elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */  
printf("Elapsed time = %5.2f seconds", elapsed_time);
```

MPI provides the routine **`MPI_Wtime()`** for returning time (in seconds).

Parallel Programming Home Page

http://www.cs.uncc.edu/par_prog

Gives step-by-step instructions for compiling and executing programs, and other information.

Compiling/Executing MPI Programs

Preliminaries

- Set up paths
- Create required directory structure
- Create a file (hostfile) listing machines to be used (required)

Details described on home page.

Hostfile

Before starting MPI for the first time, need to create a hostfile

Sample hostfile

```
ws404
#is-sm1 //Currently not executing, commented
pvm1 //Active processors, UNCC sun cluster called pvm1 - pvm8
pvm2
pvm3
pvm4
pvm5
pvm6
pvm7
pvm8
```

Compiling/executing (SPMD) MPI program

For LAM MPI version 6.5.2. At a command line:

To start MPI:

First time: `lamboot -v hostfile`

Subsequently: `lamboot`

To compile MPI programs:

`mpicc -o file file.c`

or `mpiCC -o file file.cpp`

To execute MPI program:

`mpirun -v -np no_processors file`

To remove processes for reboot

`lamclean -v`

Terminate LAM

`lamhalt`

If fails

`wipe -v lamhost`

Compiling/Executing Multiple MPI Programs

Create a file specifying programs:

Example

1 master and 2 slaves, “appfile” contains

```
n0 master  
n0-1 slave
```

To execute:

```
mpirun -v appfile
```

Sample output

```
3292 master running on n0 (o)  
3296 slave running on n0 (o)  
412 slave running on n1
```

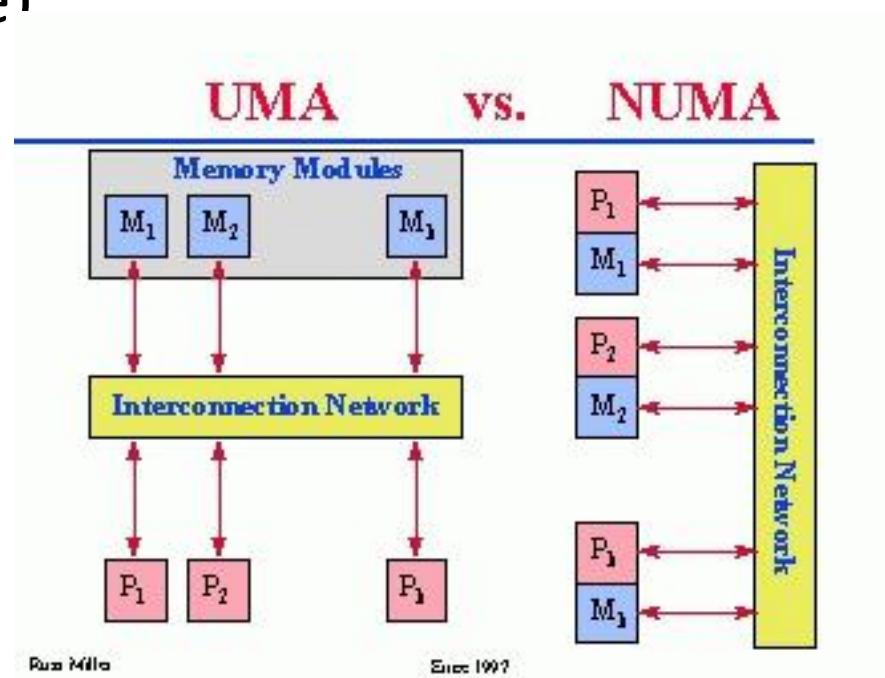
CHAPTER 2

HARDWARE



Memory and communication

- Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space)



Memory and communication

- Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.



Memory and communication

- Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.



Classes of parallel computers

- Parallel computers can be roughly classified according to the level at which the hardware supports parallelism. This classification is broadly analogous to the distance between basic computing nodes.



Classes of parallel computers

Multicore computing



- A multicore processor is a processor that includes multiple execution units ("cores") on the same chip.
- A multicore processor can issue multiple instructions per cycle from multiple instruction streams. IBM's Cell microprocessor, designed for use in the Sony PlayStation 3, is another prominent multicore processor.



- Each core in a multicore processor can potentially be superscalar as well—that is, on every cycle, each core can issue multiple instructions from one instruction stream.
- Simultaneous multithreading (of which Intel's HyperThreading is the best known) was an early form of pseudo-multicoreism.
- A processor capable of simultaneous multithreading has only one execution unit ("core"), but when that execution unit is idling (such as during a cache miss), it uses that execution unit to process a second thread



Classes of parallel computers

Symmetric multiprocessing



- A symmetric multiprocessor (SMP) is a computer system with multiple identical processors that share memory and connect via a bus.
- Bus contention prevents bus architectures from scaling. As a result, SMPs generally do not comprise more than 32 processors.



Classes of parallel computers

Distributed computing



- A distributed computer (also known as a distributed memory multiprocessor) is a distributed memory computer system in which the processing elements are connected by a network. Distributed computers are highly scalable.



Classes of parallel computers

Cluster computing





A Beowulf cluster



- A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer.
- Clusters are composed of multiple standalone machines connected by a network.
- The most common type of cluster is the Beowulf cluster, which is a cluster implemented on multiple identical commercial off-the-shelf computers connected with a TCP/IP Ethernet local area network.



Classes of parallel computers

Massive parallel processing



- A massively parallel processor (MPP) is a single computer with many networked processors.
- MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks (whereas clusters use commodity hardware for networking).
- In a MPP, "each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via a high-speed interconnect." [16] A cabinet from Blue Gene/L, ranked as the fourth fastest supercomputer in the world according to the 11/2008 TOP500 rankings. Blue Gene/L is a massively parallel processor. Blue Gene/L, the fifth fastest supercomputer in the world according to the June 2009 TOP500 ranking, is a MPP.



Classes of parallel computers

Grid computing





A cabinet from **Blue Gene/L**, ranked as the fourth fastest supercomputer in the world according to the 11/2008 **TOP500** rankings. **Blue Gene/L** is a massively parallel processor.



- Distributed computing is the most distributed form of parallel computing. It makes use of computers communicating over the Internet to work on a given problem.
- Because of the low bandwidth and extremely high latency available on the Internet, distributed computing typically deals only with embarrassingly parallel problems.
- Many distributed computing applications have been created, of which SETI@home and Folding@home are the best-known examples.



- Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface. The most common distributed computing middleware is the Berkeley Open Infrastructure for Network Computing (BOINC). Often, distributed computing software makes use of "spare cycles", performing computations at times when a computer is idling.



Classes of parallel computers

Specialized parallel computers



- Within parallel computing, there are specialized parallel devices that remain niche areas of interest. While not domain-specific, they tend to be applicable to only a few classes of parallel problems.



General-purpose computing on graphics processing units (GPGPU)



- General-purpose computing on graphics processing units (GPGPU) is a fairly recent trend in computer engineering research.
- GPUs are co-processors that have been heavily optimized for computer graphics processing.
- Computer graphics processing is a field dominated by data parallel operations—particularly linear algebra matrix operations.



- In the early days, GPGPU programs used the normal graphics APIs for executing programs. However, several new programming languages and platforms have been built to do general purpose computation on GPUs with both Nvidia and AMD releasing programming environments with CUDA and Stream SDK respectively. Other GPU programming languages include BrookGPU, PeakStream, and RapidMind.



- Nvidia has also released specific products for computation in their Tesla series. The technology consortium Khronos Group has released the OpenCL specification, which is a framework for writing programs that execute across platforms consisting of CPUs and GPUs. AMD, Apple, Intel, Nvidia and others are supporting OpenCL.

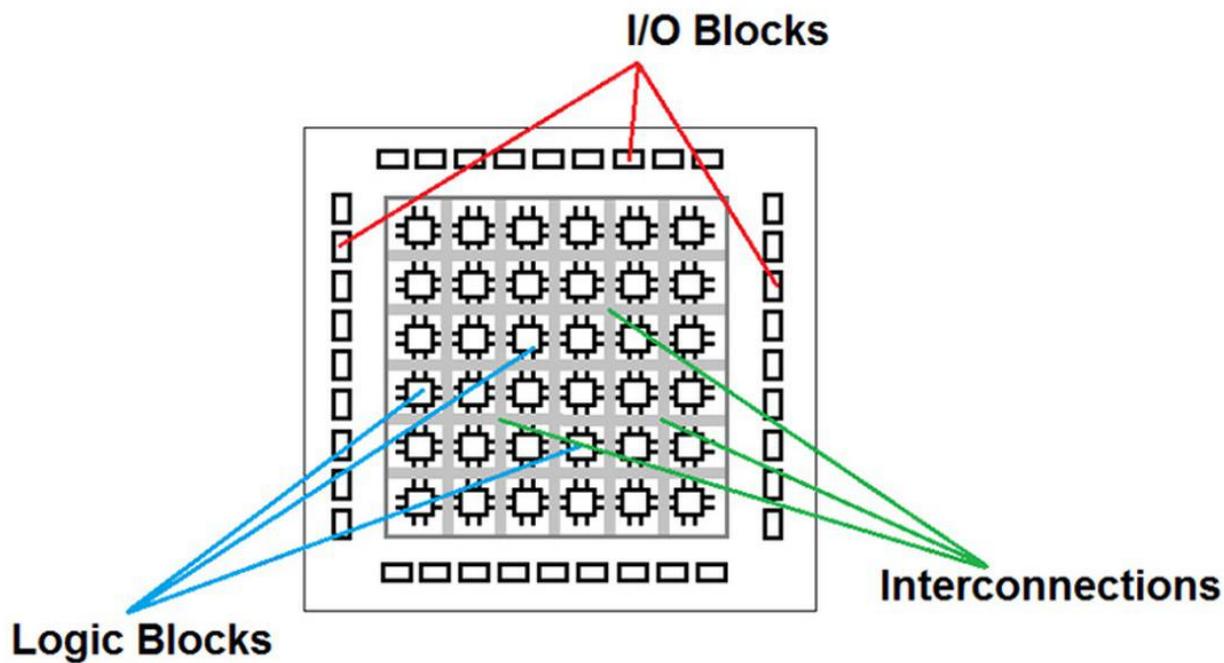


Classes of parallel computers

Reconfigurable Computing using field-programmable gate arrays



- FPGA



Application-specific integrated circuits



- Several application-specific integrated circuit (ASIC) approaches have been devised for dealing with parallel applications



- Because an ASIC is (by definition) specific to a given application, it can be fully optimized for that application. As a result, for a given application, an ASIC tends to outperform a general-purpose computer. However, ASICs are created by X-ray lithography. This process requires a mask, which can be extremely expensive. A single mask can cost over a million US dollars. (The smaller the transistors required for the chip, the more expensive the mask will be.) Meanwhile, performance increases in general-purpose computing over time (as described by Moore's Law) tend to wipe out these gains in only one or two chip generations. High initial cost, and the tendency to be overtaken by Moore's-law-driven general-purpose computing, has rendered ASICs unfeasible for most parallel computing applications. However, some have been built. One example is the peta-flop RIKEN MDGRAPE-3 machine which uses custom ASICs for molecular dynamics simulation.



Vector processors



- A vector processor is a CPU or computer system that can execute the same instruction on large sets of data.
- Cray computers became famous for their vector-processing computers in the 1970s and 1980s. However, vector processors—both as CPUs and as full computer systems—have generally disappeared. Modern processor instruction sets do include some vector processing instructions, such as with AltiVec and Streaming SIMD Extensions (SSE).



Software

Parallel programming languages



Parallel programming languages

- Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons) have been created for programming parallel computers.



Parallel programming languages

- These can generally be divided into classes based on the assumptions they make about the underlying memory architecture
 1. shared memory
 2. distributed memory
 3. shared distributed memory



Parallel programming languages

- Shared memory programming languages communicate by manipulating shared memory variables.
- Distributed memory uses message passing



Software

Automatic parallelization



Automatic parallelization

- Automatic parallelization of a sequential program by a compiler is the holy grail of parallel computing.
- Despite decades of work by compiler researchers, automatic parallelization has had only limited success. Mainstream parallel programming languages remain either explicitly parallel or (at best) partially implicit, in which a programmer gives the compiler directives for parallelization.
- A few fully implicit parallel programming languages exist—SISAL, Parallel Haskell, System C (for FPGAs), Mitrion-C, VHDL, and Verilog.



Software

Algorithmic methods



Algorithmic methods

- As parallel computers become larger and faster, it becomes feasible to solve problems that previously took too long to run. Parallel computing is used in a wide range of fields, from bioinformatics (protein folding and sequence analysis) to economics (mathematical finance). Common types of problems found in parallel computing applications are



- Dense linear algebra
- Sparse linear algebra
- Spectral methods (such as Cooley–Tukey fast Fourier transform)
- n -body problems (such as Barnes–Hut simulation)
- Structured grid problems (such as Lattice Boltzmann methods)
- Unstructured grid problems (such as found in finite element analysis)
- Monte Carlo simulation
- Combinational logic (such as brute-force cryptographic techniques)
- Graph traversal (such as sorting algorithms)
- Dynamic programming
- Branch and bound methods
- Graphical models (such as detecting hidden Markov models and constructing Bayesian networks)
- Finite-state machine simulation



Software

Fault-tolerance



Fault-tolerance

- Fault-tolerant computer system Parallel computing can also be applied to the design of fault-tolerant computer systems, particularly via lockstep systems performing the same operation in parallel.
- This provides redundancy in case one component should fail, and also allows automatic error detection and error correction if the results differ.



Thank you



Principles of Parallel Algorithm Design

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.



Chapter Overview: Algorithms and Concurrency

- Introduction to Parallel Algorithms
 - Tasks and Decomposition
 - Processes and Mapping
 - Processes Versus Processors
- Decomposition Techniques
 - Recursive Decomposition
 - Recursive Decomposition
 - Exploratory Decomposition
 - Hybrid Decomposition
- Characteristics of Tasks and Interactions
 - Task Generation, Granularity, and Context
 - Characteristics of Task Interactions.



Chapter Overview: Concurrency and Mapping

- Mapping Techniques for Load Balancing
 - Static and Dynamic Mapping
- Methods for Minimizing Interaction Overheads
 - Maximizing Data Locality
 - Minimizing Contention and Hot-Spots
 - Overlapping Communication and Computations
 - Replication vs. Communication
 - Group Communications vs. Point-to-Point Communication
- Parallel Algorithm Design Models
 - Data-Parallel, Work-Pool, Task Graph, Master-Slave, Pipeline, and Hybrid Models



specifying a nontrivial parallel algorithm

- Identifying portions of the work that can be performed **concurrently**.
- **Mapping** the concurrent pieces of work onto multiple processes running in parallel.
- **Distributing** the input, output, and intermediate data associated with the program.
- **Managing** accesses to data shared by multiple processors.
- **Synchronizing** the processors at various stages of the parallel program execution

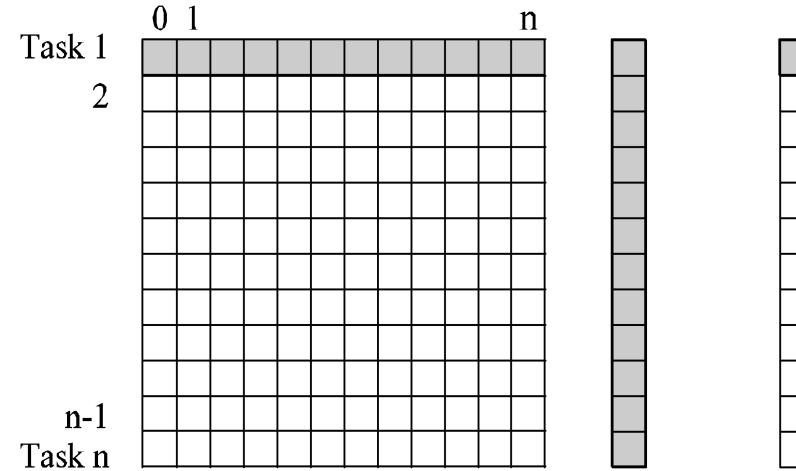


Preliminaries: Decomposition, Tasks, and Dependency Graphs

- The first step in developing a parallel algorithm is to decompose the problem into tasks that can be executed concurrently
- A given problem may be decomposed into tasks in many different ways.
- Tasks may be of same, different, or even interminate sizes.
- A decomposition can be illustrated in the form of a directed graph with nodes corresponding to tasks and edges indicating that the result of one task is required for processing the next. Such a graph is called a *task dependency graph*.



Example: Multiplying a Dense Matrix with a Vector



Computation of each element of output vector y is independent of other elements. Based on this, a dense matrix-vector product can be decomposed into n tasks. The figure highlights the portion of the matrix and vector accessed by Task 1.

Observations: While tasks share data (namely, the vector b), they do not have any control dependencies - i.e., no task needs to wait for the (partial) completion of any other. All tasks are of the same size in terms of number of operations. *Is this the maximum number of tasks we could decompose this problem into?*



Example: Database Query Processing

Consider the execution of the query:

MODEL = "CIVIC" AND YEAR = 2001 AND
(COLOR = "GREEN" OR COLOR = "WHITE")

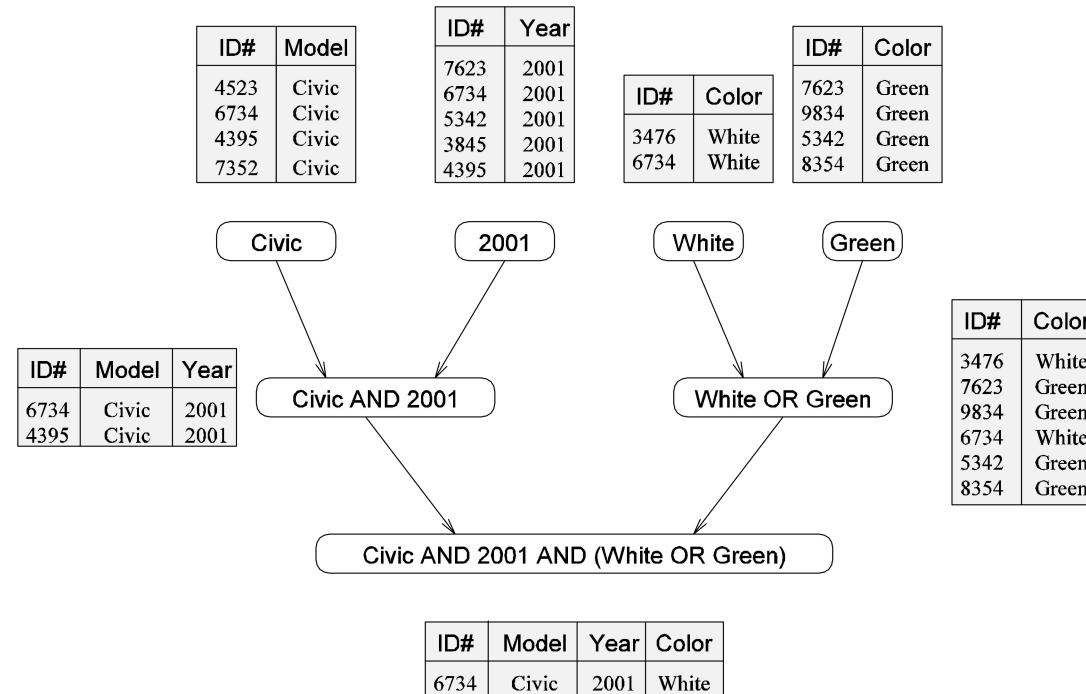
on the following database:

ID#	Model	Year	Color	Dealer	Price
4523	Civic	2002	Blue	MN	\$18,000
3476	Corolla	1999	White	IL	\$15,000
7623	Camry	2001	Green	NY	\$21,000
9834	Prius	2001	Green	CA	\$18,000
6734	Civic	2001	White	OR	\$17,000
5342	Altima	2001	Green	FL	\$19,000
3845	Maxima	2001	Blue	NY	\$22,000
8354	Accord	2000	Green	VT	\$18,000
4395	Civic	2001	Red	CA	\$17,000
7352	Civic	2002	Red	WA	\$18,000



Example: Database Query Processing

The execution of the query can be divided into subtasks in various ways. Each task can be thought of as generating an intermediate table of entries that satisfy a particular clause.

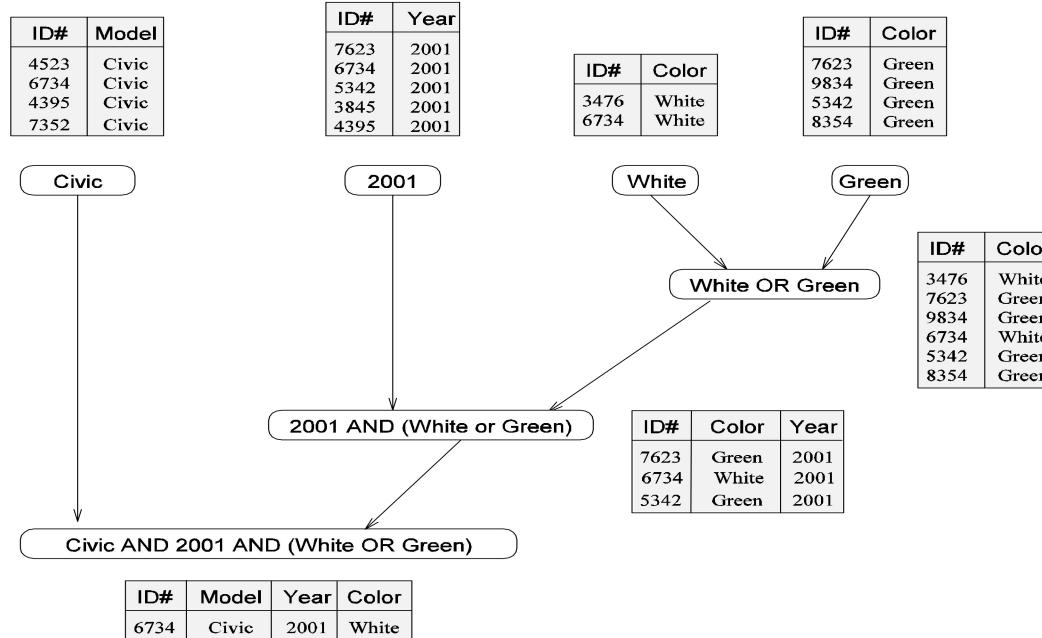


Decomposing the given query into a number of tasks.
Edges in this graph denote that the output of one task
is needed to accomplish the next.



Example: Database Query Processing

Note that the same problem can be decomposed into subtasks in other ways as well.



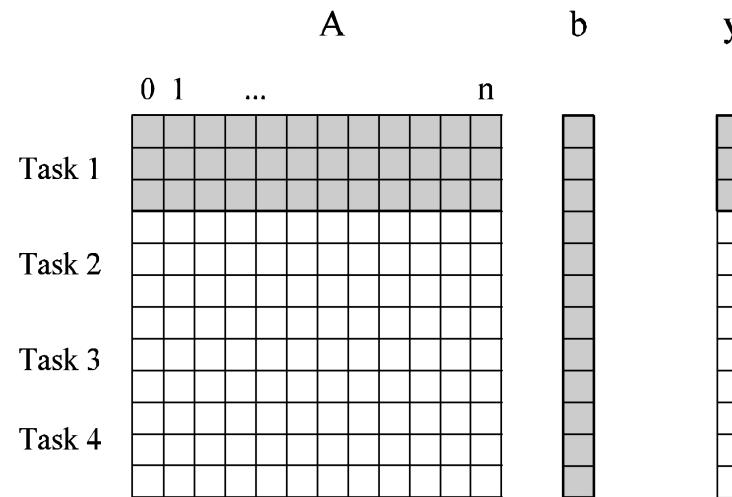
An alternate decomposition of the given problem into subtasks, along with their data dependencies.

Different task decompositions may lead to significant differences with respect to their eventual parallel performance.



Granularity of Task Decompositions

- The number of tasks into which a problem is decomposed determines its granularity.
- Decomposition into a large number of tasks results in fine-grained decomposition and that into a small number of tasks results in a coarse grained decomposition.



A coarse grained counterpart to the dense matrix-vector product example. Each task in this example corresponds to the computation of three elements of the result vector.



Degree of Concurrency

- The number of tasks that can be executed in parallel is the *degree of concurrency* of a decomposition.
- Since the number of tasks that can be executed in parallel may change over program execution, the *maximum degree of concurrency* is the maximum number of such tasks at any point during execution. *What is the maximum degree of concurrency of the database query examples?*
- The *average degree of concurrency* is the average number of tasks that can be processed in parallel over the execution of the program. *Assuming that each task in the database example takes identical processing time, what is the average degree of concurrency in each decomposition?*
- The degree of concurrency increases as the decomposition becomes finer in granularity and vice versa.



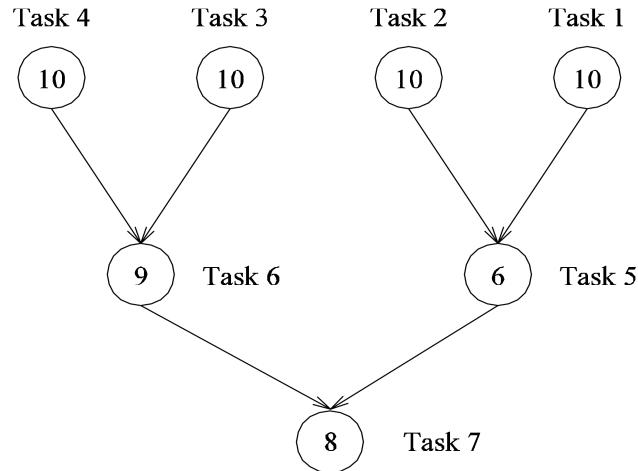
Critical Path Length

- A directed path in the task dependency graph represents a sequence of tasks that must be processed one after the other.
- The longest such path determines the shortest time in which the program can be executed in parallel.
- The length of the longest path in a task dependency graph is called the critical path length.

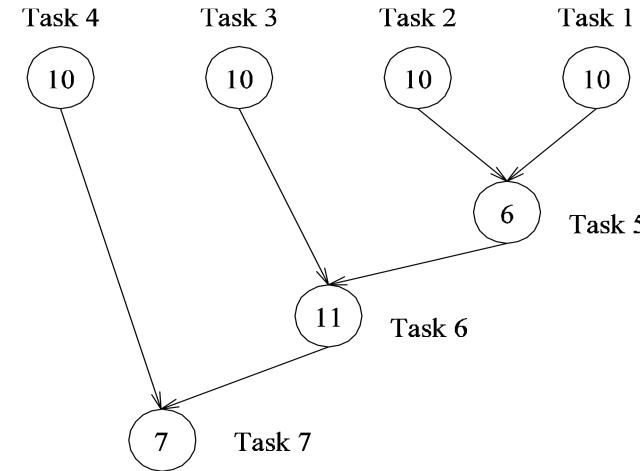


Critical Path Length

Consider the task dependency graphs of the two database query decompositions:



(a)



(b)

What are the critical path lengths for the two task dependency graphs? If each task takes 10 time units, what is the shortest parallel execution time for each decomposition? How many processors are needed in each case to achieve this minimum parallel execution time? What is the maximum degree of concurrency?



Limits on Parallel Performance

- It would appear that the parallel time can be made arbitrarily small by making the decomposition finer in granularity.
- There is an inherent bound on how fine the granularity of a computation can be. *For example, in the case of multiplying a dense matrix with a vector, there can be no more than (n^2) concurrent tasks.*
- Concurrent tasks may also have to exchange data with other tasks. This results in communication overhead. The tradeoff between the granularity of a decomposition and associated overheads often determines performance bounds.



Task Interaction Graphs

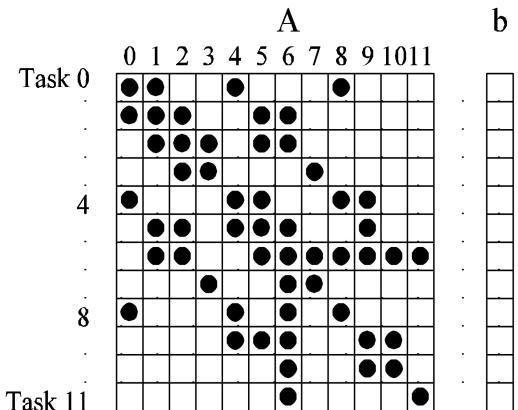
- Subtasks generally exchange data with others in a decomposition. For example, even in the trivial decomposition of the dense matrix-vector product, if the vector is not replicated across all tasks, they will have to communicate elements of the vector.
- The graph of tasks (nodes) and their interactions/data exchange (edges) is referred to as a *task interaction graph*.
- Note that *task interaction graphs* represent data dependencies, whereas *task dependency graphs* represent control dependencies.



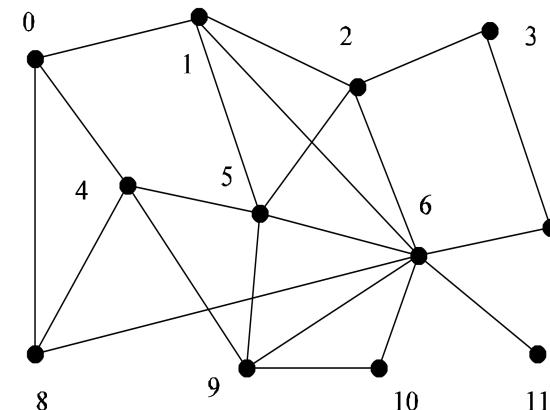
Task Interaction Graphs: An Example

Consider the problem of multiplying a sparse matrix \mathbf{A} with a vector \mathbf{b} . The following observations can be made:

- As before, the computation of each element of the result vector can be viewed as an independent task.
- Unlike a dense matrix-vector product though, only non-zero elements of matrix \mathbf{A} participate in the computation.
- If, for memory optimality, we also partition \mathbf{b} across tasks, then one can see that the task interaction graph of the computation is identical to the graph of the matrix \mathbf{A} (the graph for which \mathbf{A} represents the adjacency structure).



(a)



(b)



Task Interaction Graphs, Granularity, and Communication

In general, if the granularity of a decomposition is finer, the associated overhead (as a ratio of useful work associated with a task) increases.

Example: Consider the sparse matrix-vector product example from previous foil. Assume that each node takes unit time to process and each interaction (edge) causes an overhead of a unit time.

Viewing node 0 as an independent task involves a useful computation of one time unit and overhead (communication) of three time units.

Now, if we consider nodes 0, 4, and 5 as one task, then the task has useful computation totaling to three time units and communication corresponding to four time units (four edges). Clearly, this is a more favorable ratio than the former case.



Processes and Mapping

- In general, the number of tasks in a decomposition exceeds the number of processing elements available.
- For this reason, a parallel algorithm must also provide a mapping of tasks to processes.

Note: We refer to the mapping as being from tasks to processes, as opposed to processors. This is because typical programming APIs, as we shall see, do not allow easy binding of tasks to physical processors. Rather, we aggregate tasks into processes and rely on the system to map these processes to physical processors. We use processes, not in the UNIX sense of a process, rather, simply as a collection of tasks and associated data.



Processes and Mapping

- Appropriate mapping of tasks to processes is critical to the parallel performance of an algorithm.
- Mappings are determined by both the task dependency and task interaction graphs.
- Task dependency graphs can be used to ensure that work is equally spread across all processes at any point (minimum idling and optimal load balance).
- Task interaction graphs can be used to make sure that processes need minimum interaction with other processes (minimum communication).



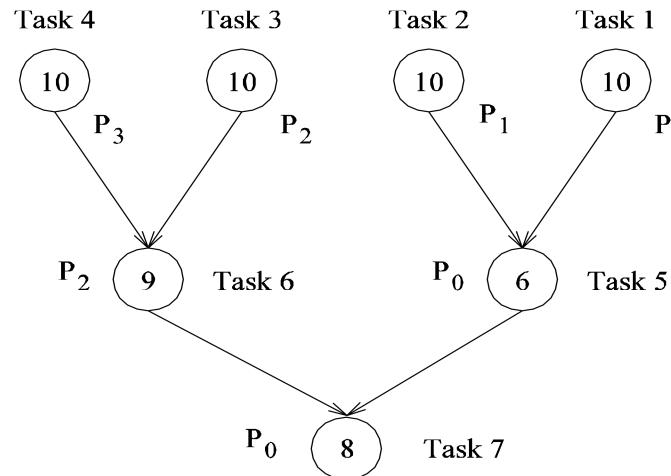
Processes and Mapping

An appropriate mapping must minimize parallel execution time by:

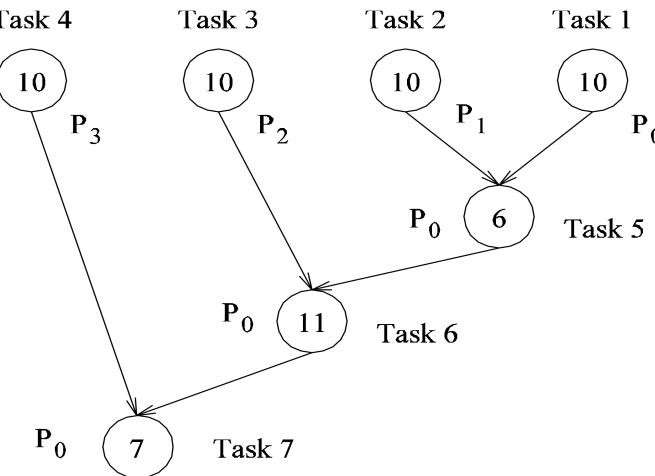
- Mapping independent tasks to different processes.
- Assigning tasks on critical path to processes as soon as they become available.
- Minimizing interaction between processes by mapping tasks with dense interactions to the same process.

Note: These criteria often conflict with each other. For example, a decomposition into one task (or no decomposition at all) minimizes interaction but does not result in a speedup at all! Can you think of other such conflicting cases?

Processes and Mapping: Example



(a)



(b)

Mapping tasks in the database query decomposition to processes. These mappings were arrived at by viewing the dependency graph in terms of levels (no two nodes in a level have dependencies). Tasks within a single level are then assigned to different processes.



Decomposition Techniques

So how does one decompose a task into various subtasks?

While there is no single recipe that works for all problems, we present a set of commonly used techniques that apply to broad classes of problems. These include:

- recursive decomposition
- data decomposition
- exploratory decomposition
- speculative decomposition



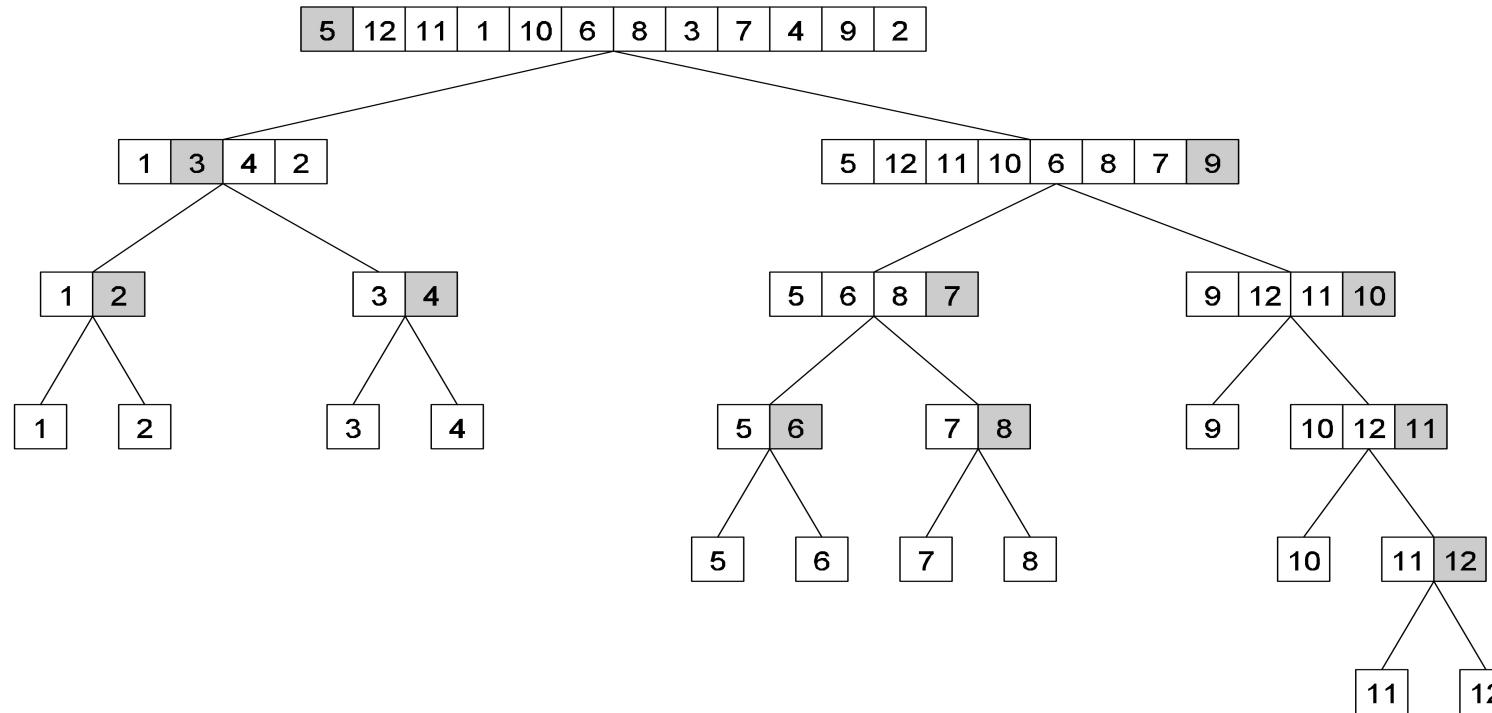
Recursive Decomposition

- Generally suited to problems that are solved using the divide-and-conquer strategy.
- A given problem is first decomposed into a set of sub-problems.
- These sub-problems are recursively decomposed further until a desired granularity is reached.



Recursive Decomposition: Example

A classic example of a divide-and-conquer algorithm on which we can apply recursive decomposition is Quicksort.



In this example, once the list has been partitioned around the pivot, each sublist can be processed concurrently (i.e., each sublist represents an independent subtask). This can be repeated recursively.



Recursive Decomposition: Example

The problem of finding the minimum number in a given list (or indeed any other associative operation such as sum, AND, etc.) can be fashioned as a divide-and-conquer algorithm. The following algorithm illustrates this.

We first start with a simple serial loop for computing the minimum entry in a given list:

```
1. procedure SERIAL_MIN (A, n)
2. begin
3.   min = A[0];
4.   for i := 1 to n - 1 do
5.     if (A[i] < min) min := A[i];
6.   endfor;
7.   return min;
8. end SERIAL_MIN
```



Recursive Decomposition: Example

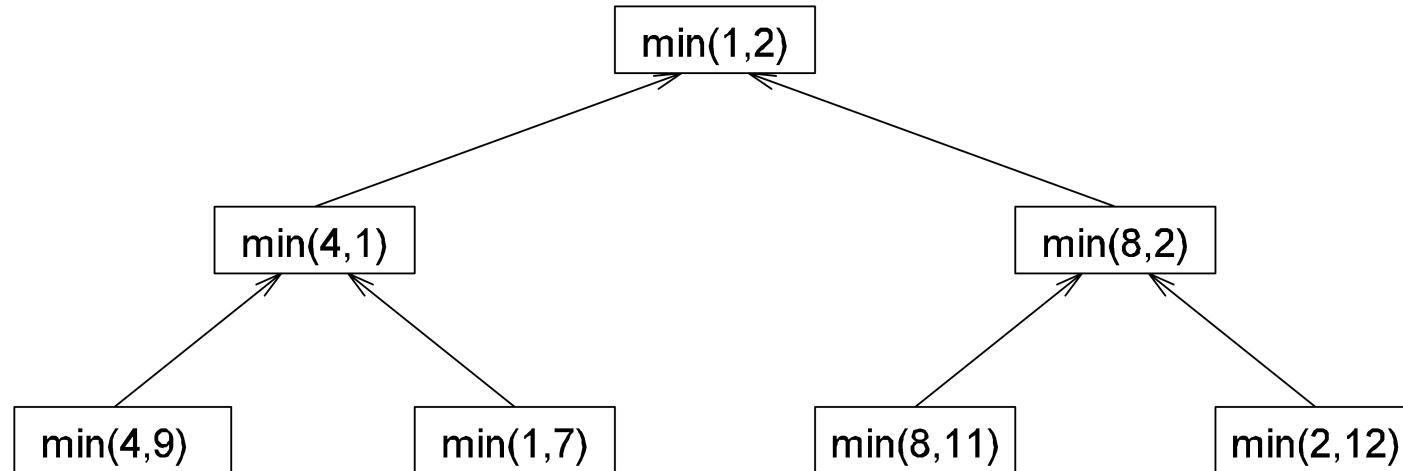
We can rewrite the loop as follows:

```
1. procedure RECURSIVE_MIN (A, n)
2. begin
3.   if ( n = 1 ) then
4.     min := A [0] ;
5.   else
6.     lmin := RECURSIVE_MIN ( A, n/2 );
7.     rmin := RECURSIVE_MIN ( &(A[n/2]), n - n/2 );
8.     if (lmin < rmin) then
9.       min := lmin;
10.    else
11.      min := rmin;
12.    endelse;
13.  endelse;
14.  return min;
15. end RECURSIVE_MIN
```



Recursive Decomposition: Example

The code in the previous foil can be decomposed naturally using a recursive decomposition strategy. We illustrate this with the following example of finding the minimum number in the set $\{4, 9, 1, 7, 8, 11, 2, 12\}$. The task dependency graph associated with this computation is as follows:



Data Decomposition

- Identify the data on which computations are performed.
- Partition this data across various tasks.
- This partitioning induces a decomposition of the problem.
- Data can be partitioned in various ways - this critically impacts performance of a parallel algorithm.



Data Decomposition: Output Data Decomposition

- Often, each element of the output can be computed independently of others (but simply as a function of the input).
- A partition of the output across tasks decomposes the problem naturally.



Output Data Decomposition: Example

Consider the problem of multiplying two $n \times n$ matrices \mathbf{A} and \mathbf{B} to yield matrix \mathbf{C} . The output matrix \mathbf{C} can be partitioned into four tasks as follows:

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 1: $C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1}$

Task 2: $C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}$

Task 3: $C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1}$

Task 4: $C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}$



Output Data Decomposition: Example

A partitioning of output data does not result in a unique decomposition into tasks. For example, for the same problem as in previous foil, with identical output data distribution, we can derive the following two (other) decompositions:

Decomposition I	Decomposition II
Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$	Task 1: $\mathbf{C}_{1,1} = \mathbf{A}_{1,1} \mathbf{B}_{1,1}$
Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$	Task 2: $\mathbf{C}_{1,1} = \mathbf{C}_{1,1} + \mathbf{A}_{1,2} \mathbf{B}_{2,1}$
Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,1} \mathbf{B}_{1,2}$	Task 3: $\mathbf{C}_{1,2} = \mathbf{A}_{1,2} \mathbf{B}_{2,2}$
Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,2} \mathbf{B}_{2,2}$	Task 4: $\mathbf{C}_{1,2} = \mathbf{C}_{1,2} + \mathbf{A}_{1,1} \mathbf{B}_{1,2}$
Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,1} \mathbf{B}_{1,1}$	Task 5: $\mathbf{C}_{2,1} = \mathbf{A}_{2,2} \mathbf{B}_{2,1}$
Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,2} \mathbf{B}_{2,1}$	Task 6: $\mathbf{C}_{2,1} = \mathbf{C}_{2,1} + \mathbf{A}_{2,1} \mathbf{B}_{1,1}$
Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$	Task 7: $\mathbf{C}_{2,2} = \mathbf{A}_{2,1} \mathbf{B}_{1,2}$
Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$	Task 8: $\mathbf{C}_{2,2} = \mathbf{C}_{2,2} + \mathbf{A}_{2,2} \mathbf{B}_{2,2}$



Output Data Decomposition: Example

Consider the problem of counting the instances of given itemsets in a database of transactions. In this case, the output (itemset frequencies) can be partitioned across tasks.

(a) Transactions (input), itemsets (input), and frequencies (output)

Database Transactions	Items	Itemset Frequency
	Itemsets	
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,	C, D	1
A, E, F, K, L	D, K	2
B, C, D, G, H, L	B, C, F	0
G, H, L	C, D, K	0
D, E, F, K, L		
F, G, H, L		

(b) Partitioning the frequencies (and itemsets) among the tasks

Database Transactions	Items	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	3
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	2
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 1

Database Transactions	Items	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, K	2
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K,		
A, E, F, K, L		
B, C, D, G, H, L		
G, H, L		
D, E, F, K, L		
F, G, H, L		

task 2



Output Data Decomposition: Example

From the previous example, the following observations can be made:

- If the database of transactions is replicated across the processes, each task can be independently accomplished with no communication.
- If the database is partitioned across processes as well (for reasons of memory utilization), each task first computes partial counts. These counts are then aggregated at the appropriate task.



Input Data Partitioning

- Generally applicable if each output can be naturally computed as a function of the input.
- In many cases, this is the only natural decomposition because the output is not clearly known a-priori (e.g., the problem of finding the minimum in a list, sorting a given list, etc.).
- A task is associated with each input data partition. The task performs as much of the computation with its part of the data. Subsequent processing combines these partial results.



Input Data Partitioning: Example

In the database counting example, the input (i.e., the transaction set) can be partitioned. This induces a task decomposition in which each task generates partial counts for all itemsets. These are combined subsequently for aggregate counts.

Partitioning the transactions among the tasks

Database Transactions	Itemsets	Itemset Frequency	task 1		task 2	
			Itemsets	Itemset Frequency	Itemsets	Itemset Frequency
A, B, C, E, G, H	A, B, C	1	A, B, C	0	D, E	1
B, D, E, F, K, L	D, E	2	D, E	1	C, F, G	0
A, B, F, H, L	C, F, G	0	C, F, G	0	A, E	1
D, E, F, H	A, E	1	A, E	1	C, D	1
F, G, H, K,	C, D	0	C, D	1	D, K	1
	D, K	1	D, K	1	B, C, F	0
	B, C, F	0	B, C, F	0	C, D, K	0
	C, D, K	0	C, D, K	0		



Partitioning Input and Output Data

Often input and output data decomposition can be combined for a higher degree of concurrency. For the itemset counting example, the transaction set (input) and itemset counts (output) can both be decomposed as follows:

Partitioning both transactions and frequencies among the tasks

Database Transactions	Items	Itemset Frequency
A, B, C, E, G, H	A, B, C	1
B, D, E, F, K, L	D, E	2
A, B, F, H, L	C, F, G	0
D, E, F, H	A, E	1
F, G, H, K,		

task 1

Database Transactions	Items	Itemset Frequency
A, B, C, E, G, H	C, D	0
B, D, E, F, K, L	D, K	1
A, B, F, H, L	B, C, F	0
D, E, F, H	C, D, K	0
F, G, H, K,		

task 2

Database Transactions	Items	Itemset Frequency
A, E, F, K, L	A, B, C	0
B, C, D, G, H, L	D, E	1
G, H, L	C, F, G	0
D, E, F, K, L	A, E	1
F, G, H, L		

task 3

Database Transactions	Items	Itemset Frequency
A, E, F, K, L	C, D	1
B, C, D, G, H, L	D, K	1
G, H, L	B, C, F	0
D, E, F, K, L	C, D, K	0
F, G, H, L		

task 4



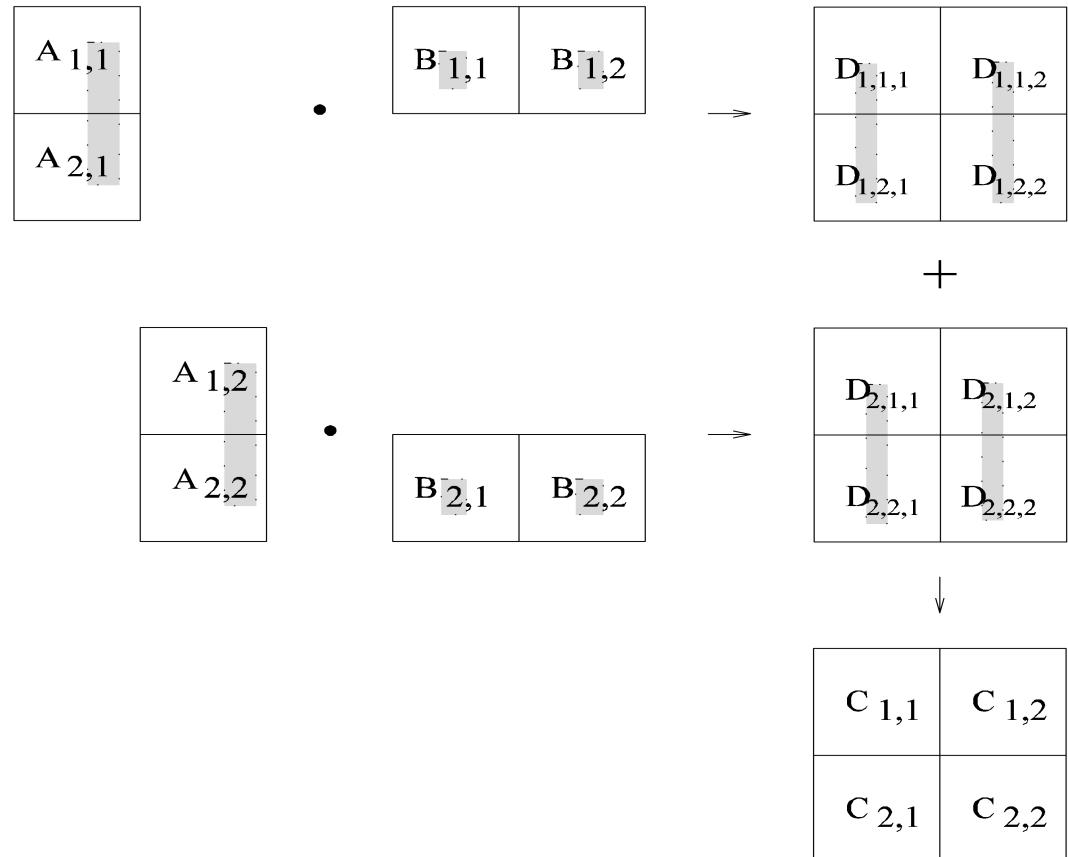
Intermediate Data Partitioning

- Computation can often be viewed as a sequence of transformation from the input to the output data.
- In these cases, it is often beneficial to use one of the intermediate stages as a basis for decomposition.



Intermediate Data Partitioning: Example

Let us revisit the example of dense matrix multiplication. We first show how we can visualize this computation in terms of intermediate matrices D .



Intermediate Data Partitioning: Example

A decomposition of intermediate data structure leads to the following decomposition into 8 + 4 tasks:

Stage I

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} \left(\begin{array}{cc} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{array} \right) \\ \left(\begin{array}{cc} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{array} \right) \end{pmatrix}$$

Stage II

$$\begin{pmatrix} D_{1,1,1} & D_{1,1,2} \\ D_{1,2,2} & D_{1,2,2} \end{pmatrix} + \begin{pmatrix} D_{2,1,1} & D_{2,1,2} \\ D_{2,2,2} & D_{2,2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Task 01: $D_{1,1,1} = A_{1,1} B_{1,1}$

Task 03: $D_{1,1,2} = A_{1,1} B_{1,2}$

Task 05: $D_{1,2,1} = A_{2,1} B_{1,1}$

Task 07: $D_{1,2,2} = A_{2,1} B_{1,2}$

Task 09: $C_{1,1} = D_{1,1,1} + D_{2,1,1}$

Task 11: $C_{2,1} = D_{1,2,1} + D_{2,2,1}$

Task 02: $D_{2,1,1} = A_{1,2} B_{2,1}$

Task 04: $D_{2,1,2} = A_{1,2} B_{2,2}$

Task 06: $D_{2,2,1} = A_{2,2} B_{2,1}$

Task 08: $D_{2,2,2} = A_{2,2} B_{2,2}$

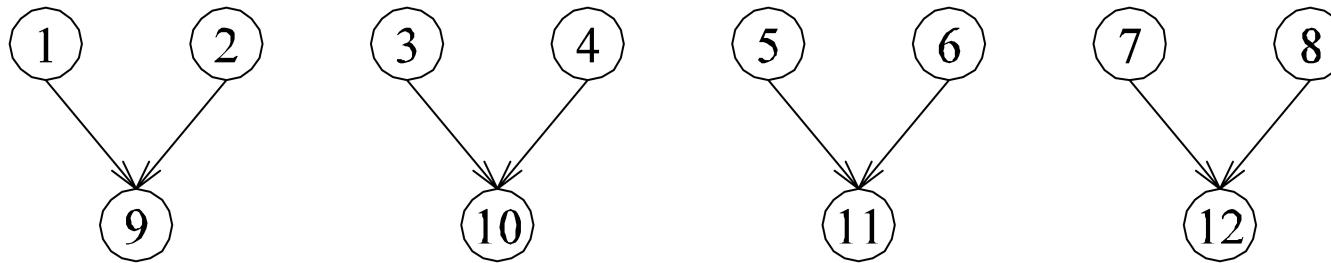
Task 10: $C_{1,2} = D_{1,1,2} + D_{2,1,2}$

Task 12: $C_{2,2} = D_{1,2,2} + D_{2,2,2}$



Intermediate Data Partitioning: Example

The task dependency graph for the decomposition (shown in previous foil) into 12 tasks is as follows:



The Owner Computes Rule

- The *Owner Computes Rule* generally states that the process assigned a particular data item is responsible for all computation associated with it.
- In the case of input data decomposition, the owner computes rule implies that all computations that use the input data are performed by the process.
- In the case of output data decomposition, the owner computes rule implies that the output is computed by the process to which the output data is assigned.



Principles of Parallel Algorithm Design 2

Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

To accompany the text “Introduction to Parallel Computing”,
Addison Wesley, 2003.



Exploratory Decomposition

- In many cases, the decomposition of the problem goes hand-in-hand with its execution.
- These problems typically involve the exploration (search) of a state space of solutions.
- Problems in this class include a variety of discrete optimization problems (0/1 integer programming, QAP, etc.), theorem proving, game playing, etc.



Exploratory Decomposition: Example

A simple application of exploratory decomposition is in the solution to a 15 puzzle (a tile puzzle). We show a sequence of three moves that transform a given initial state (a) to desired final state (d).

1	2	3	4
5	6	7	8
9	10	11	
13	14	15	12

(a)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

(b)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	12

(c)

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

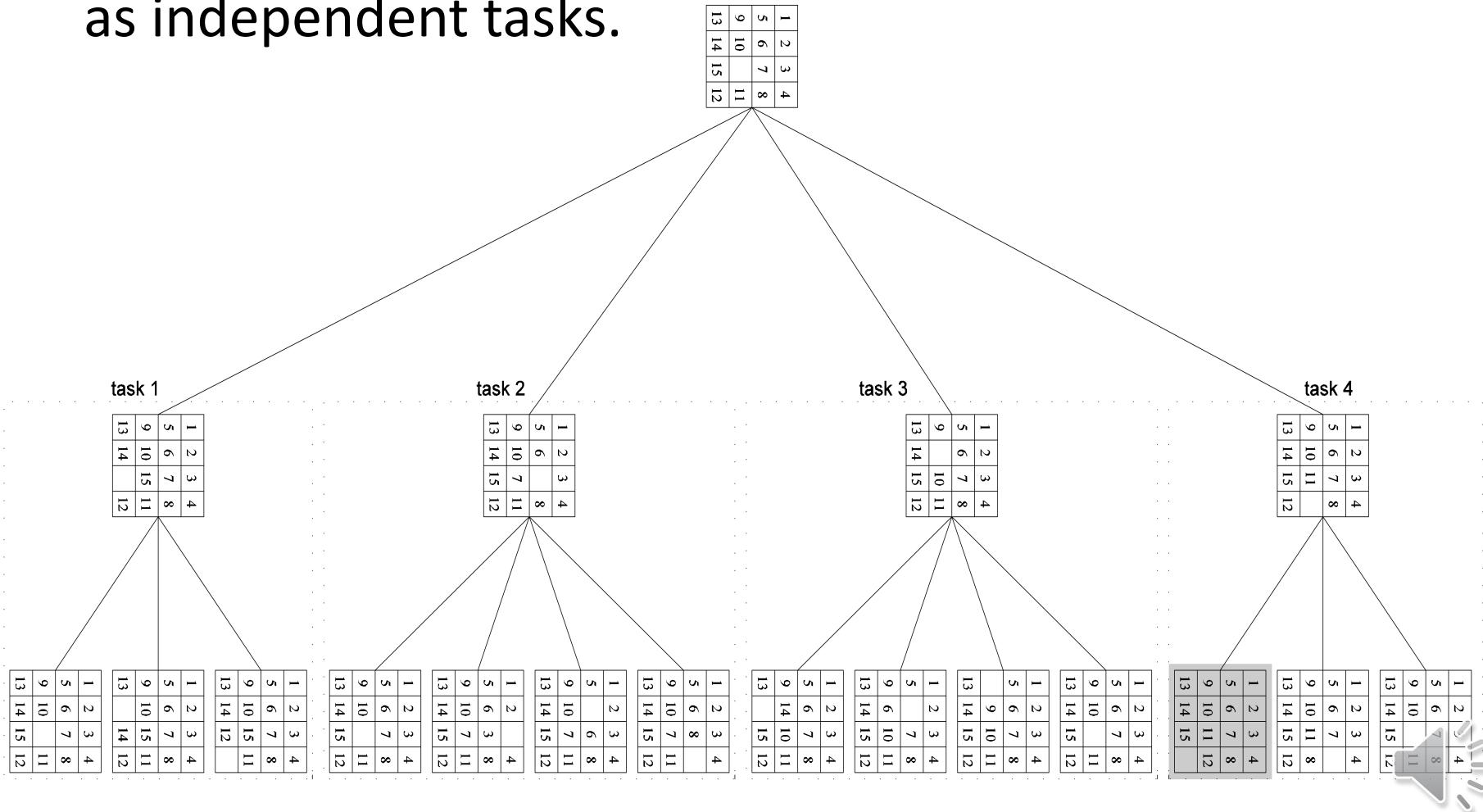
(d)

Of-course, the problem of computing the solution, in general, is much more difficult than in this simple example.



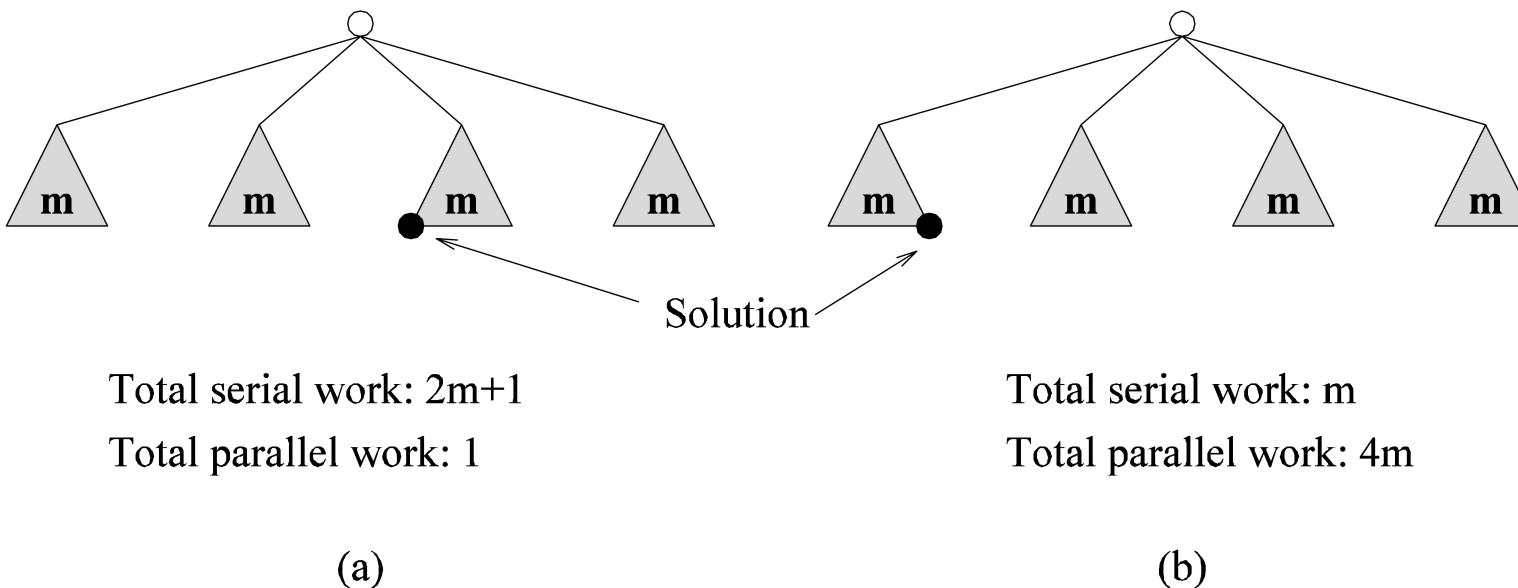
Exploratory Decomposition: Example

The state space can be explored by generating various successor states of the current state and to view them as independent tasks.



Exploratory Decomposition: Anomalous Computations

- In many instances of exploratory decomposition, the decomposition technique may change the amount of work done by the parallel formulation.
- This change results in super- or sub-linear speedups.



Speculative Decomposition

- In some applications, dependencies between tasks are not known a-priori.
- For such applications, it is impossible to identify independent tasks.
- There are generally two approaches to dealing with such applications: conservative approaches, which identify independent tasks only when they are guaranteed to not have dependencies, and, optimistic approaches, which schedule tasks even when they may potentially be erroneous.
- Conservative approaches may yield little concurrency and optimistic approaches may require roll-back mechanism in the case of an error.



Speculative Decomposition: Example

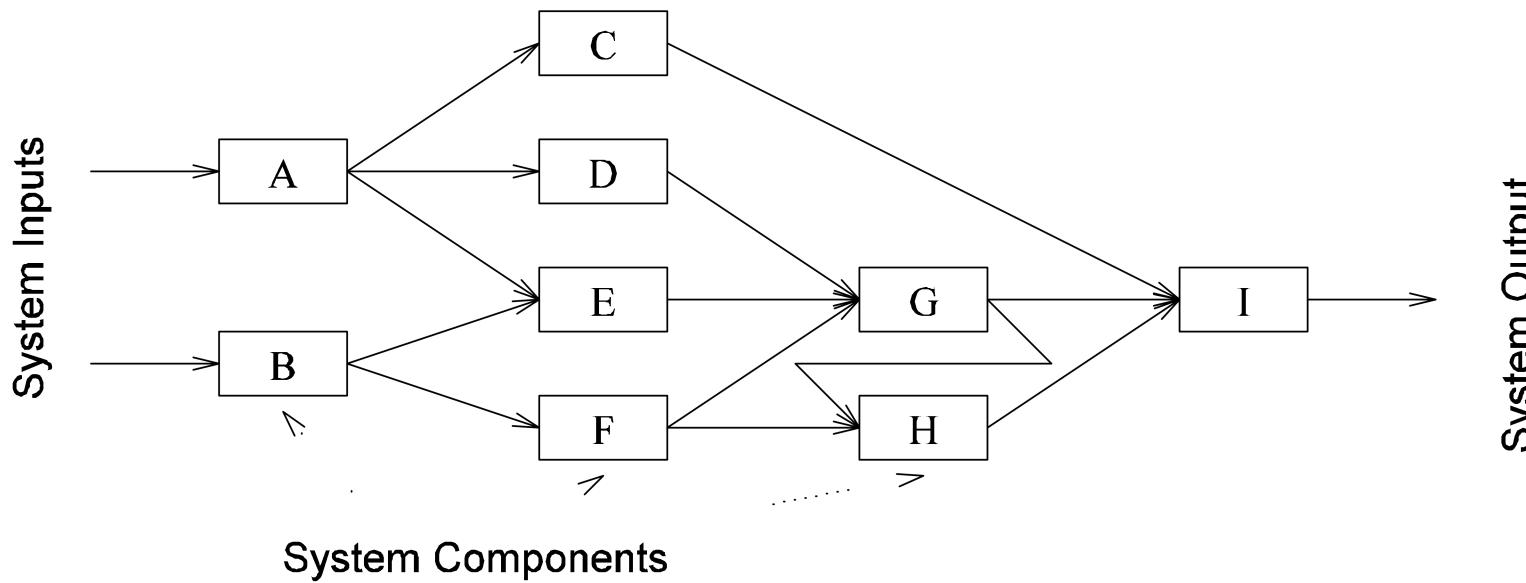
A classic example of speculative decomposition is in discrete event simulation.

- The central data structure in a discrete event simulation is a time-ordered event list.
- Events are extracted precisely in time order, processed, and if required, resulting events are inserted back into the event list.
- Consider your day today as a discrete event system - you get up, get ready, drive to work, work, eat lunch, work some more, drive back, eat dinner, and sleep.
- Each of these events may be processed independently, however, in driving to work, you might meet with an unfortunate accident and not get to work at all.
- Therefore, an optimistic scheduling of other events will have to be rolled back.



Speculative Decomposition: Example

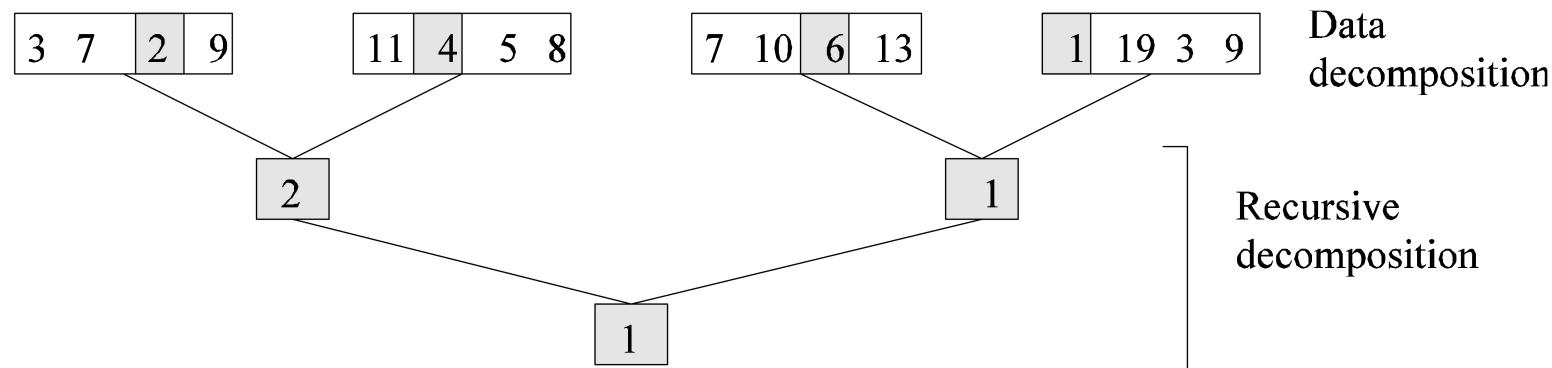
Another example is the simulation of a network of nodes (for instance, an assembly line or a computer network through which packets pass). The task is to simulate the behavior of this network for various inputs and node delay parameters (note that networks may become unstable for certain values of service rates, queue sizes, etc.).



Hybrid Decompositions

Often, a mix of decomposition techniques is necessary for decomposing a problem. Consider the following examples:

- In quicksort, recursive decomposition alone limits concurrency (Why?). A mix of data and recursive decompositions is more desirable.
- In discrete event simulation, there might be concurrency in task processing. A mix of speculative decomposition and data decomposition may work well.
- Even for simple problems like finding a minimum of a list of numbers, a mix of data and recursive decomposition works well.



Characteristics of Tasks

Once a problem has been decomposed into independent tasks, the characteristics of these tasks critically impact choice and performance of parallel algorithms. Relevant task characteristics include:

- Task generation.
- Task sizes.
- Size of data associated with tasks.



Task Generation

- Static task generation: Concurrent tasks can be identified a-priori. Typical matrix operations, graph algorithms, image processing applications, and other regularly structured problems fall in this class. These can typically be decomposed using data or recursive decomposition techniques.
- Dynamic task generation: Tasks are generated as we perform computation. A classic example of this is in game playing - each 15 puzzle board is generated from the previous one. These applications are typically decomposed using exploratory or speculative decompositions.



Task Sizes

- Task sizes may be uniform (i.e., all tasks are the same size) or non-uniform.
- Non-uniform task sizes may be such that they can be determined (or estimated) a-priori or not.
- Examples in this class include discrete optimization problems, in which it is difficult to estimate the effective size of a state space.



Size of Data Associated with Tasks

- The size of data associated with a task may be small or large when viewed in the context of the size of the task.
- A small context of a task implies that an algorithm can easily communicate this task to other processes dynamically (e.g., the 15 puzzle).
- A large context ties the task to a process, or alternately, an algorithm may attempt to reconstruct the context at another processes as opposed to communicating the context of the task (e.g., 0/1 integer programming).



Characteristics of Task Interactions

- Tasks may communicate with each other in various ways. The associated dichotomy is:
- Static interactions: The tasks and their interactions are known a-priori. These are relatively simpler to code into programs.
- Dynamic interactions: The timing or interacting tasks cannot be determined a-priori. These interactions are harder to code, especially, as we shall see, using message passing APIs.



Characteristics of Task Interactions

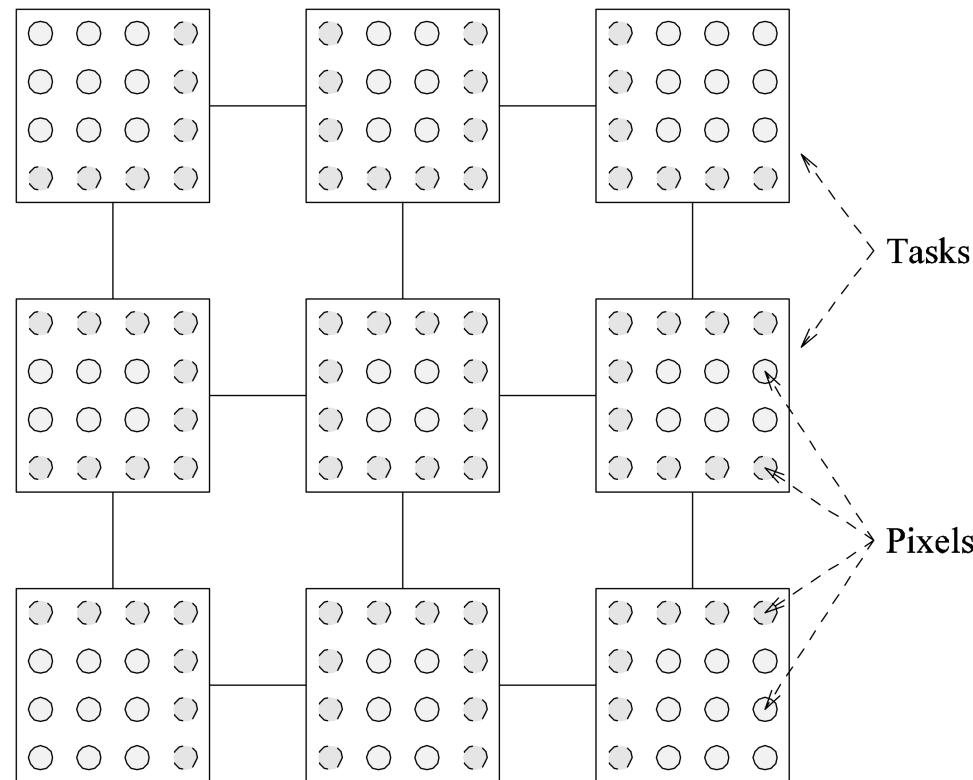
- Regular interactions: There is a definite pattern (in the graph sense) to the interactions. These patterns can be exploited for efficient implementation.
- Irregular interactions: Interactions lack well-defined topologies.



Characteristics of Task Interactions: Example

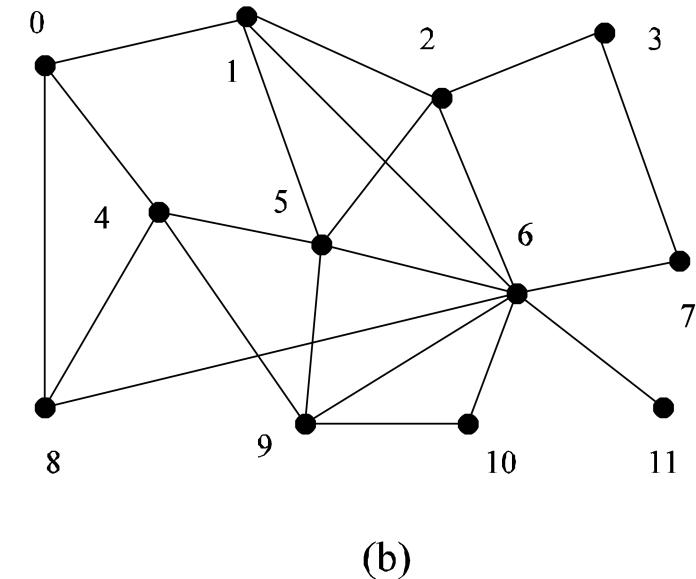
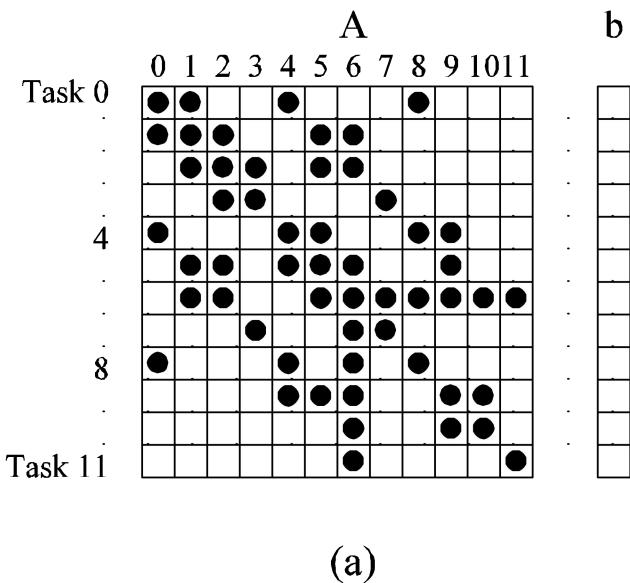
A simple example of a regular static interaction pattern is in image

dithering. The underlying communication pattern is a structured (2-D mesh) one as shown here:



Characteristics of Task Interactions: Example

The multiplication of a sparse matrix with a vector is a good example of a static irregular interaction pattern. Here is an example of a sparse matrix and its associated interaction pattern.



Characteristics of Task Interactions

- Interactions may be read-only or read-write.
- In read-only interactions, tasks just read data items associated with other tasks.
- In read-write interactions tasks read, as well as modify data items associated with other tasks.
- In general, read-write interactions are harder to code, since they require additional synchronization primitives.



Characteristics of Task Interactions

- Interactions may be one-way or two-way.
- A one-way interaction can be initiated and accomplished by one of the two interacting tasks.
- A two-way interaction requires participation from both tasks involved in an interaction.
- One way interactions are somewhat harder to code in message passing APIs.



Mapping Techniques

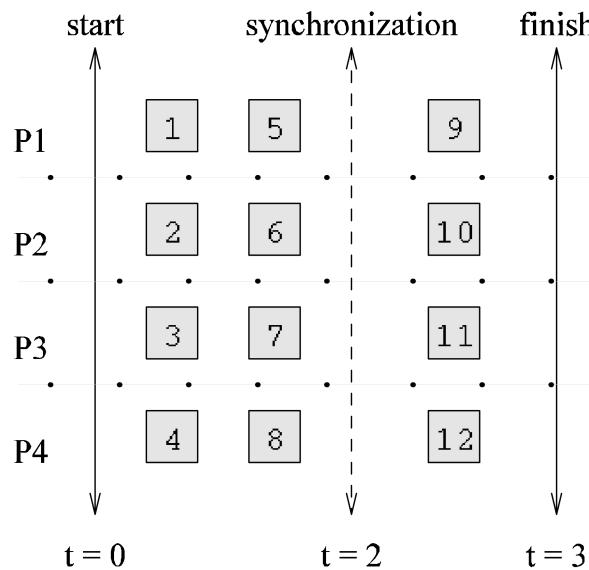
- Once a problem has been decomposed into concurrent tasks, these must be mapped to processes (that can be executed on a parallel platform).
- Mappings must minimize overheads.
- Primary overheads are communication and idling.
- Minimizing these overheads often represents contradicting objectives.
- Assigning all work to one processor trivially minimizes communication at the expense of significant idling.



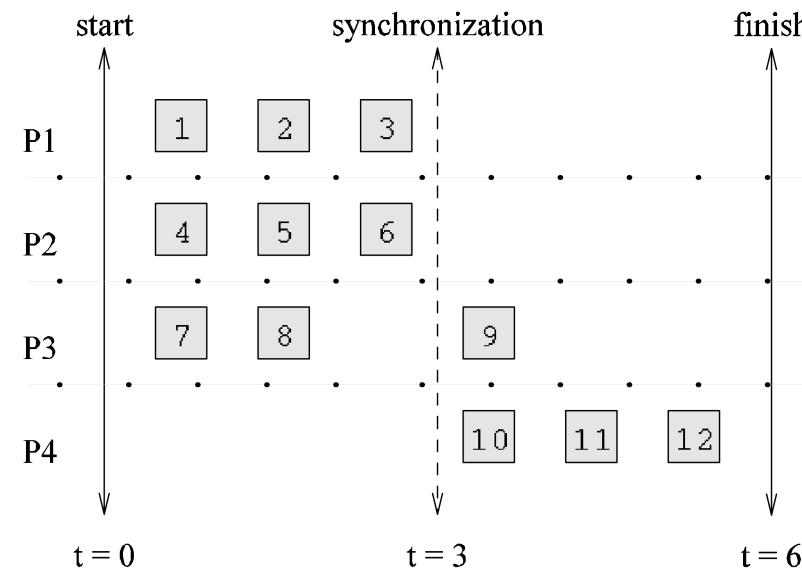
Mapping Techniques for Minimum Idling

Mapping must simultaneously minimize idling and load balance.

Merely balancing load does not minimize idling.



(a)



(b)



Mapping Techniques for Minimum Idling

Mapping techniques can be static or dynamic.

- Static Mapping: Tasks are mapped to processes a-priori. For this to work, we must have a good estimate of the size of each task. Even in these cases, the problem may be NP complete.
- Dynamic Mapping: Tasks are mapped to processes at runtime. This may be because the tasks are generated at runtime, or that their sizes are not known.

Other factors that determine the choice of techniques include the size of data associated with a task and the nature of underlying domain.



Schemes for Static Mapping

- Mappings based on data partitioning.
- Mappings based on task graph partitioning.
- Hybrid mappings.



Mappings Based on Data Partitioning

We can combine data partitioning with the "owner-computes" rule to partition the computation into subtasks. The simplest data decomposition schemes for dense matrices are 1-D block distribution schemes.

row-wise distribution

P_0
P_1
P_2
P_3
P_4
P_5
P_6
P_7

column-wise distribution

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7



Block Array Distribution Schemes

Block distribution schemes can be generalized to higher dimensions as well.

P_0	P_1	P_2	P_3
P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}
P_{12}	P_{13}	P_{14}	P_{15}

(a)

P_0	P_1	P_2	P_3	P_4	P_5	P_6	P_7
P_8	P_9	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}

(b)



Block Array Distribution Schemes: Examples

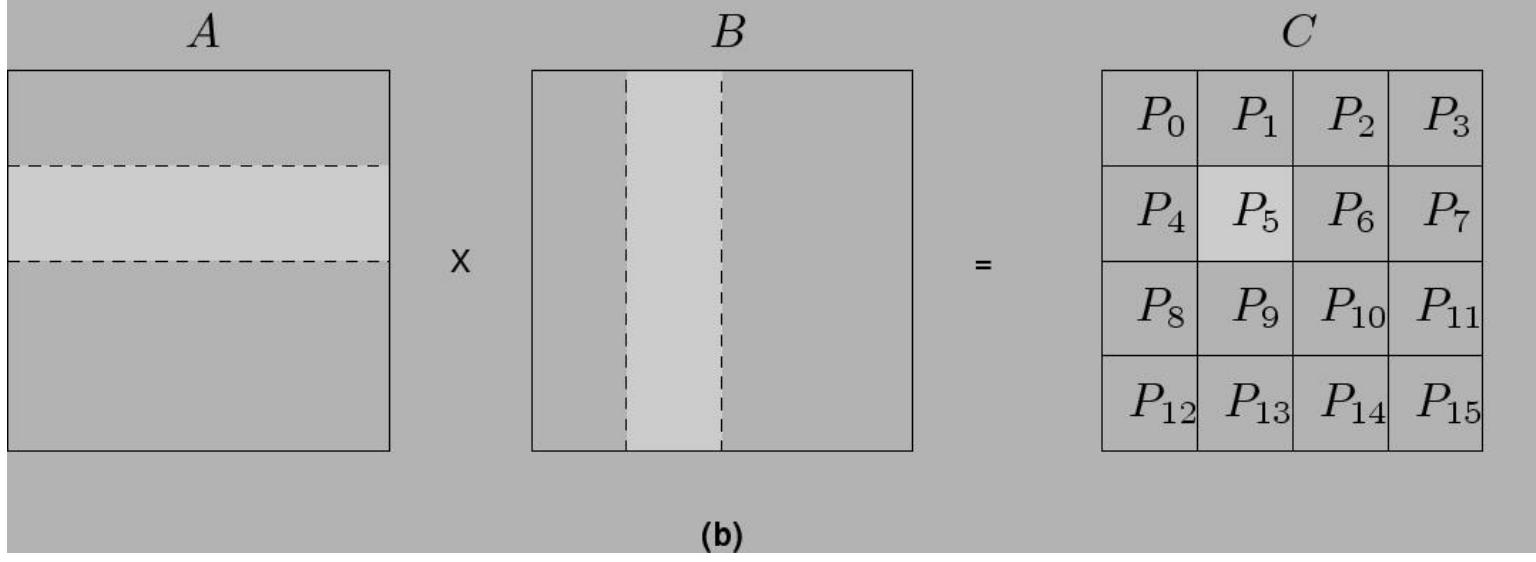
- For multiplying two dense matrices A and B , we can partition the output matrix C using a block decomposition.
- For load balance, we give each task the same number of elements of C . (Note that each element of C corresponds to a single dot product.)
- The choice of precise decomposition (1-D or 2-D) is determined by the associated communication overhead.
- In general, higher dimension decomposition allows the use of larger number of processes.



Data Sharing in Dense Matrix Multiplication



(a)



(b)



Cyclic and Block Cyclic Distributions

- If the amount of computation associated with data items varies, a block decomposition may lead to significant load imbalances.
- A simple example of this is in LU decomposition (or Gaussian Elimination) of dense matrices.



LU Factorization of a Dense Matrix

A decomposition of LU factorization into 14 tasks - notice the significant load imbalance.

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \rightarrow \begin{pmatrix} L_{1,1} & 0 & 0 \\ L_{2,1} & L_{2,2} & 0 \\ L_{3,1} & L_{3,2} & L_{3,3} \end{pmatrix} \cdot \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} \\ 0 & U_{2,2} & U_{2,3} \\ 0 & 0 & U_{3,3} \end{pmatrix}$$

$$1: A_{1,1} \rightarrow L_{1,1}U_{1,1}$$

$$2: L_{2,1} = A_{2,1}U_{1,1}^{-1}$$

$$3: L_{3,1} = A_{3,1}U_{1,1}^{-1}$$

$$4: U_{1,2} = L_{1,1}^{-1}A_{1,2}$$

$$5: U_{1,3} = L_{1,1}^{-1}A_{1,3}$$

$$6: A_{2,2} = A_{2,2} - L_{2,1}U_{1,2}$$

$$7: A_{3,2} = A_{3,2} - L_{3,1}U_{1,2}$$

$$8: A_{2,3} = A_{2,3} - L_{2,1}U_{1,3}$$

$$9: A_{3,3} = A_{3,3} - L_{3,1}U_{1,3}$$

$$10: A_{2,2} \rightarrow L_{2,2}U_{2,2}$$

$$11: L_{3,2} = A_{3,2}U_{2,2}^{-1}$$

$$12: U_{2,3} = L_{2,2}^{-1}A_{2,3}$$

$$13: A_{3,3} = A_{3,3} - L_{3,2}U_{2,3}$$

$$14: A_{3,3} \rightarrow L_{3,3}U_{3,3}$$



Block Cyclic Distributions

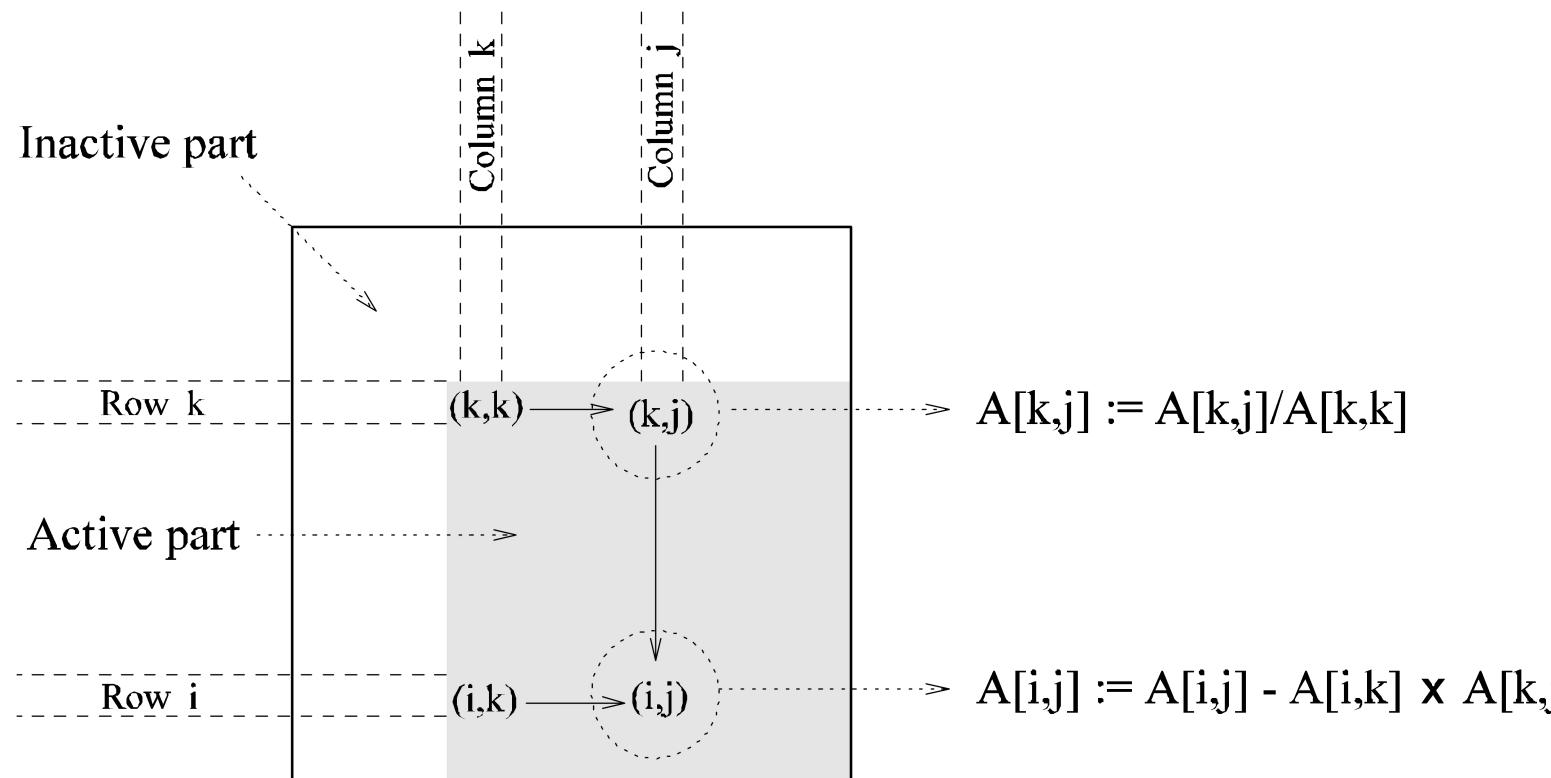
- Variation of the block distribution scheme that can be used to alleviate the load-imbalance and idling problems.
- Partition an array into many more blocks than the number of available processes.
- Blocks are assigned to processes in a round-robin manner so that each process gets several non-adjacent blocks.



Block-Cyclic Distribution for Gaussian Elimination

The active part of the matrix in Gaussian Elimination changes.

By assigning blocks in a block-cyclic fashion, each processor receives blocks from different parts of the matrix.



Block-Cyclic Distribution: Examples

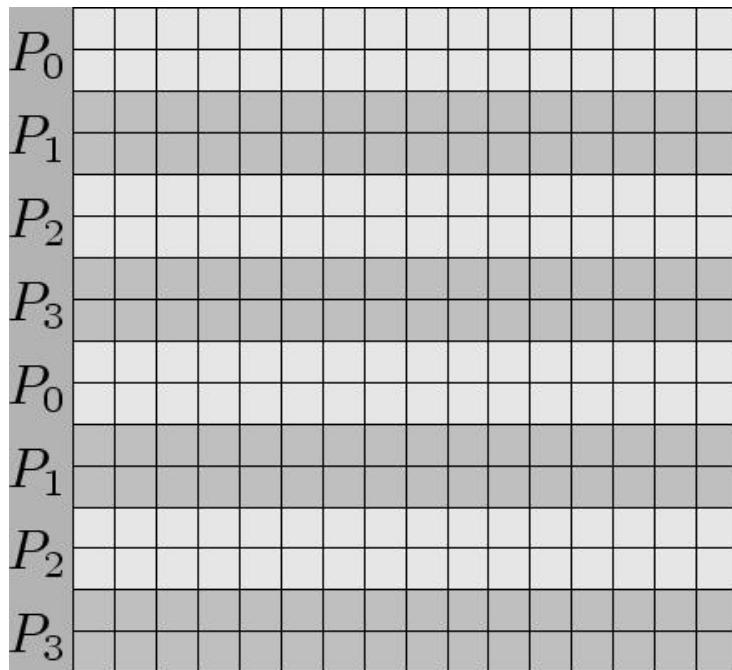
One- and two-dimensional block-cyclic distributions among 4 processes.

P₀ T ₁	P₃ T ₄	P₆ T ₅
P₁ T ₂	P₄ T ₆ T ₁₀	P₇ T ₈ T ₁₂
P₂ T ₃	P₅ T ₇ T ₁₁	P₈ T ₉ T ₁₃ T ₁₄

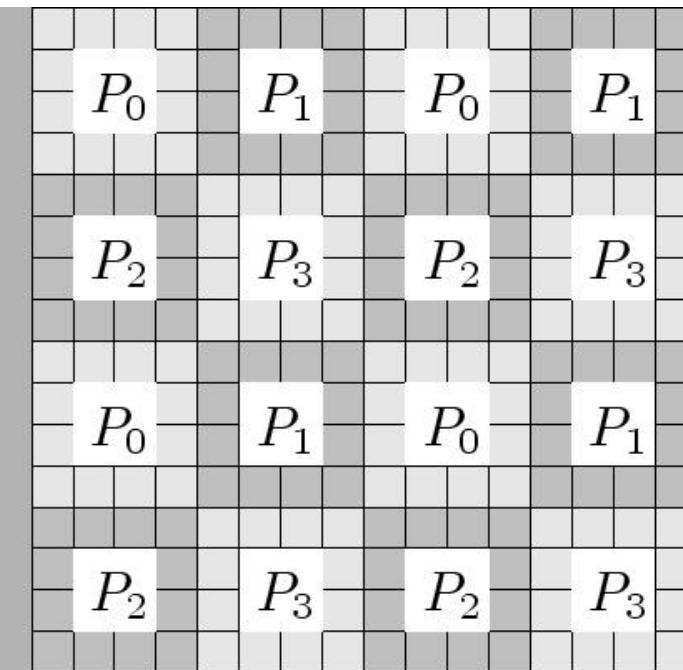


Block-Cyclic Distribution

- A cyclic distribution is a special case in which block size is one.
- A block distribution is a special case in which block size is n/p , where n is the dimension of the matrix and p is the number of processes.



(a)



(b)

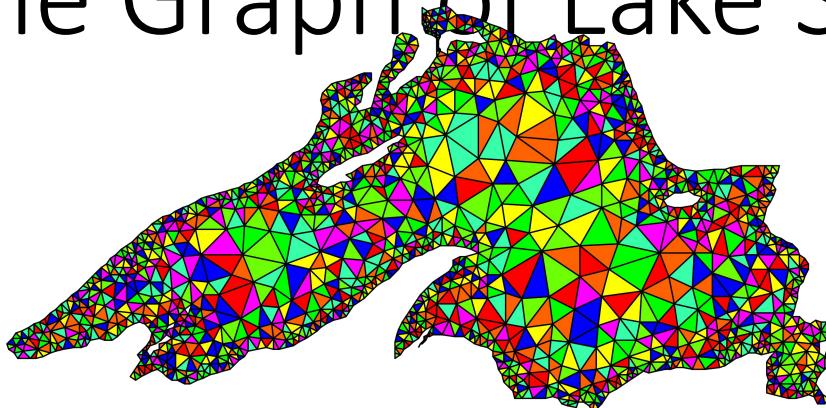


Graph Partitioning Based Data Decomposition

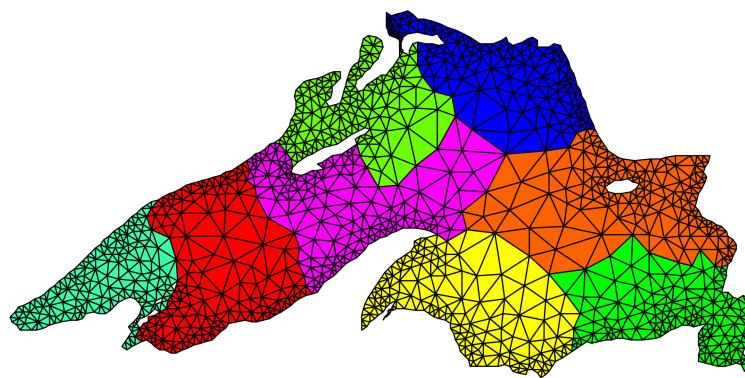
- In case of sparse matrices, block decompositions are more complex.
- Consider the problem of multiplying a sparse matrix with a vector.
- The graph of the matrix is a useful indicator of the work (number of nodes) and communication (the degree of each node).
- In this case, we would like to partition the graph so as to assign equal number of nodes to each process, while minimizing edge count of the graph partition.



Partitioning the Graph of Lake Superior



Random Partitioning



Partitioning for minimum edge-cut.



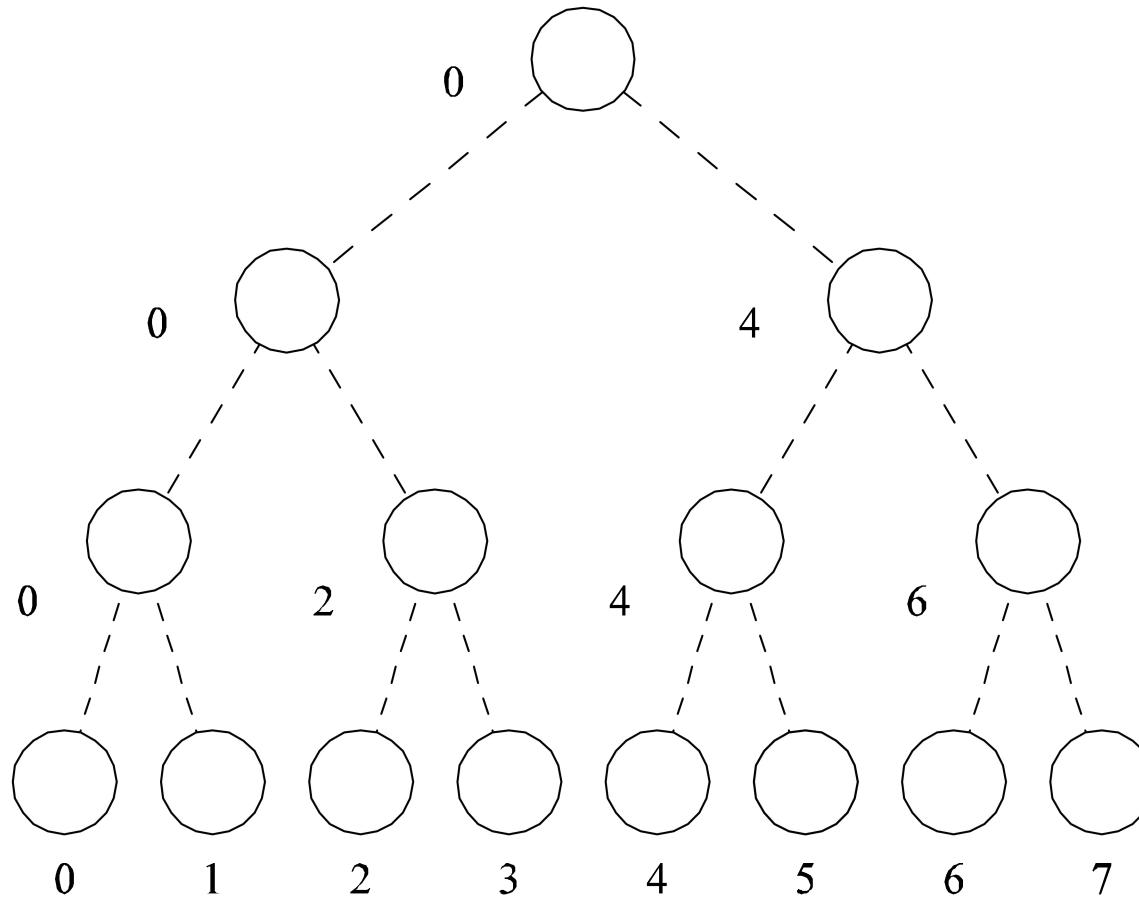
Mappings Based on Task Partitioning

- Partitioning a given task-dependency graph across processes.
- Determining an optimal mapping for a general task-dependency graph is an NP-complete problem.
- Excellent heuristics exist for structured graphs.



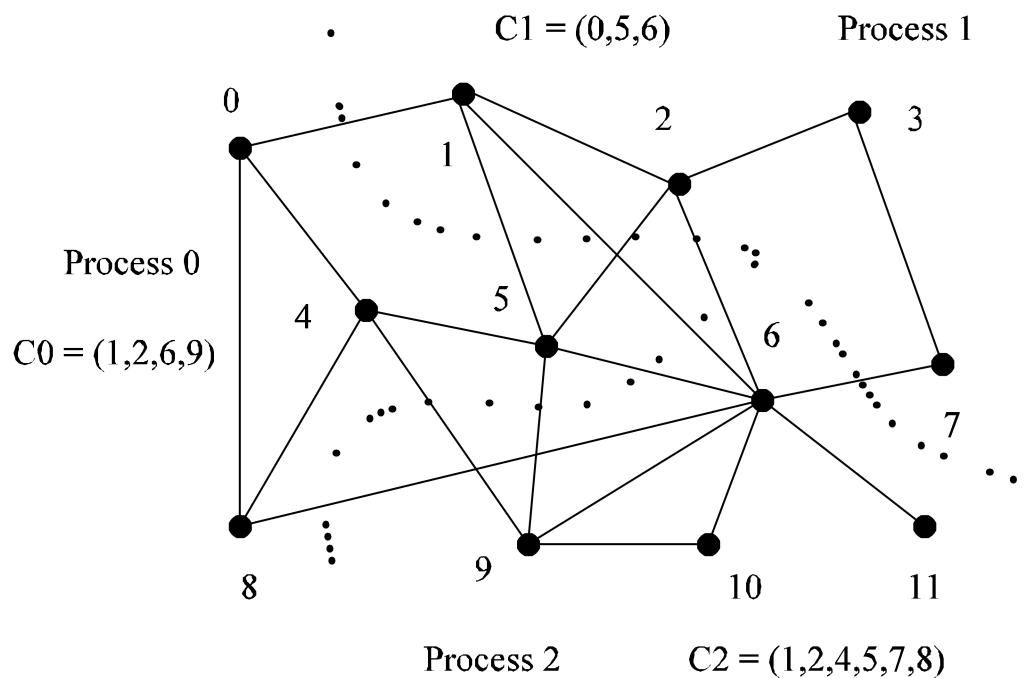
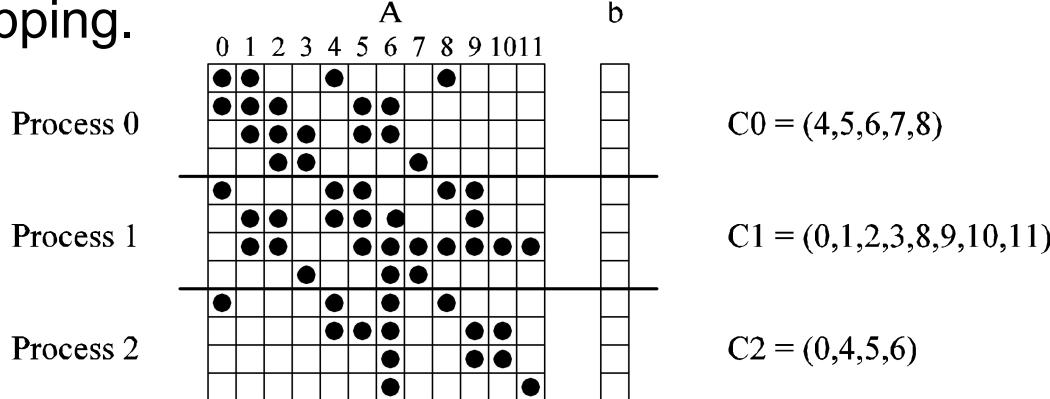
Task Partitioning: Mapping a Binary Tree

Example illustrates the dependency graph of one view of quick-sort and how it can be assigned to processes in a hypercube.



Task Partitioning: Mapping a Sparse Graph

Sparse graph for computing a sparse matrix-vector product and its mapping.

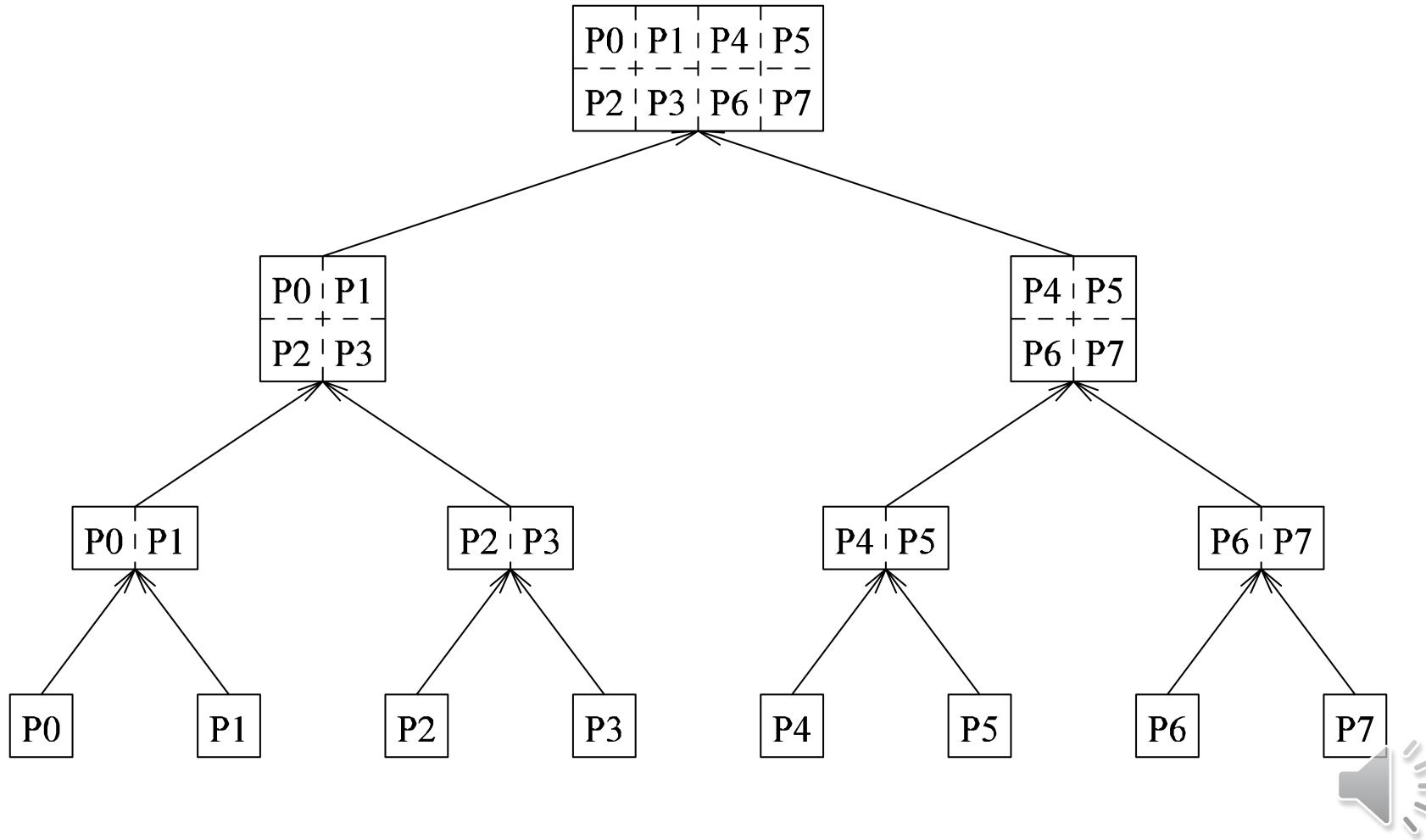


Hierarchical Mappings

- Sometimes a single mapping technique is inadequate.
- For example, the task mapping of the binary tree (quicksort) cannot use a large number of processors.
- For this reason, task mapping can be used at the top level and data partitioning within each level.



An example of task partitioning at top level with data partitioning at the lower level.



Schemes for Dynamic Mapping

- Dynamic mapping is sometimes also referred to as dynamic load balancing, since load balancing is the primary motivation for dynamic mapping.
- Dynamic mapping schemes can be centralized or distributed.



Centralized Dynamic Mapping

- Processes are designated as masters or slaves.
- When a process runs out of work, it requests the master for more work.
- When the number of processes increases, the master may become the bottleneck.
- To alleviate this, a process may pick up a number of tasks (a chunk) at one time. This is called Chunk scheduling.
- Selecting large chunk sizes may lead to significant load imbalances as well.
- A number of schemes have been used to gradually decrease chunk size as the computation progresses.



Distributed Dynamic Mapping

- Each process can send or receive work from other processes.
- This alleviates the bottleneck in centralized schemes.
- There are four critical questions: how are sensing and receiving processes paired together, who initiates work transfer, how much work is transferred, and when is a transfer triggered?
- Answers to these questions are generally application specific. We will look at some of these techniques later in this class.



Minimizing Interaction Overheads

- Maximize data locality: Where possible, reuse intermediate data. Restructure computation so that data can be reused in smaller time windows.
- Minimize volume of data exchange: There is a cost associated with each word that is communicated. For this reason, we must minimize the volume of data communicated.
- Minimize frequency of interactions: There is a startup cost associated with each interaction. Therefore, try to merge multiple interactions to one, where possible.
- Minimize contention and hot-spots: Use decentralized techniques, replicate data where necessary.



Minimizing Interaction Overheads (continued)

- Overlapping computations with interactions: Use non-blocking communications, multithreading, and prefetching to hide latencies.
- Replicating data or computations.
- Using group communications instead of point-to-point primitives.
- Overlap interactions with other interactions.



Parallel Algorithm Models

An algorithm model is a way of structuring a parallel algorithm by selecting a decomposition and mapping technique and applying the appropriate strategy to minimize interactions.

- Data Parallel Model: Tasks are statically (or semi-statically) mapped to processes and each task performs similar operations on different data.
- Task Graph Model: Starting from a task dependency graph, the interrelationships among the tasks are utilized to promote locality or to reduce interaction costs.



Parallel Algorithm Models (continued)

- Master-Slave Model: One or more processes generate work and allocate it to worker processes. This allocation may be static or dynamic.
- Pipeline / Producer-Consumer Model: A stream of data is passed through a succession of processes, each of which perform some task on it.
- Hybrid Models: A hybrid model may be composed either of multiple models applied hierarchically or multiple models applied sequentially to different phases of a parallel algorithm.



INTRODUCTION TO MPI

Dany Vohl

Astronomy Data and Computing Services,
Swinburne



*Based on contents developed by
Amr H. Hassan*



TABLE OF CONTENT

- Introduction
 - Notions of parallel computing
 - *Shared Memory versus Distributed Memory*
 - Message Passing Interface
 - Working examples with mpi4py
 - Point-to-Point Communication
 - *Sum two vectors*
 - Collective Communication Patterns
 - *Computing Pi*



DISCLAIMER

- ▶ We chose Python over C/C++ for simplicity and to avoid the hassle of compilations and Linking.
- ▶ The code in some cases is not optimized or follow best practices.
 - ▶ *The important point is to illustrate the concept.*
- ▶ Most of these tools are actually designed and developed for C/C++.
 - ▶ *Their python version might not be complete.*

ASTRONOMY DATA AND COMPUTING SERVICES





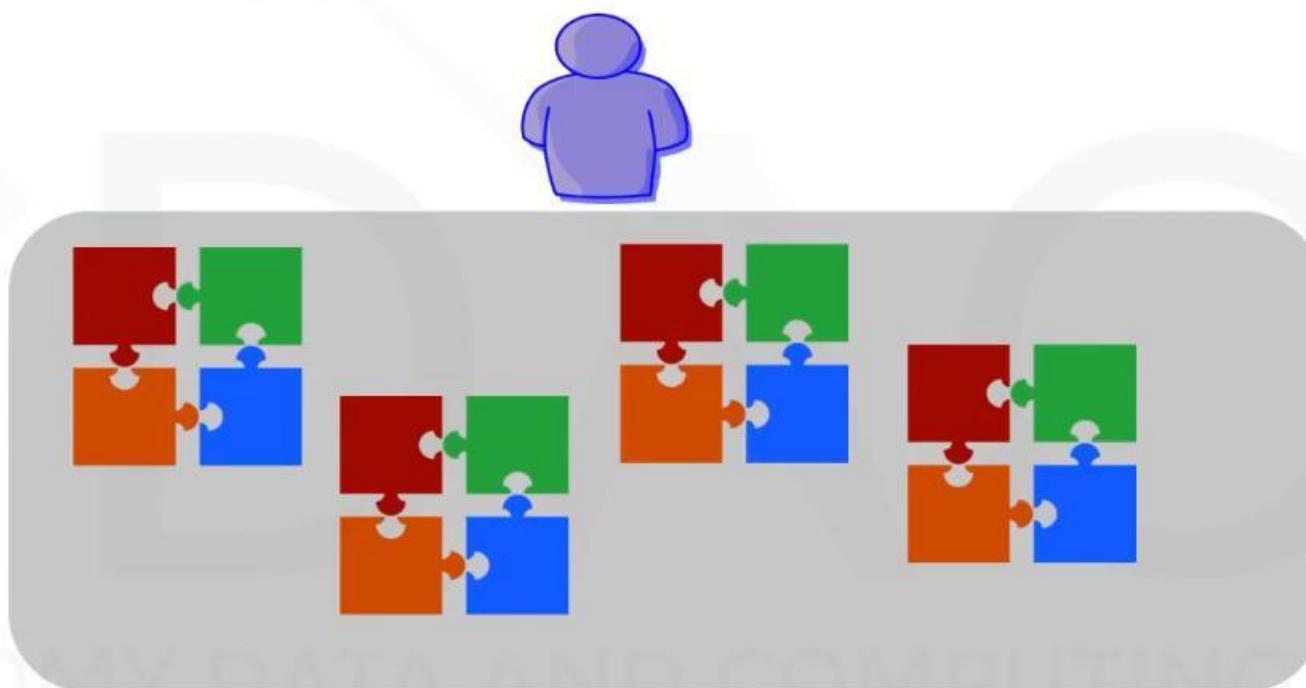
INTRODUCTION

*Shared Memory VS
Distributed Memory*



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

SHARED MEMORY

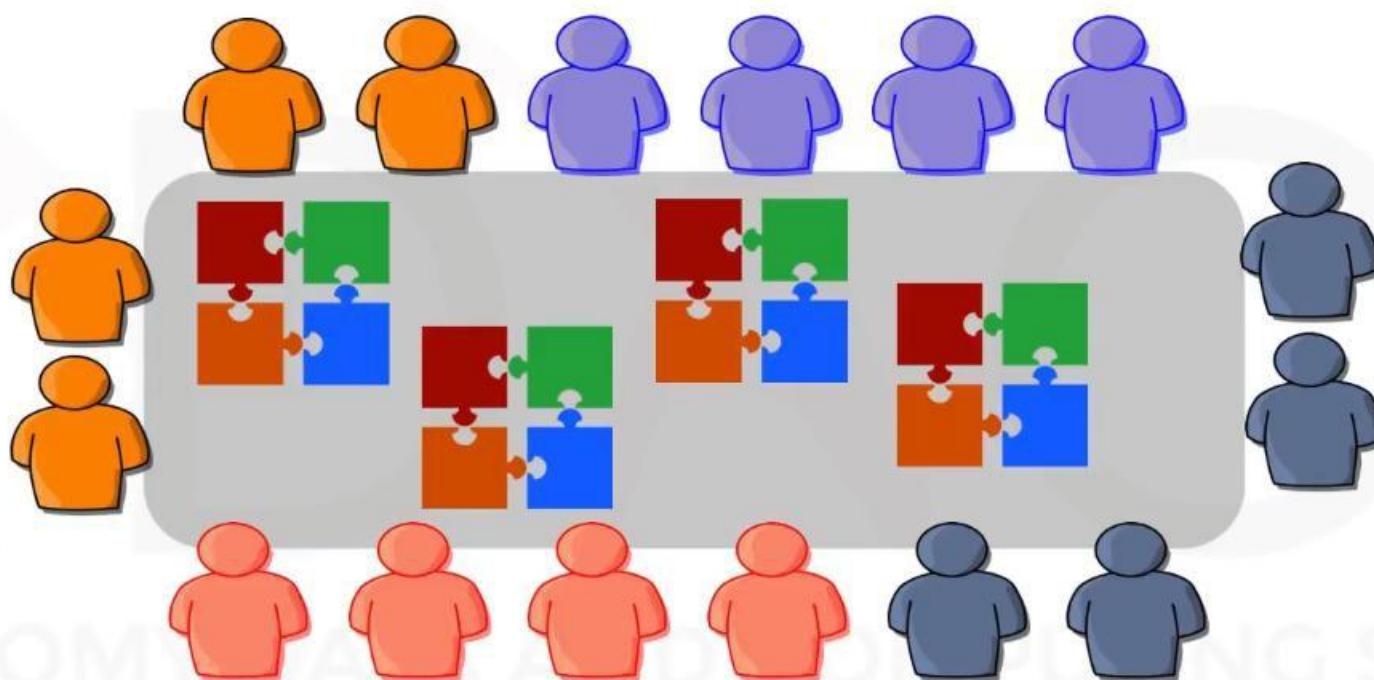


Modified From: Hanjun Kim, Princeton University



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

SHARED MEMORY



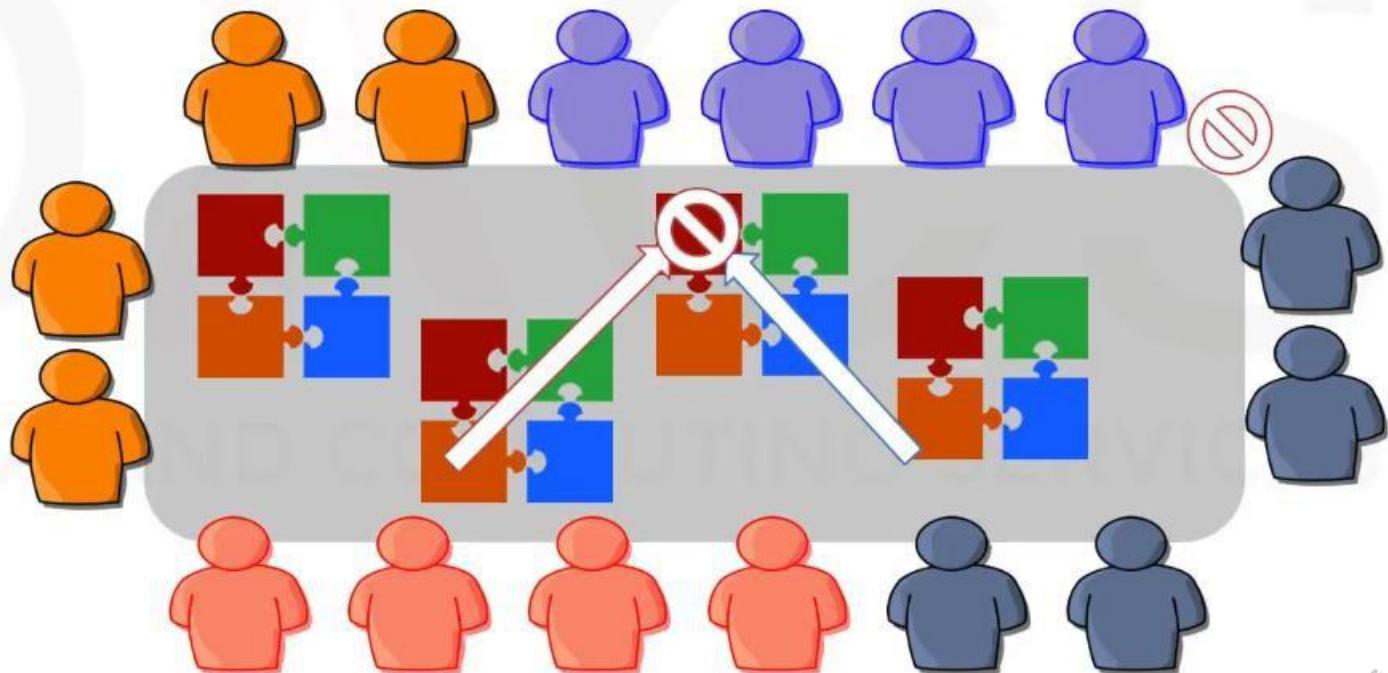
Modified From: Hanjun Kim, Princeton University



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

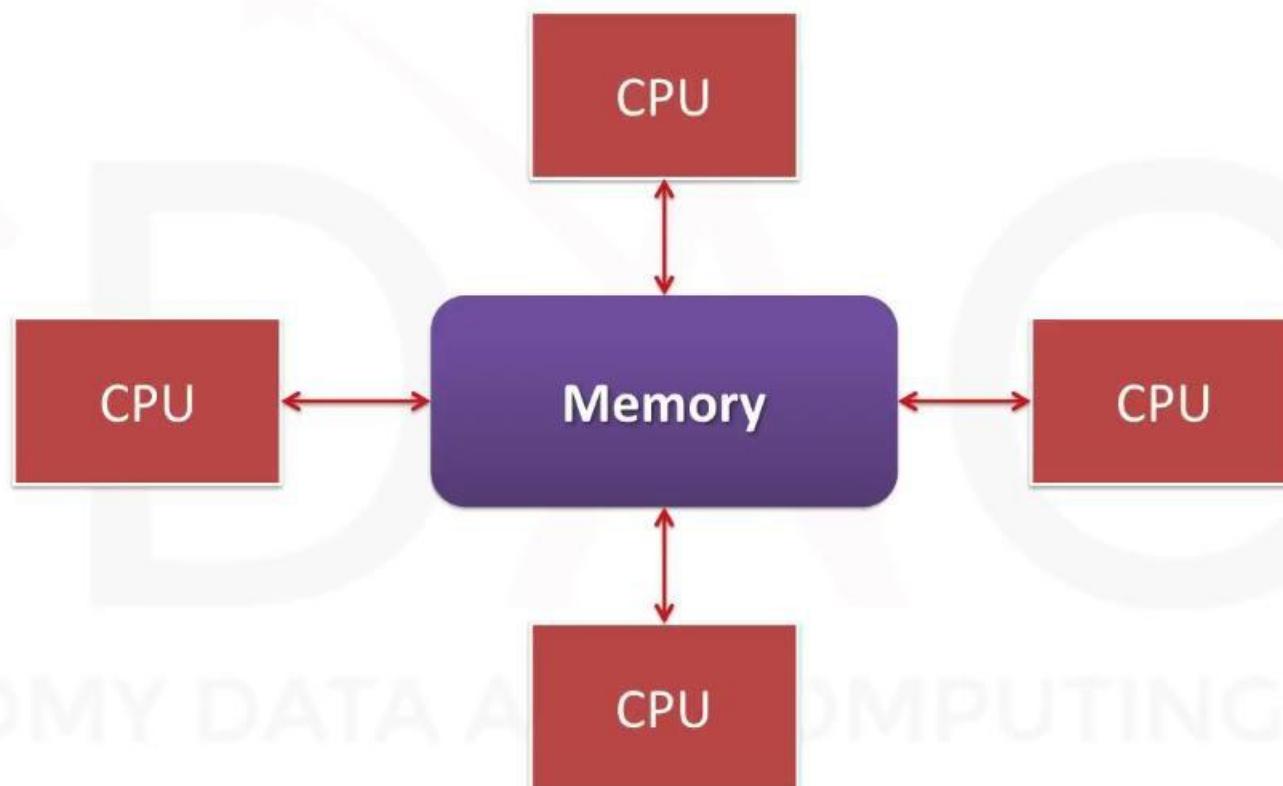
SHARED MEMORY

- ▶ Splitting the problem among many people in shared memory
 - ▶ Pros - Easy to use
 - ▶ Cons - Limited Scalability, High coherence overhead



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

SHARED MEMORY

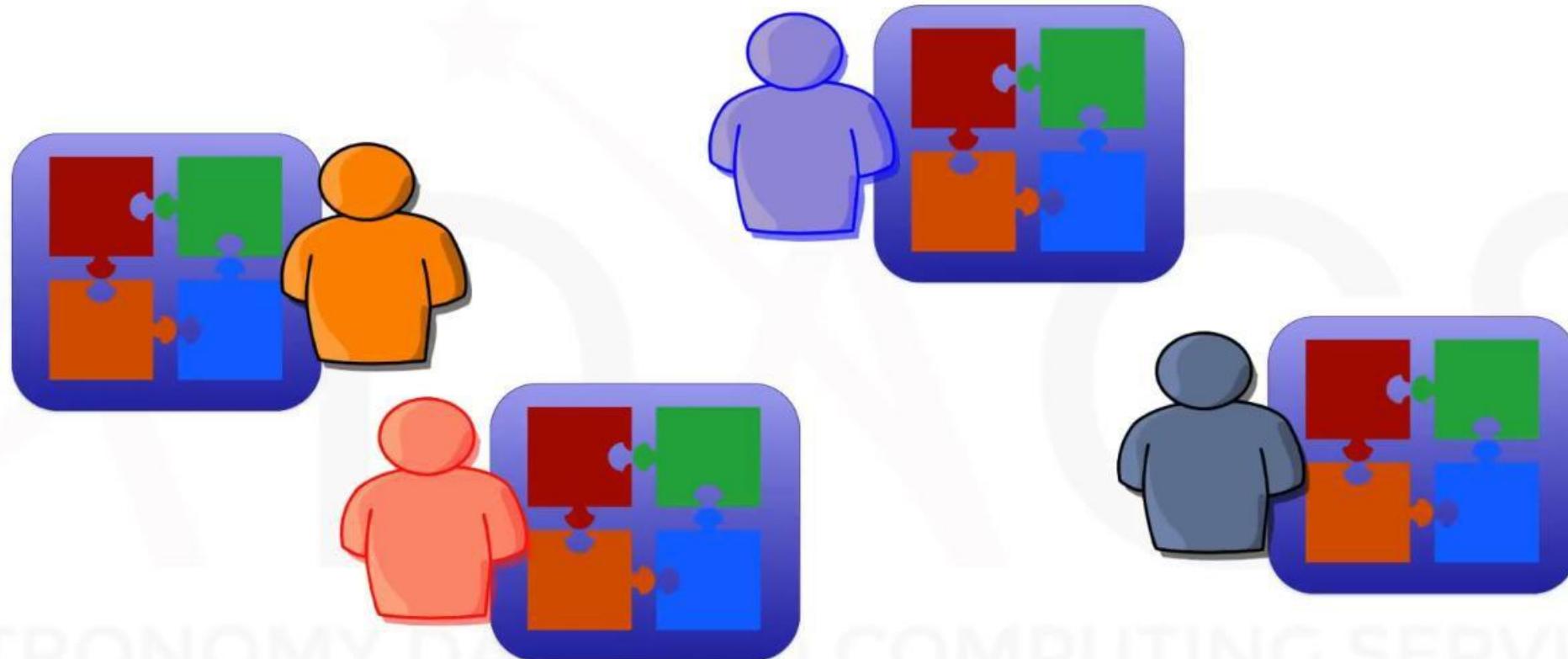


ASTRONOMY DATA AND COMPUTING SERVICES



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

DISTRIBUTED MEMORY



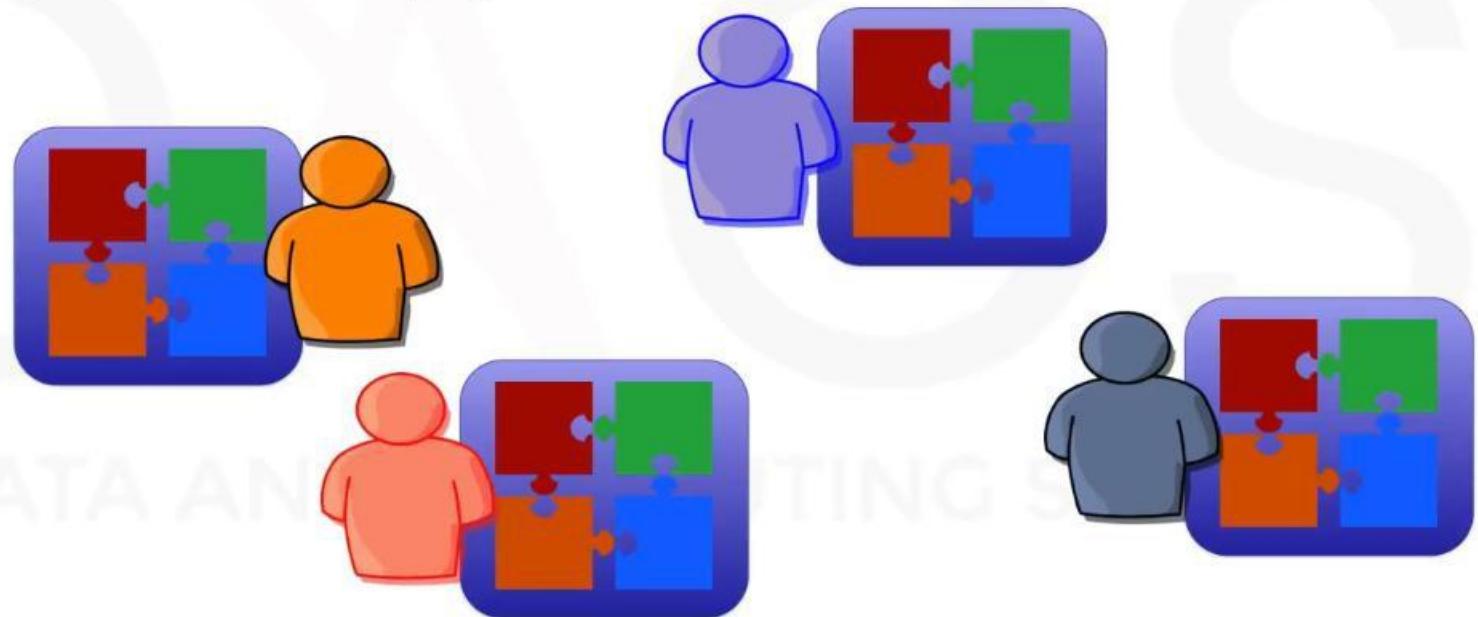
Modified From: Hanjun Kim, Princeton University



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

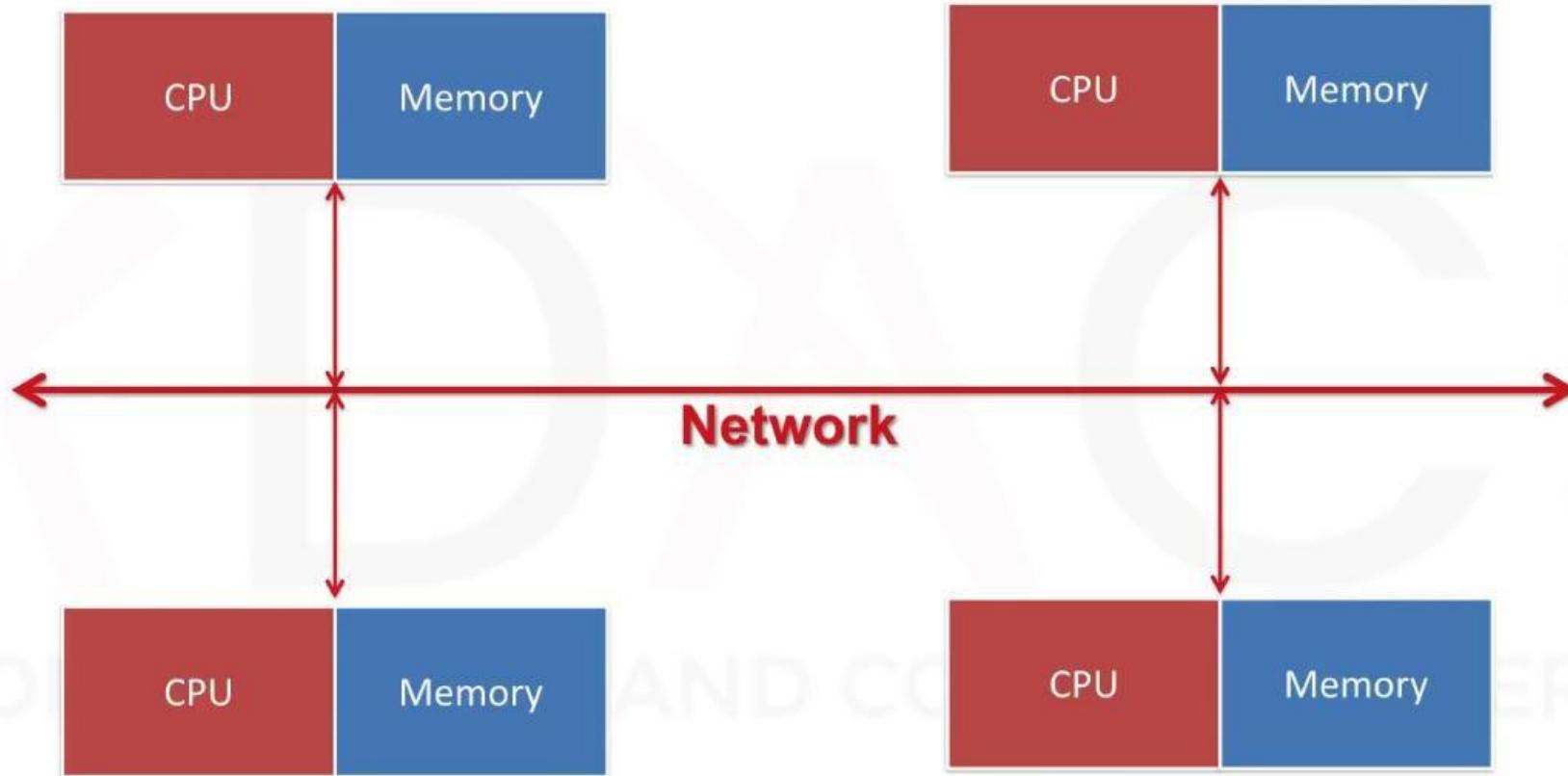
DISTRIBUTED MEMORY

- ▶ Splitting the problem among many people distributed memory
 - ▶ Scalable seats (*Scalable resources*)
 - ▶ Less contention from *private memory spaces*



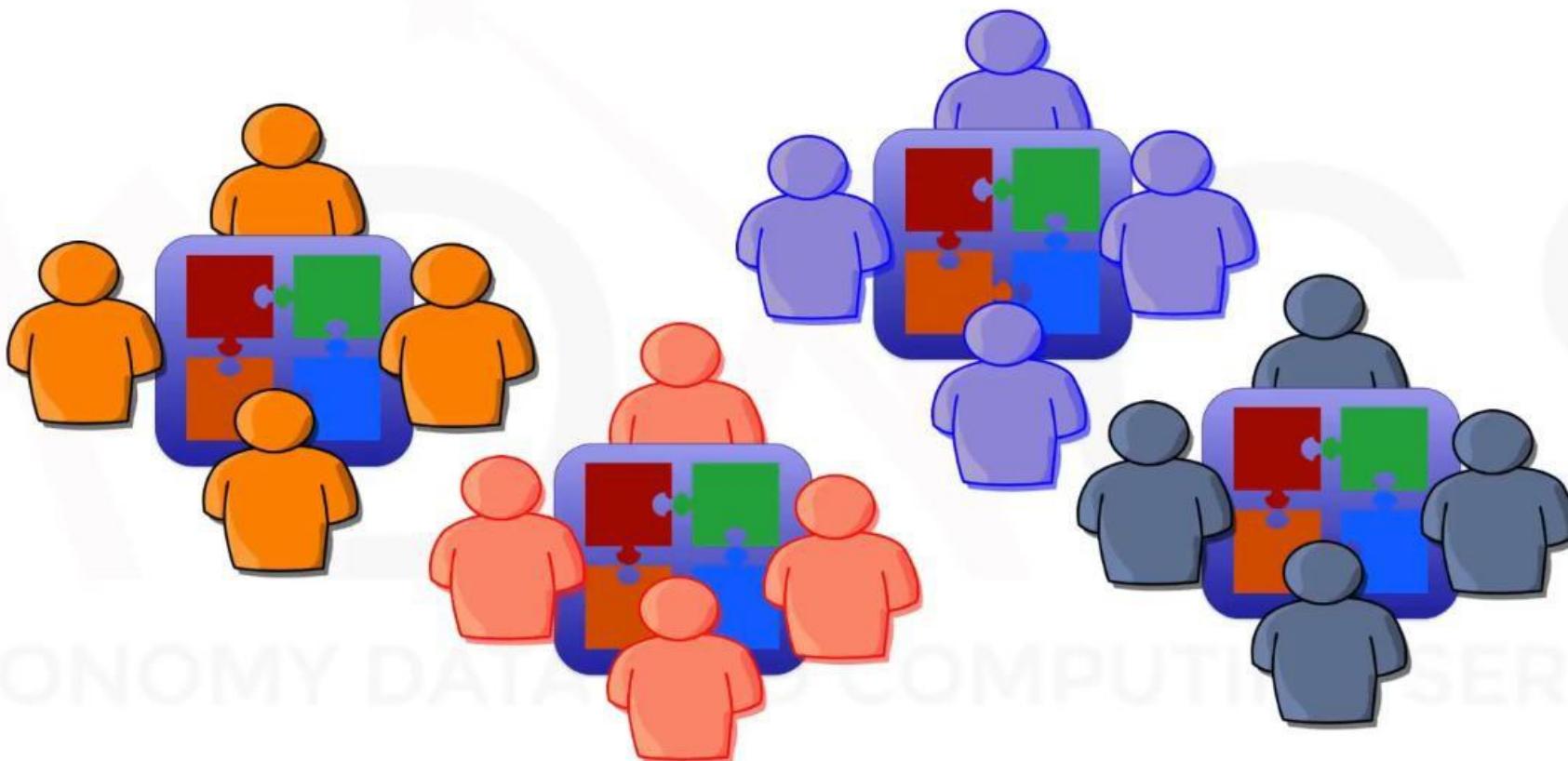
SHARED MEMORY VERSUS DISTRIBUTED MEMORY

DISTRIBUTED MEMORY



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

MIXED VERSION

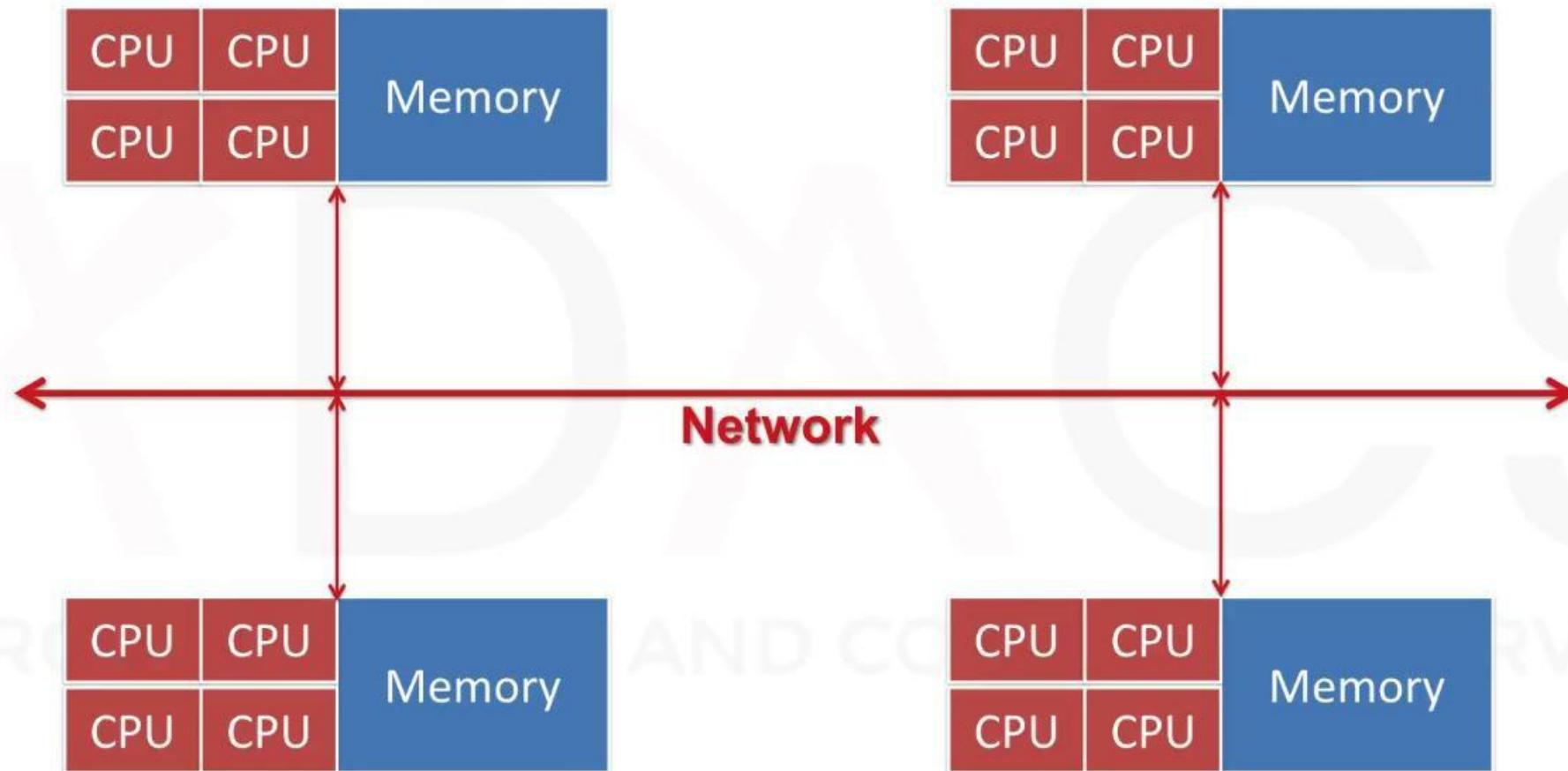


Modified From: Hanjun Kim, Princeton University



SHARED MEMORY VERSUS DISTRIBUTED MEMORY

MIXED VERSION





INTRODUCTION

Message Passing Interface



WHAT IS MPI, AND WHY USE IT?

"The Message Passing Interface Standard (MPI) is a message passing library standard based on the consensus of the MPI Forum, which has over 40 participating organizations, including vendors, researchers, software library developers, and users. **The goal of [MPI] is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs.** (...) The advantages of developing message passing software using MPI closely match the design goals of portability, efficiency, and flexibility. **MPI is not an IEEE or ISO standard, but has in fact, become the "industry standard" for writing message passing programs on HPC platforms.**"

– <https://computing.llnl.gov/tutorials/mpi/>



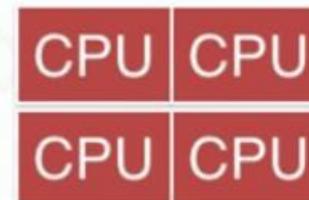
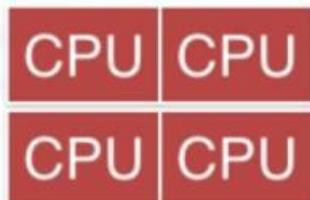
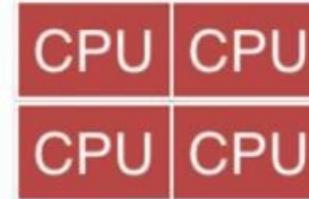
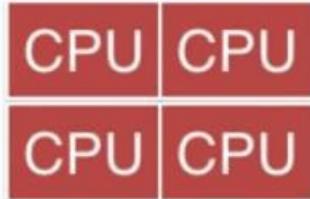
WHAT IS MPI, AND WHY USE IT?

- De facto Standard Framework for Distributed computing.
- Simple communication model between processes in a program.
- Multiple implementations; highly efficient for different platforms.
- Well established community (since 1994).

ASTRONOMY DATA AND COMPUTING SERVICES



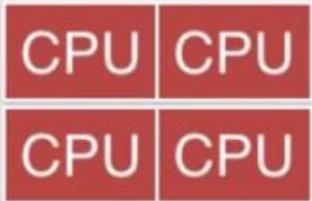
MESSAGE PASSING INTERFACE



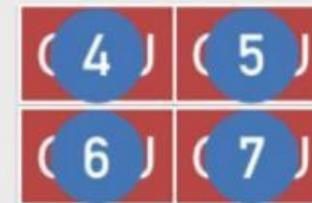
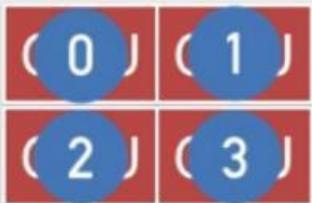
MESSAGE PASSING INTERFACE



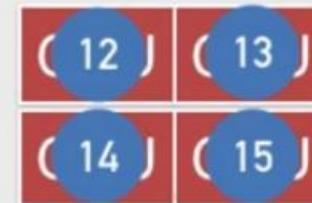
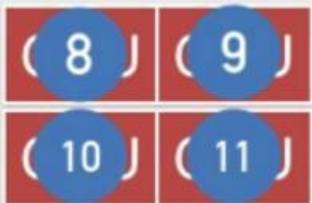
MPI_COMM_WORLD



MESSAGE PASSING INTERFACE

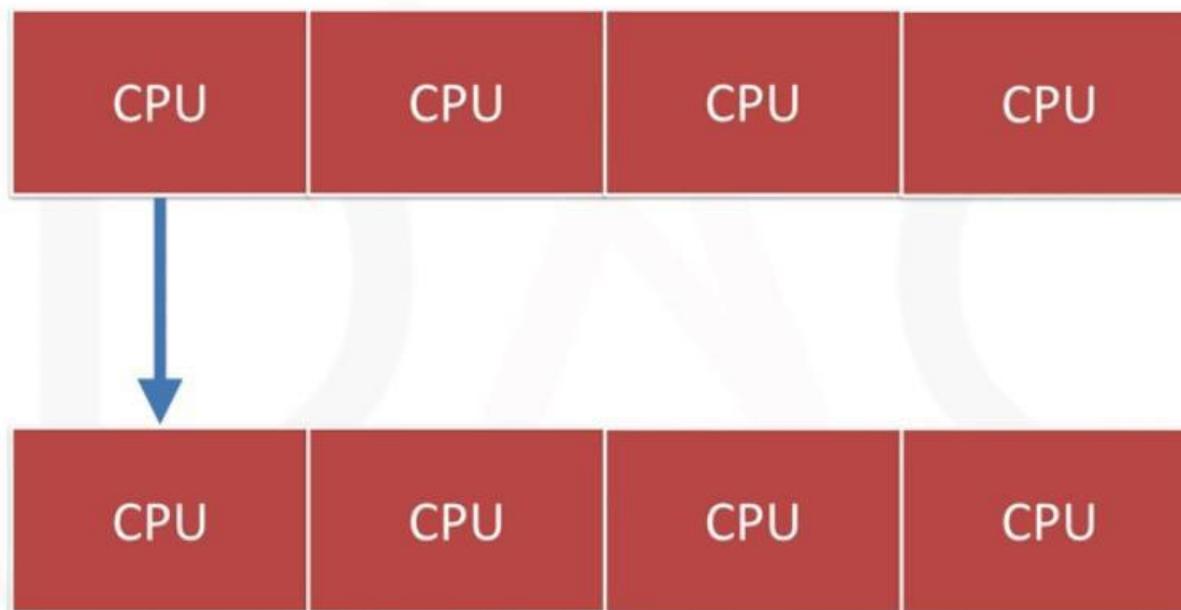


MPI_COMM_WORLD



MESSAGE PASSING INTERFACE

Point-to-Point Communication

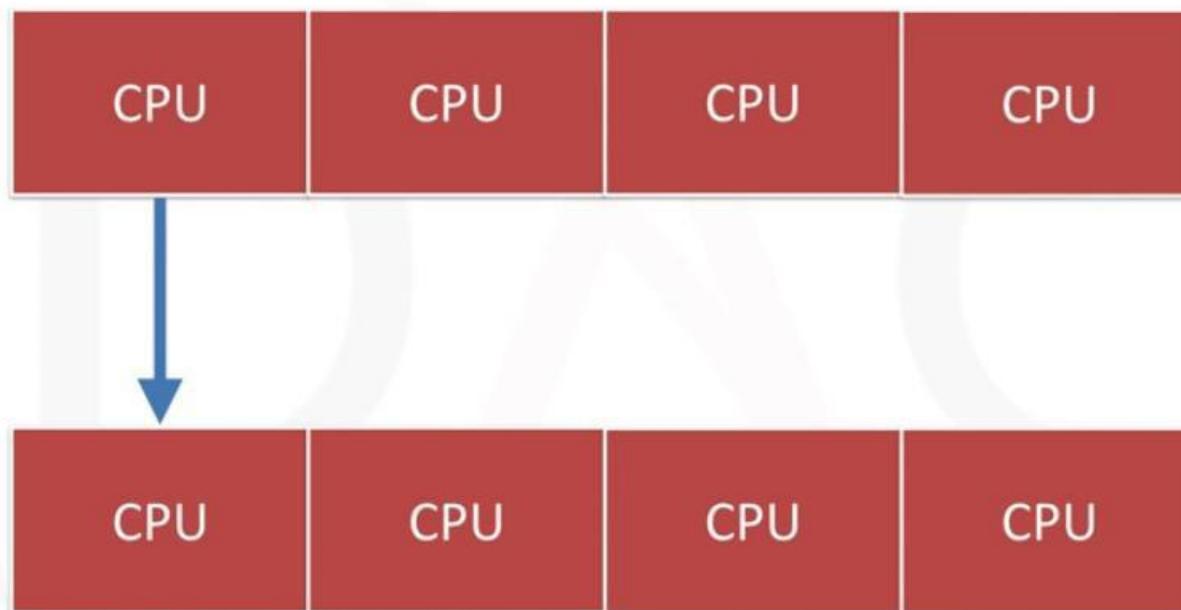


MESSAGE PASSING INTERFACE



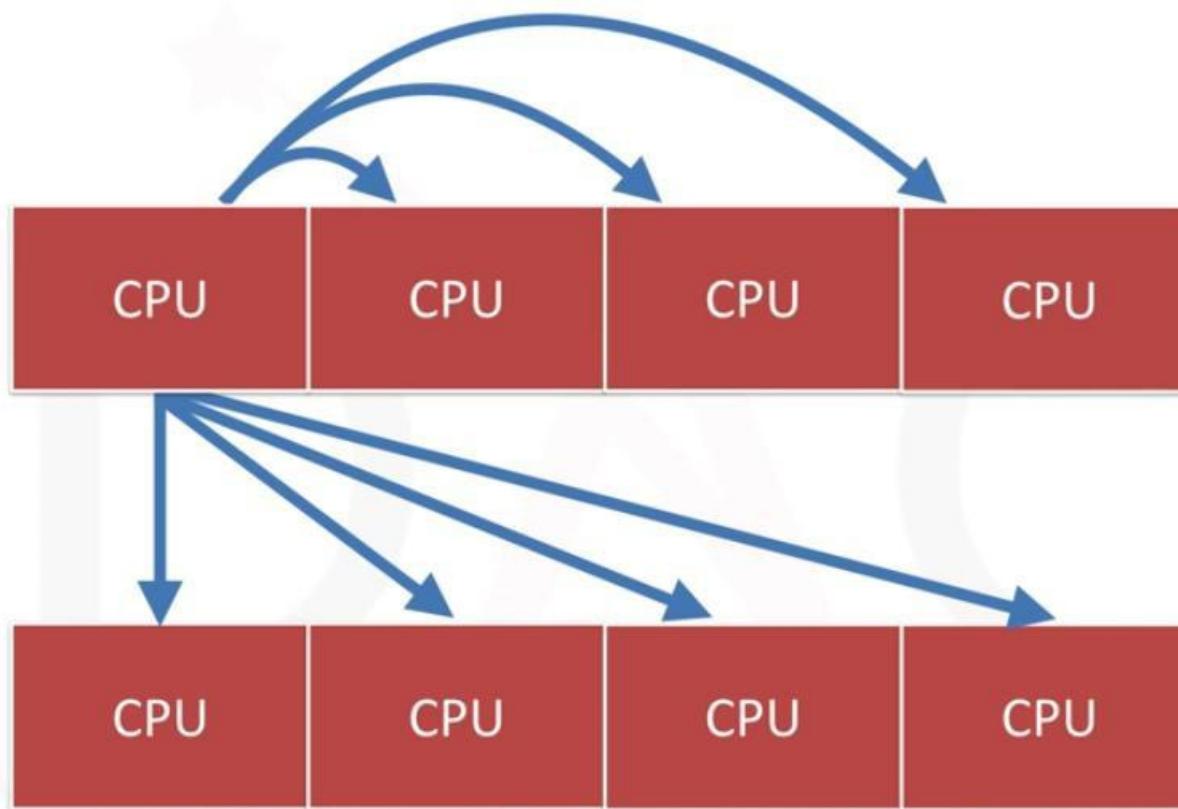
MESSAGE PASSING INTERFACE

Point-to-Point Communication



MESSAGE PASSING INTERFACE

Collective Communication





WORKING EXAMPLES WITH MPI4PY

Message Passing Interface

ASTRONOMY DATA AND COMPUTATION SERVICES



MPI FOR PYTHON

- Python package : mpi4py
- mpi4py supports:
 1. Pickle-based communication of generic Python object
 - For convenience
 2. Direct array data communication of buffer-provider objects
 - Faster option (near C-speed)
 - (e.g., NumPy arrays).

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON

- Important Class "**Comm**"

1. Communication for Generic python objects

- Use "lower case" methods: **send()**, **receive()**

2. Communication for buffer-provider objects (e.g numpy arrays)

- Use "upper case" methods: **Send()**, **Receive()**

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON

- ▶ Importing library:

```
from mpi4py import MPI
```

- ▶ Getting important information:

```
comm = MPI.COMM_WORLD
```

```
rank = MPI.COMM_WORLD.Get_rank()
```

```
size = MPI.COMM_WORLD.Get_size()
```

```
name = MPI.Get_processor_name()
```



MPI

VersionControlTutorial — dvohl@sstar001:~ — ssh dvohl@g2.hpc.swin.edu.au — 82×25

```
%quickref -> Quick reference.  
help      -> Python's own help system.  
object?   -> Details about 'object', use 'object??' for extra details.
```

In [1]: from mpi4py import MPI

In [2]: comm = MPI.COMM_WORLD
MPI.COMBINER_CONTIGUOUS MPI.COMBINER_STRUCT
MPI.COMBINER_DARRAY MPI.COMBINER_STRUCT_INTEGER
MPI.COMBINER_DUP MPI.COMBINER_SUBARRAY
MPI.COMBINER_F90_COMPLEX MPI.COMBINER_VECTOR
MPI.COMBINER_F90_INTEGER MPI.COMM_NULL
MPI.COMBINER_F90_REAL MPI.COMM_SELF
MPI.COMBINER_HINDEXED MPI.COMM_WORLD
MPI.COMBINER_HINDEXED_INTEGER MPI.COMPLEX
MPI.COMBINER_HVECTOR MPI.COMPLEX16
MPI.COMBINER_HVECTOR_INTEGER MPI.COMPLEX32
MPI.COMBINER_INDEXED MPI.COMPLEX4
MPI.COMBINER_INDEXED_BLOCK MPI.COMPLEX8
MPI.COMBINER_NAMED MPI.CONGRUENT
MPI.COMBINER_RESIZED

In [2]: comm = MPI.COMM_WORLD

In [3]:



MPI FOR PYTHON

HELLO, WORLD

```
from mpi4py import MPI
comm=MPI.COMM_WORLD
print ("Hello! I'm rank %d out of %d processes running in
total ..."%(comm.rank,comm.size))
comm.Barrier()
```

```
>> mpirun -np 4 python hello-world.py
```

ASTRONOMY DATA AND COMPUTING SERVICES



MPI

fre

CO

pr

to

CO

>>

VersionControlTutorial — dvoohl@sstar001:~ — ssh dvoohl@g2.hpc.swin.edu.au — 82x25

```
[dvoohl@sstar001 ~]$ nano hello_world.py
[dvoohl@sstar001 ~]$ mpirun -np 4 python hello-world.py
Hello! I'm rand 0 from 4 running in total ...
Hello! I'm rand 1 from 4 running in total ...
Hello! I'm rand 3 from 4 running in total ...
Hello! I'm rand 2 from 4 running in total ...
[dvoohl@sstar001 ~]$
```





*Message Passing Interface
(Point-to-Point Communication)*



MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- ▶ Python objects (`pickle` under the hood)
- ▶ An object to be sent is passed as a parameter to the communication call,
and the received object is simply the return value.

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- ▶ Python objects (pickle under the hood)

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```



MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- ▶ Buffer-provider objects (NumPy arrays; the fast way)
 - ▶ *Buffer arguments to these calls must be explicitly specified*
 - ▶ Use a 2/3-list/tuple
 - ▶ e.g. [data, MPI.DOUBLE]
 - ▶ uses the byte-size of data and the extent of the MPI datatype to define the count
 - ▶ or [data, count, MPI.DOUBLE]

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- ▶ Buffer-provider objects (NumPy arrays; the fast way)

```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

# passing MPI datatypes explicitly
if rank == 0:
    data = numpy.arange(1000, dtype='i')
    comm.Send([data, MPI.INT], dest=1, tag=77)
elif rank == 1:
    data = numpy.empty(1000, dtype='i')
    comm.Recv([data, MPI.INT], source=0, tag=77)

(...)
```



MPI FOR PYTHON SENDING AND RECEIVING DATA (POINT-TO-POINT COMM.)

- ▶ Buffer-provider objects (NumPy arrays; the fast way)

(...)

```
# automatic MPI datatype discovery
if rank == 0:
    data = numpy.arange(100, dtype=numpy.float64)
    comm.Send(data, dest=1, tag=13)
elif rank == 1:
    data = numpy.empty(100, dtype=numpy.float64)
    comm.Recv(data, source=0, tag=13)
```

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON

SUM TWO VECTORS

b_0	b_1	b_2	b_3
+	+	+	+
c_0	c_1	c_2	c_3
↓	↓	↓	↓
a_0	a_1	a_2	a_3



p_0	p_1	p_2	p_3
b	b	b	b
+	+	+	+
c	c	c	c
↓	↓	↓	↓
a	a	a	a

Astro Data Grid Services

ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON

SUM TWO VECTORS

b_0	b_1	b_2	b_3
+	+	+	+

c_0	c_1	c_2	c_3
+	+	+	+



a_0	a_1	a_2	a_3
-----	-----	-----	-----



p_0

b
+

c

a

p_1

b
+

c

a

p_2

b
+

c

a

p_3

b
+

c

a

```
for (i=0; i<n; i++)
    a[i] = b[i] + c[i];
```

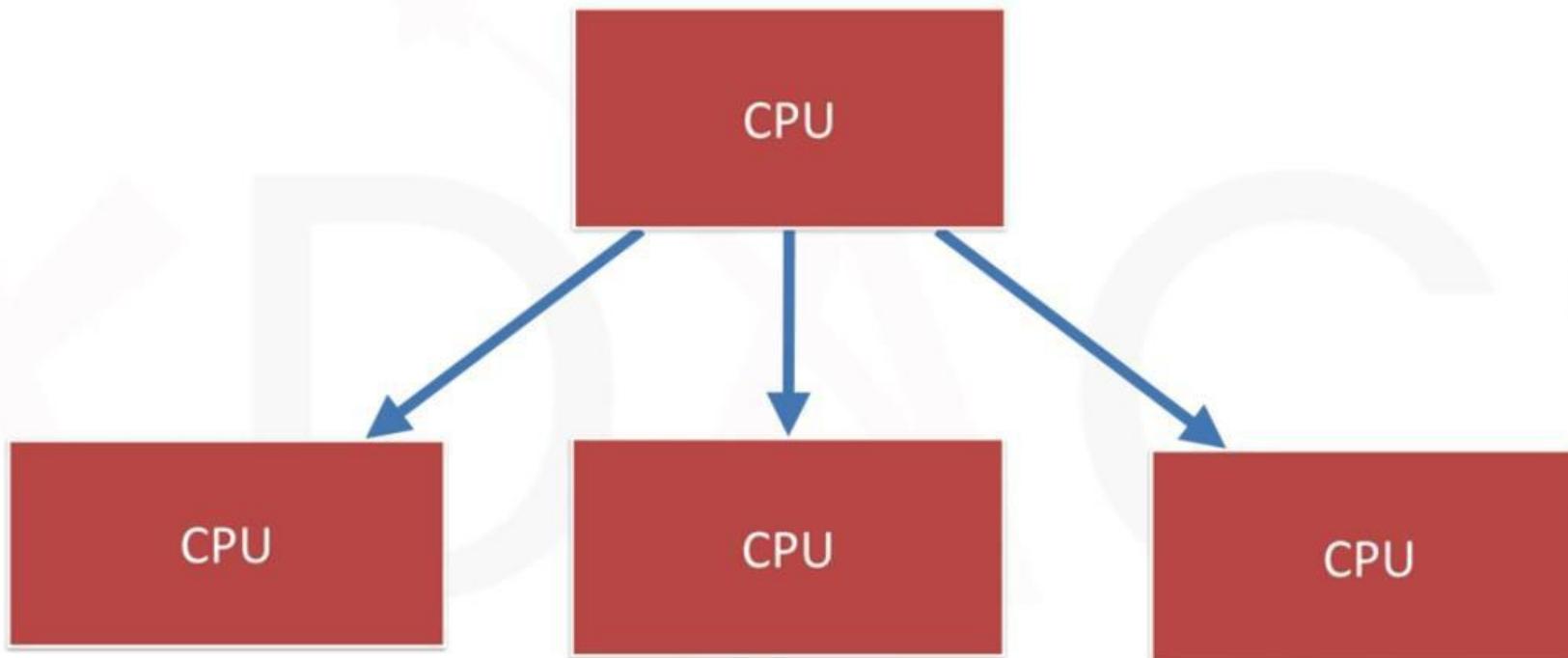


```
for (i=my_low; i<my_high; i++)
    a[i] = b[i] + c[i];
```



MPI FOR PYTHON

SUM TWO VECTORS

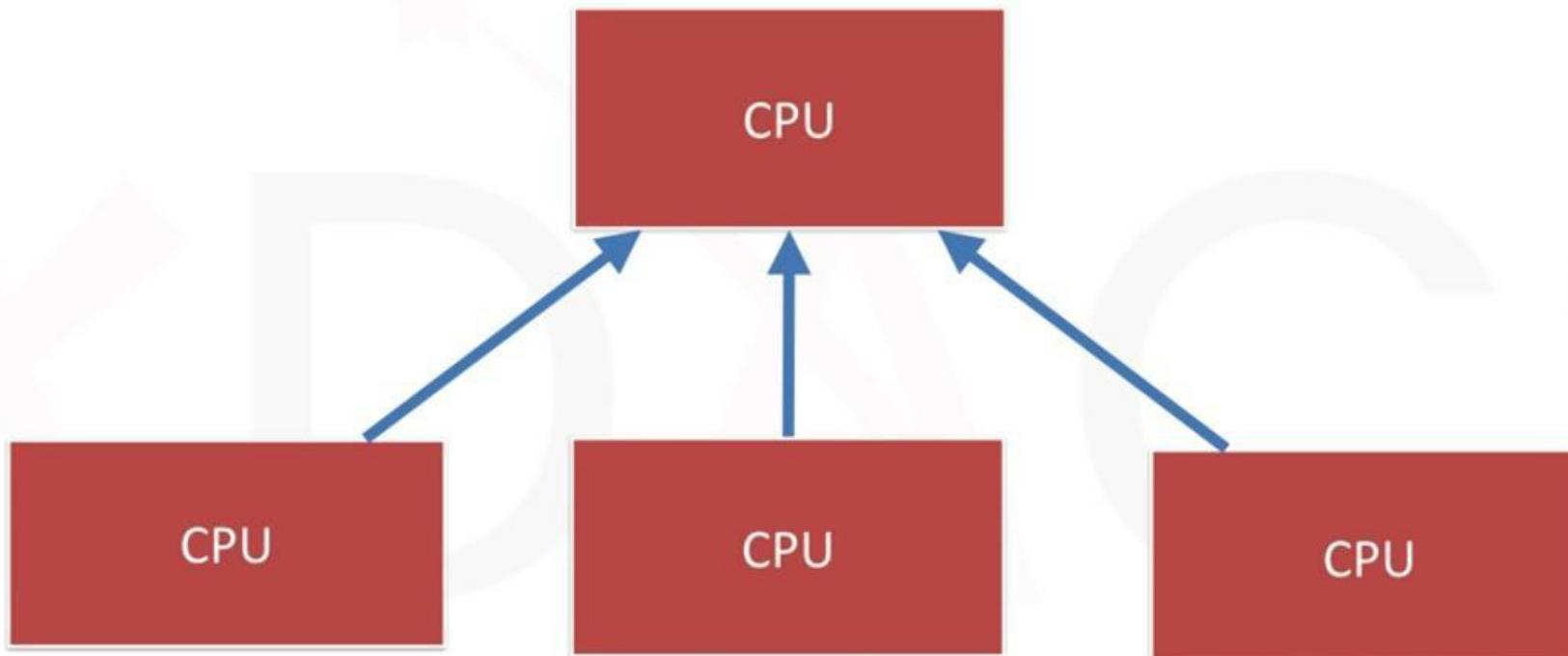


Client-Server Communication



MPI FOR PYTHON

SUM TWO VECTORS



Client-Server Communication



```
from mpi4py import MPI
import numpy

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()
Count = 999
PCount = Count / (size - 1)

(...)
```



MPI FOR PYTHON

SUM TWO VECTORS

```
(...)  
# pass explicit MPI datatypes  
if rank == 0:  
    Adata = numpy.arange(0, Count, 1, dtype='i')  
    Bdata = numpy.arange(Count, 0, -1, dtype='i')  
    Cdata = numpy.empty(Count, dtype='i')  
    for p in range(1, size):  
        Start = PCount * (p - 1)  
        End = PCount * p  
        print ("%d:[%d to %d]" % (p, Start, End))  
  
        Dim = [Start, End]  
        comm.send(Dim, dest=p, tag=1)  
        comm.Send(Adata[Start:End], dest=p, tag=2)  
        comm.Send(Bdata[Start:End], dest=p, tag=2)  
  
for p in range(1, size):  
    Start = PCount * (p - 1)  
    End = PCount * p  
    comm.Recv(Cdata[Start:End], source=p, tag=3)  
    print (Cdata)  
(...)
```



```
(...)  
else:  
  
    Dim = comm.recv(source=0, tag=1)  
    print ("Rank %d: Received[%d to %d]" % (rank, Dim[0], Dim[1]))  
  
    Adata = numpy.empty(Dim[1] - Dim[0], dtype='i')  
    Bdata = numpy.empty(Dim[1] - Dim[0], dtype='i')  
  
    comm.Recv(Adata, source=0, tag=2)  
    comm.Recv(Bdata, source=0, tag=2)  
  
    Cdata = numpy.add(Adata, Bdata)  
  
    comm.Send(Cdata, dest=0, tag=3)
```



MPI

 dvohl — dvohl@sstar001:~ — ssh dvohl@g2.hpc.swin.edu.au — 82×25



ડોબ્લી — dvoohl@ssstar001:~ — ssh dvoohl@a2.hpc.swin.edu.au — 82x25

MPI [dvohl@sstar001 ~]\$ clear

```
status=MPI.Status()
data=comm.recv(source=MPI.ANY_SOURCE,
               tag=MPI.ANY_TAG,
               status=status)
source=status.Get_source()
tag=status.Get_tag()
```





*Message Passing Interface
(Collective Communication)*

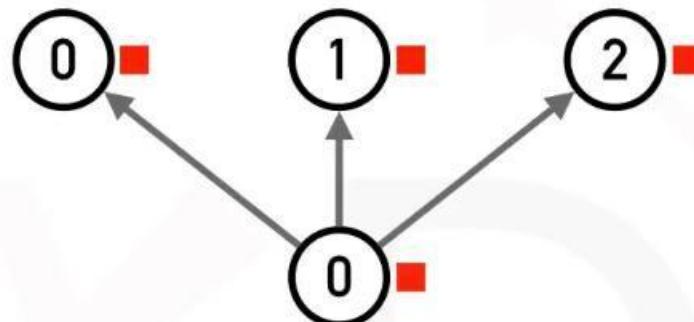


MPI FOR PYTHON

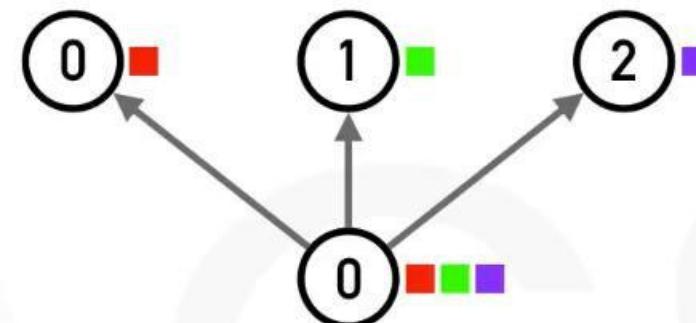
SENDING AND RECEIVING DATA

(COLLECTIVE COMM.)

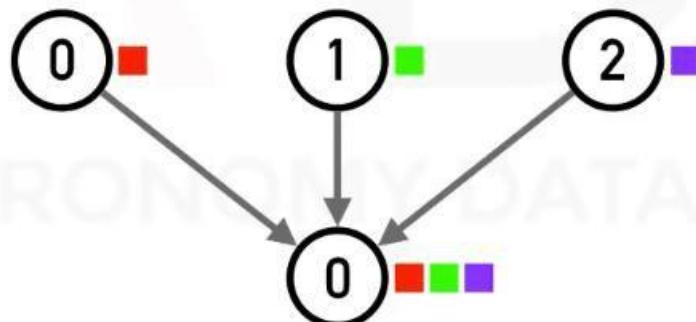
Broadcast



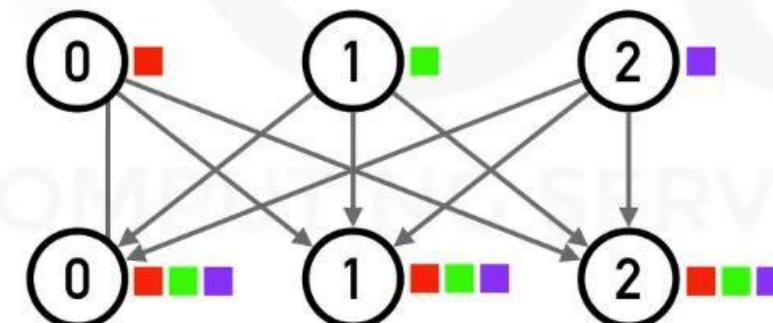
Scatter

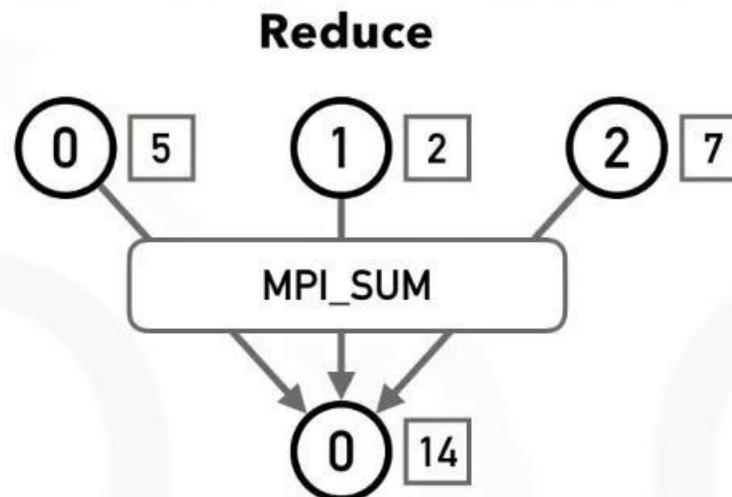


Gather



All Gather





MPI_MAX – Returns the maximum element.

MPI_MIN – Returns the minimum element.

MPI_SUM – Sums the elements.

MPI_PROD – Multiplies all elements.

MPI_LAND – Performs a logical “and” across the elements.

MPI_LOR – Performs a logical “or” across the elements.

MPI_MAXLOC – the maximum value and the rank of the process that owns it.

MPI_MINLOC – the minimum value and the rank of the process that owns it.



MPI FOR PYTHON

COMPUTING π

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx = \sum_{i=0}^N \frac{4.0}{(1+x^2)} \Delta x$$

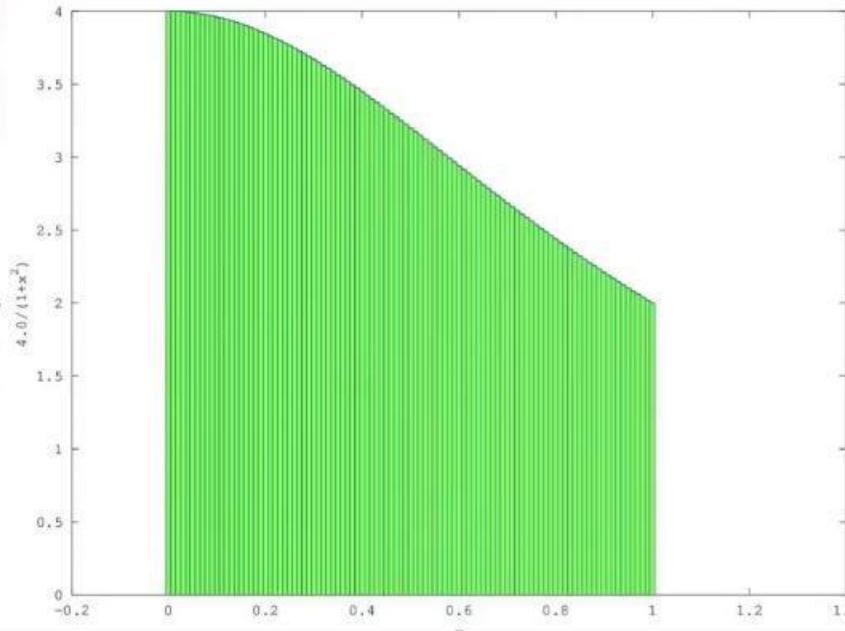
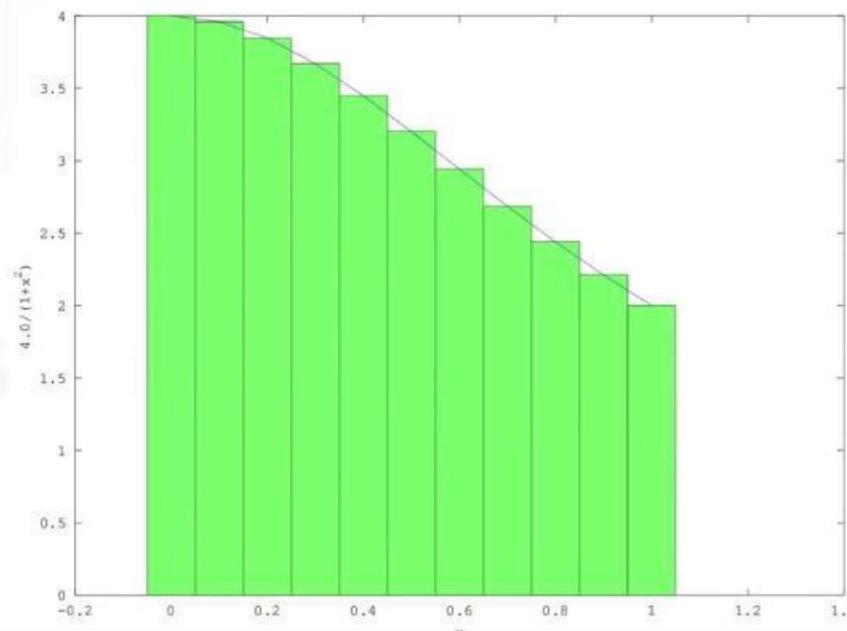
ASTRONOMY DATA AND COMPUTING SERVICES



MPI FOR PYTHON

COMPUTING π

$$\pi = \int_0^1 \frac{4.0}{(1+x^2)} dx = \sum_{i=0}^N \frac{4.0}{(1+x^2)} \Delta x$$



```
import time
def Pi(num_steps):
    start = time.time()
    step = 1.0/num_steps
    sum = 0
    for i in xrange(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
    pi = step * sum
    end =time.time()
    print ("Pi with %d steps is %f in %f secs" %(num_steps, pi, end-start))

if __name__ == '__main__':
    Pi(100000000)
```



MP

dvohl — dvohl@sstar001:~ — ssh dvohl@g2.hpc.swin.edu.au — 80x24

GNU nano 2.0.9

File: pi_serial.py

Modified

```
.....import time
def Pi(num_steps):
    start = time.time()
    step = 1.0/num_steps
    sum = 0
    for i in xrange(num_steps):
        x= (i+0.5)*step
        sum = sum + 4.0/(1.0+x*x)
    pi = step * sum
    end =time.time()
    print ("Pi with %d steps is %f in %f secs" %(num_steps, pi, end-start))

if __name__ == '__main__':
    Pi(100000000)
```

File Name to Write: pi_serial.py

^G Get Help

^C Cancel

^T To Files

M-D DOS Format

M-M Mac Format

M-A Append

M-P Prepend

M-B Backup File



dvoohl — dvoohl@sstar001:~ — ssh dvoohl@g2.hpc.swin.edu.au — 80x24

```
[dvoohl@sstar001 ~]$ nano pi_serial.py
[dvohl@sstar001 ~]$ python pi_serial.py
Pi with 100000000 steps is 3.141593 in 30.438623 secs
[dvohl@sstar001 ~]$
```



```
def Pi(num_steps):
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    Proc_num_steps = num_steps / size
    Start = rank * Proc_num_steps
    End = (rank + 1) * Proc_num_steps
    print ("%d: [%d,%d]" % (rank, Start, End))

    step = 1.0 / num_steps
    sum = 0
    for i in xrange(Start, End):
        x = (i + 0.5) * step
        sum = sum + 4.0 / (1.0 + x * x)

    return sum

(...)
```



(...)

```
if __name__ == '__main__':
    num_steps = 100000000
    start = time.time()
    localsum = Pi(num_steps)

    localpi = localsum / num_steps
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    pi = comm.reduce(localpi, op=MPI.SUM, root=0)
    end = time.time()
    if rank == 0:
        print ("Pi with %d steps is %f in %f secs" % (num_steps, pi, end - start))
```



MPI

```
[dvoohl@sstar001 ~]$ nano pi_mpi.py
[dvoohl@sstar001 ~]$ mpirun -np 14 python pi_mpi.py
1: [7142857,14285714]
5: [35714285,42857142]
(... ) 6: [42857142,49999999]
3: [21428571,28571428]
if __name__ == "__main__":
    n = 100000000
    s = 0
    for i in range(1, n + 1):
        if i % 2 == 0:
            s -= 4 / (i * (i + 1))
        else:
            s += 4 / (i * (i + 1))
    print("Pi with {} steps is {} in {} secs".format(n, s, time.time() - start))
    exit(0)
else:
    print("This script must be run with the command: python pi_mpi.py")
    exit(1)
```





REFERENCES

ASTRONOMY DATA AND COMPUTING SERVICES



REFERENCES

- ▶ <https://computing.llnl.gov/tutorials/mpi/>
- ▶ <https://mpi4py.readthedocs.io/en/stable/>
- ▶ <https://cas-eresearch.github.io/astroinformatics2017/day3.html>

ASTRONOMY DATA AND COMPUTING SERVICES



CUDA C/C++ BASICS

NVIDIA Corporation



What is CUDA?

- CUDA Architecture
 - Expose GPU parallelism for general-purpose computing
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++



Introduction to CUDA C/C++

- What will you learn in this session?
 - Start from “Hello World!”
 - Write and launch CUDA C/C++ kernels
 - Manage GPU memory
 - Manage communication and synchronization



Prerequisites

- You (probably) need experience with C or C++
- You don't need GPU experience
- You don't need parallel programming experience
- You don't need graphics experience



CONCEPTS

Heterogeneous Computing

Blocks

Threads

Indexing

Shared memory

`__syncthreads()`

Asynchronous operation

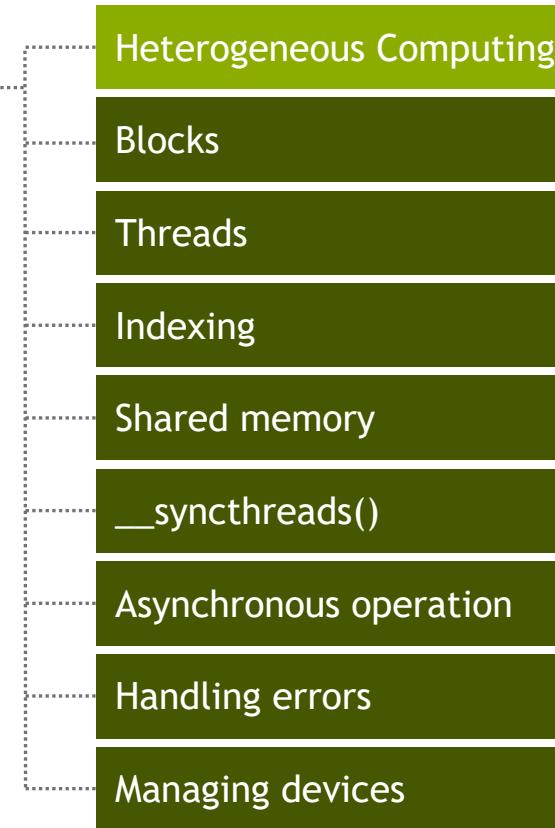
Handling errors

Managing devices



HELLO WORLD!

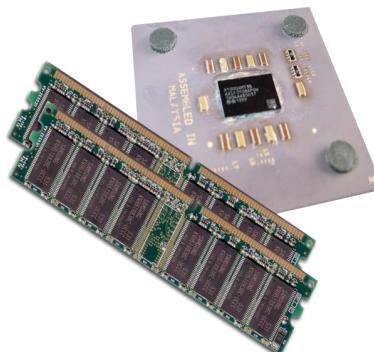
CONCEPTS



Heterogeneous Computing

- Terminology:

- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



Host



Device



Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N 1024
#define RADIUS 3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int index = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[index] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[index - RADIUS] = in[gindex - RADIUS];
        temp[index + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[index + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **) & d_in, size);
    cudaMalloc((void **) & d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<N,BLOCK_SIZE,BLOCK_SIZE>>(d_in + RADIUS,
                                                d_out + RADIUS);

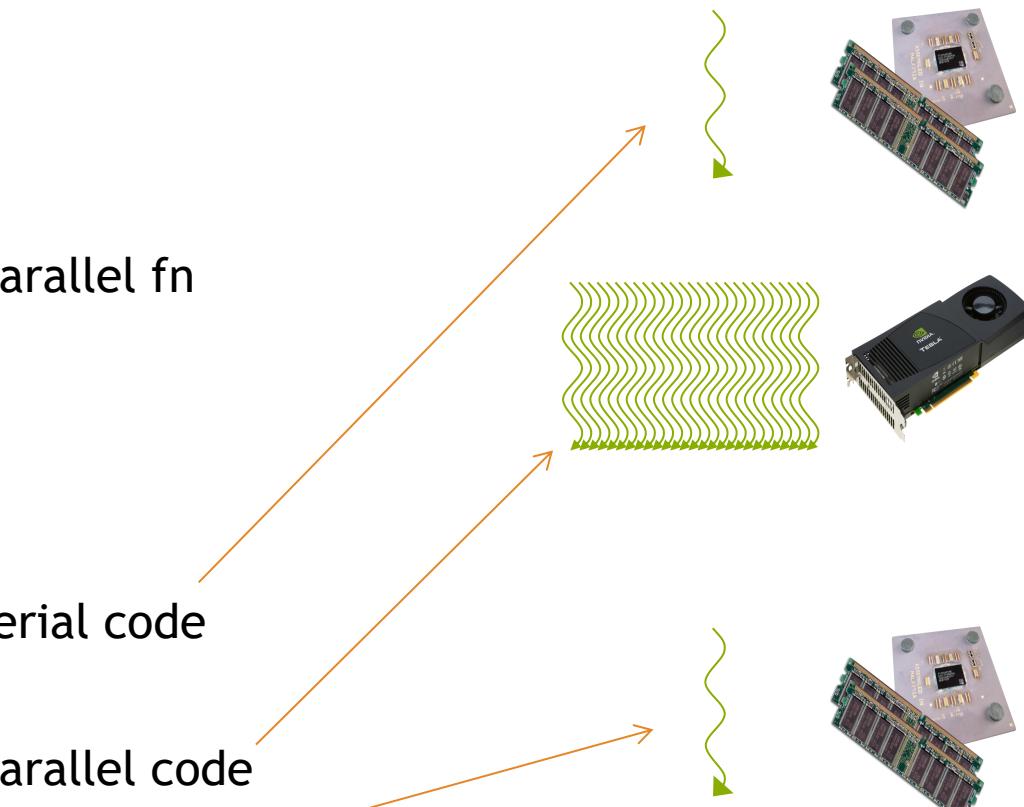
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

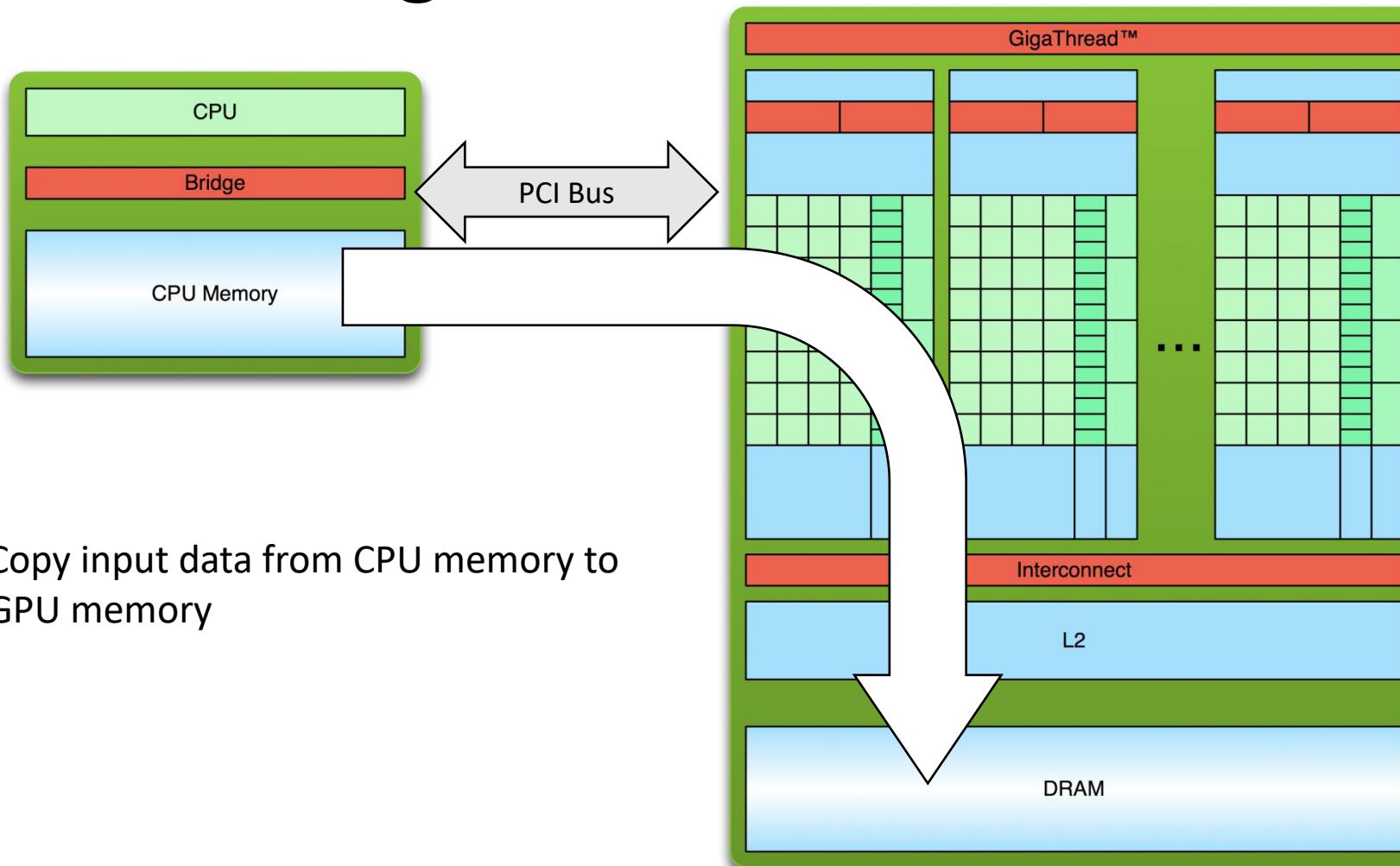
parallel fn

serial code

parallel code
serial code



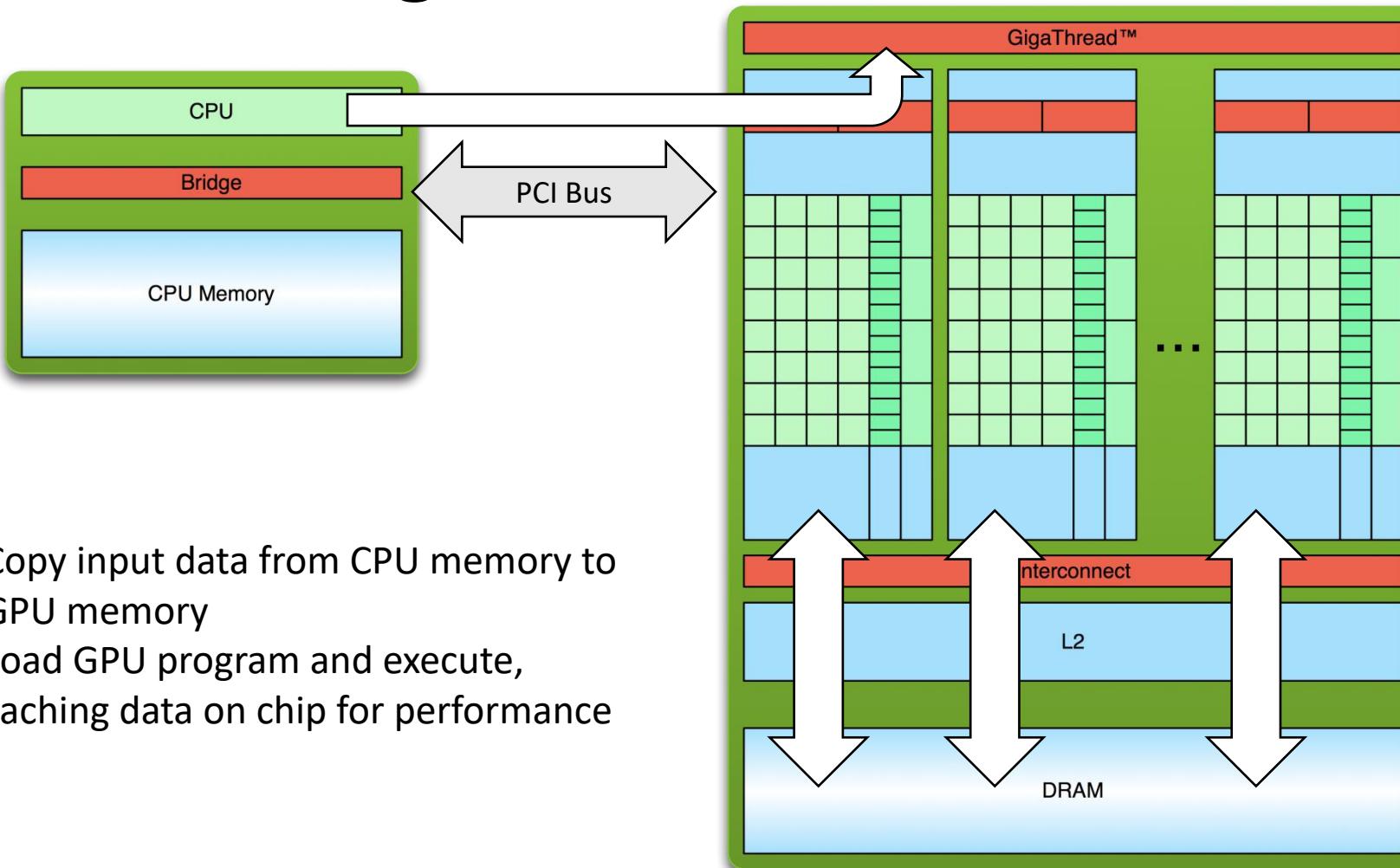
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory



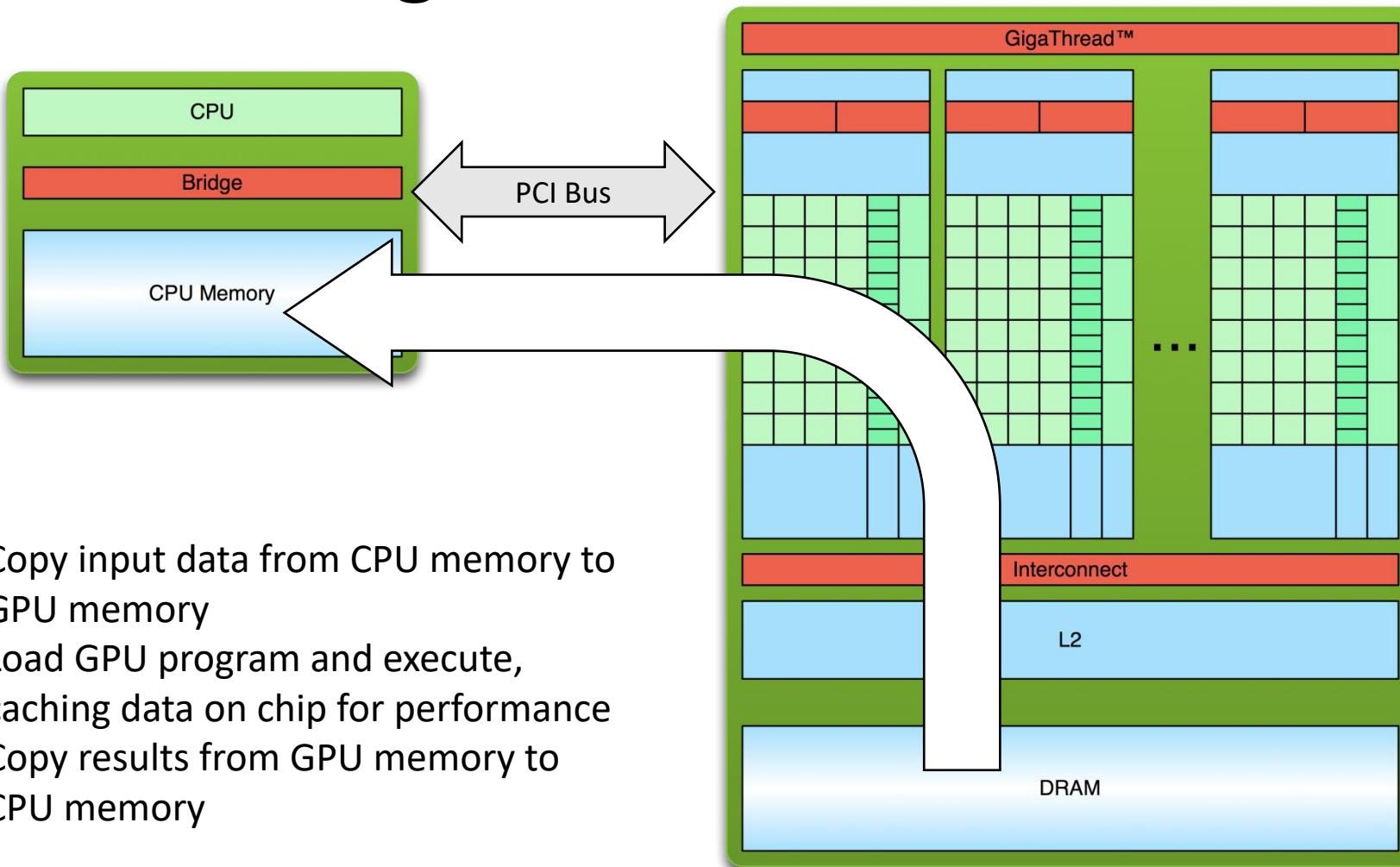
Simple Processing Flow



1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance



Simple Processing Flow



Hello World!

```
int main(void) {  
    printf("Hello World!\n");  
    return 0;  
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc  
hello_world.  
cu  
$ a.out  
Hello World!  
$
```



Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- Two new syntactic elements...



Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc, cl.exe`



Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!



Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

```
int main(void) {  
    mykernel<<<1,1>>>();  
    printf("Hello World!\n");  
    return 0;  
}
```

- **mykernel()** does nothing,
somewhat anticlimactic!

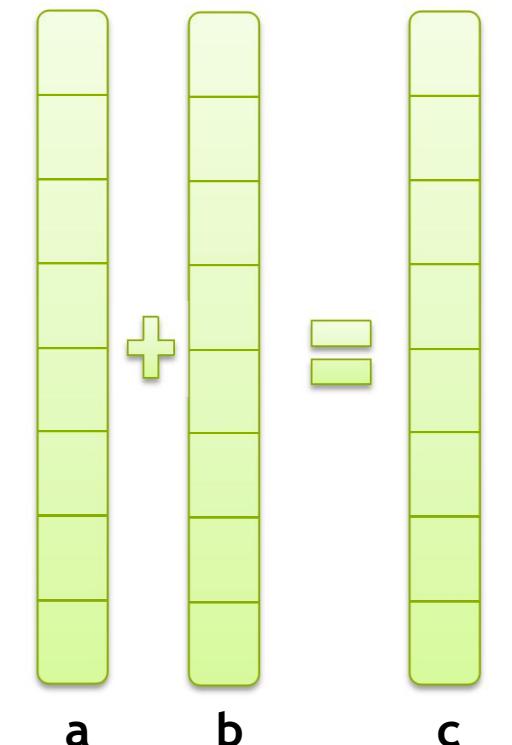
Output:

```
$ nvcc  
hello.cu  
$ a.out  
Hello World!  
$
```



Parallel Programming in CUDA C/C++

- But wait... GPU computing is about massive parallelism!
- We need a more interesting example...
- We'll start by adding two integers and build up to vector addition



Addition on the Device

- A simple kernel to add two integers

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- As before `__global__` is a CUDA C/C++ keyword meaning
 - `add()` will execute on the device
 - `add()` will be called from the host



Addition on the Device

- Note that we use pointers for the variables

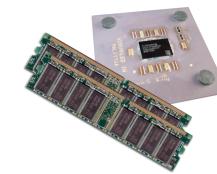
```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- `add()` runs on the device, so `a`, `b` and `c` must point to device memory
- We need to allocate memory on the GPU



Memory Management

- Host and device memory are separate entities
 - *Device* pointers point to GPU memory
 - May be passed to/from host code
 - May *not* be dereferenced in host code
 - *Host* pointers point to CPU memory
 - May be passed to/from device code
 - May *not* be dereferenced in device code
- Simple CUDA API for handling device memory
 - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
 - Similar to the C equivalents `malloc()`, `free()`, `memcpy()`



Addition on the Device: add()

- Returning to our add() kernel

```
__global__ void add(int *a, int *b, int *c) {  
    *c = *a + *b;  
}
```

- Let's take a look at main()...



Addition on the Device: main()

```
int main(void) {
    int a, b, c;                      // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Setup input values
    a = 2;
    b = 7;
```



Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<1,1>>>(d_a, d_b, d_c);

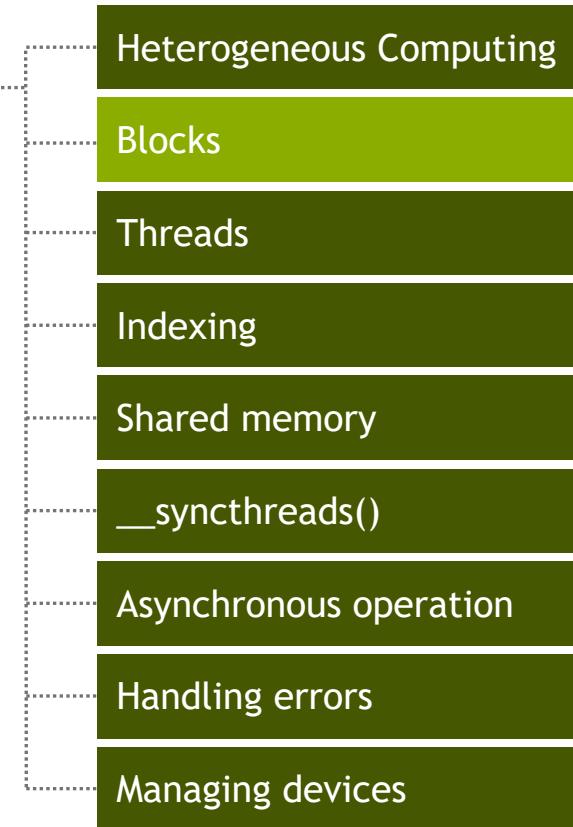
// Copy result back to host
cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



RUNNING IN PARALLEL

CONCEPTS



Moving to Parallel

- GPU computing is about massive parallelism
 - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();  
      ↓  
add<<< N, 1 >>>();
```

- Instead of executing add () once, execute N times in parallel



Vector Addition on the Device

- With `add()` running in parallel we can do vector addition
- Terminology: each parallel invocation of `add()` is referred to as a **block**
 - The set of blocks is referred to as a **grid**
 - Each invocation can refer to its block index using `blockIdx.x`

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- By using `blockIdx.x` to index into the array, each block handles a different index



Vector Addition on the Device

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- On the device, each block can execute in parallel:

Block 0

```
c[0] = a[0] + b[0];
```

Block 1

```
c[1] = a[1] + b[1];
```

Block 2

```
c[2] = a[2] + b[2];
```

Block 3

```
c[3] = a[3] + b[3];
```



Vector Addition on the Device: add()

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {  
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];  
}
```

- Let's take a look at `main()`...



Vector Addition on the Device: main()

```
#define N 512

int main(void) {
    int *a  *b  *c          // host copies of a, b, c
    int *d_a, *d_b, *d_c;   // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```



Vector Addition on the Device: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N blocks
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



Review (1 of 2)

- Difference between *host* and *device*
 - *Host* CPU
 - *Device* GPU
- Using `__global__` to declare a function as device code
 - Executes on the device
 - Called from the host
- Passing parameters from host code to a device function



Review (2 of 2)

- Basic device memory management
 - `cudaMalloc()`
 - `cudaMemcpy()`
 - `cudaFree()`
- Launching parallel kernels
 - Launch `N` copies of `add()` with `add<<<N,1>>>(...);`
 - Use `blockIdx.x` to access block index



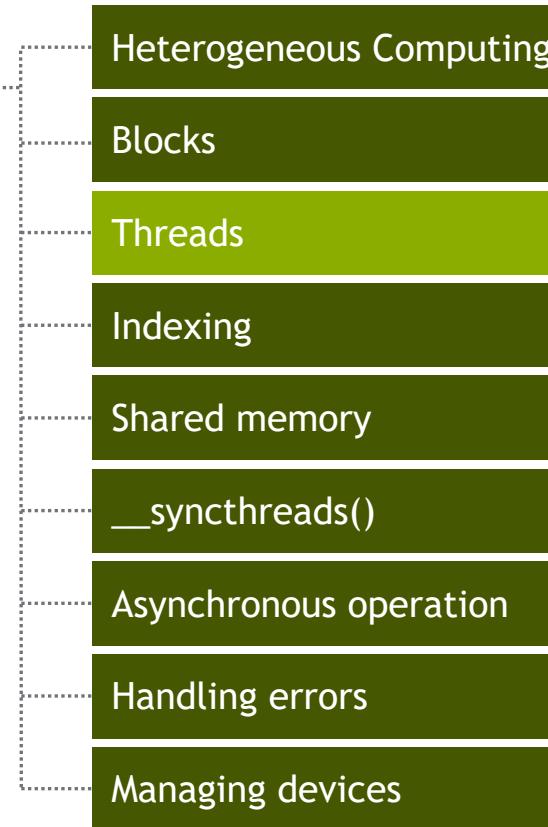
CUDA C/C++ BASICS

NVIDIA Corporation



INTRODUCING THREADS

CONCEPTS



CUDA Threads

- Terminology: a block can be split into parallel **threads**
- Let's change `add()` to use parallel *threads* instead of parallel *blocks*

```
__global__ void add(int *a, int *b, int *c) {  
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];  
}
```

- We use `threadIdx.x` instead of `blockIdx.x`
- Need to make one change in `main()`...



Vector Addition Using Threads: main()

```
#define N 512

int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                          // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```



Vector Addition Using Threads: main()

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU with N threads
add<<<1,N>>>(d_a, d_b, d_c);

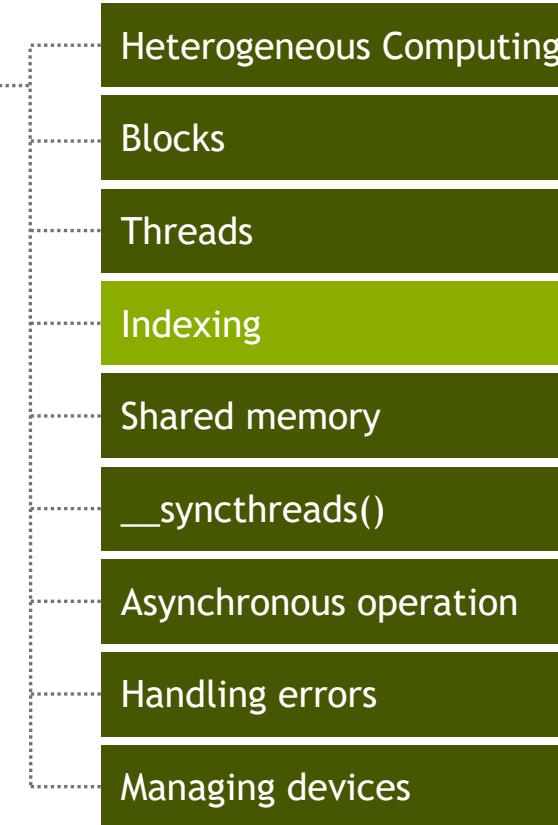
// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



COMBINING THREADS AND BLOCKS

CONCEPTS



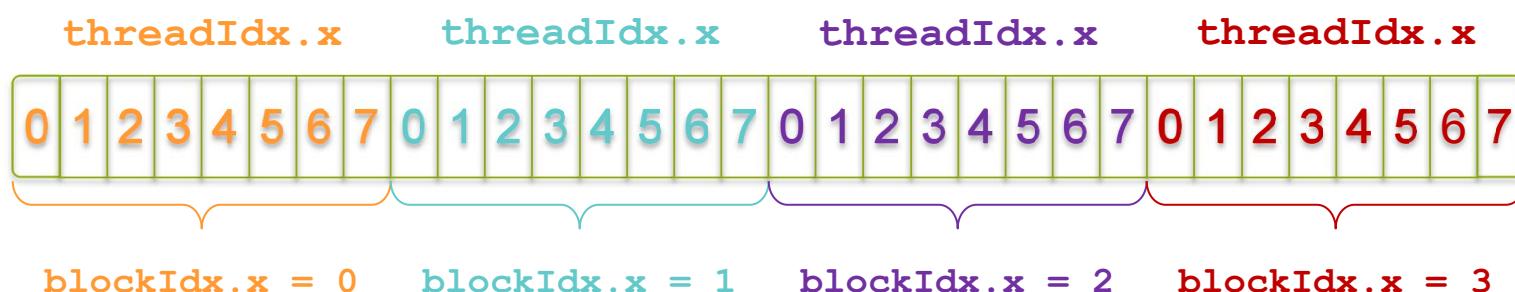
Combining Blocks and Threads

- We've seen parallel vector addition using:
 - Many blocks with one thread each
 - One block with many threads
- Let's adapt vector addition to use both blocks and threads
- Why? We'll come to that...
- First let's discuss data indexing...



Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
 - Consider indexing an array with one element per thread (8 threads/block)



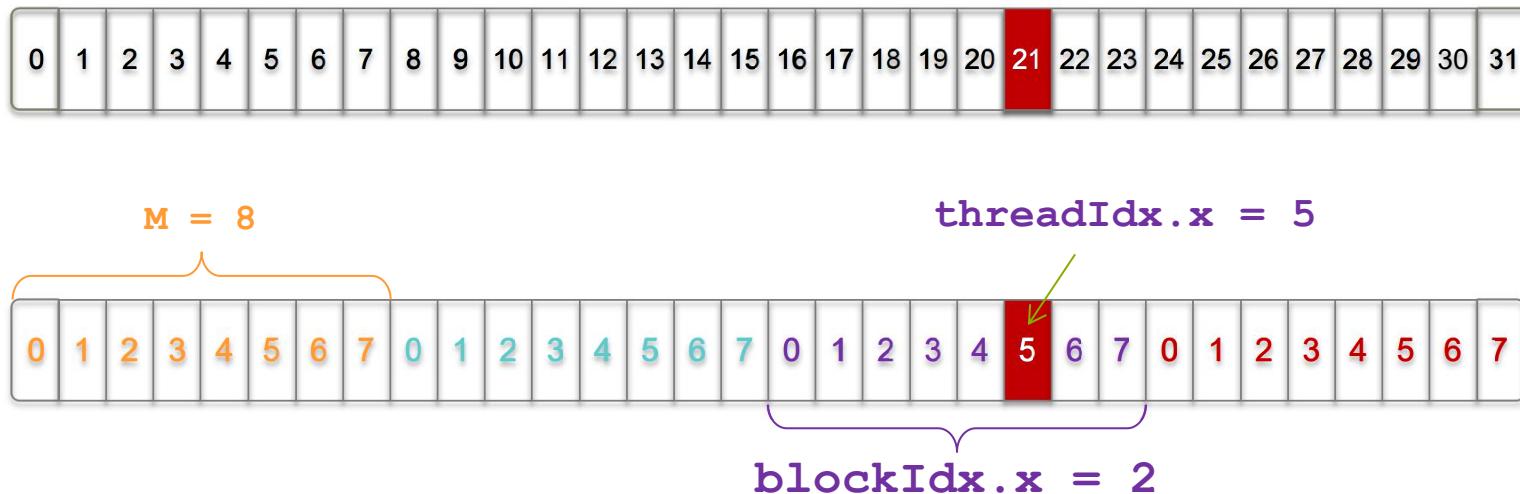
- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```



Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```



Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?



Addition with Blocks and Threads:

```
main()
```

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                                // host copies of a, b, c
    int *d_a, *d_b, *d_c;                            // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **) &d_a, size);
    cudaMalloc((void **) &d_b, size);
    cudaMalloc((void **) &d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```



Addition with Blocks and Threads:

```
main()
```

```
// Copy inputs to device
cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

// Cleanup
free(a); free(b); free(c);
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
return 0;
}
```



Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`
- Avoid accessing beyond the end of the arrays:

```
__global__ void add(int *a, int *b, int *c, int n) {  
    int index = threadIdx.x + blockIdx.x * blockDim.x;  
    if (index < n)  
        c[index] = a[index] + b[index];  
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```



Why Bother with Threads?

- Threads seem unnecessary
 - They add a level of complexity
 - What do we gain?
- Unlike parallel blocks, threads have mechanisms to:
 - Communicate
 - Synchronize
- To look closer, we need a new example...

Review

- Launching parallel kernels
 - Launch **N** copies of `add()` with `add<<<N/M,M>>>(...);`
 - Use `blockIdx.x` to access block index
 - Use `threadIdx.x` to access thread index within block
- Allocate elements to threads:

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```