



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems

Spring 2021

Instructor:

Dr. Manal Jalloul

Outline

These slides constitute the course outline and include the course syllabus and course policies.

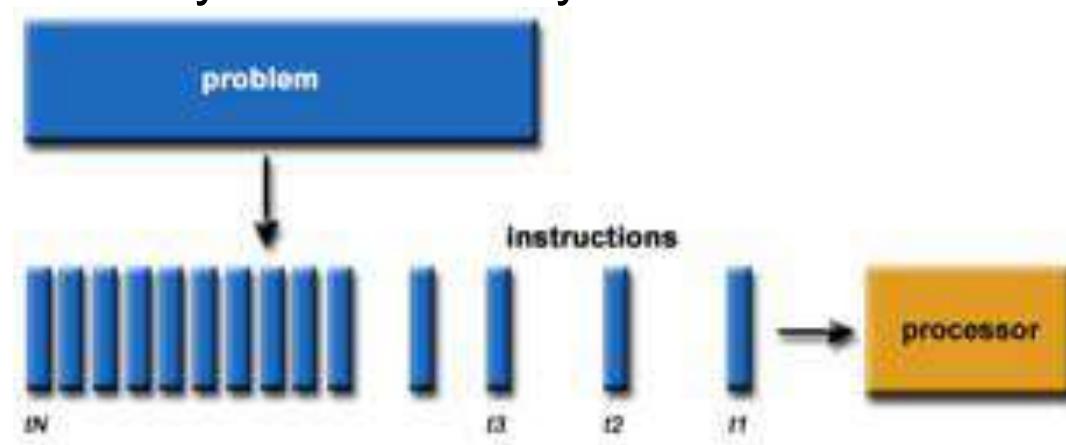
Parallel Computing

The use of multiple processors or computers to solve problems at a greater computational speed than using a single computer/processor.

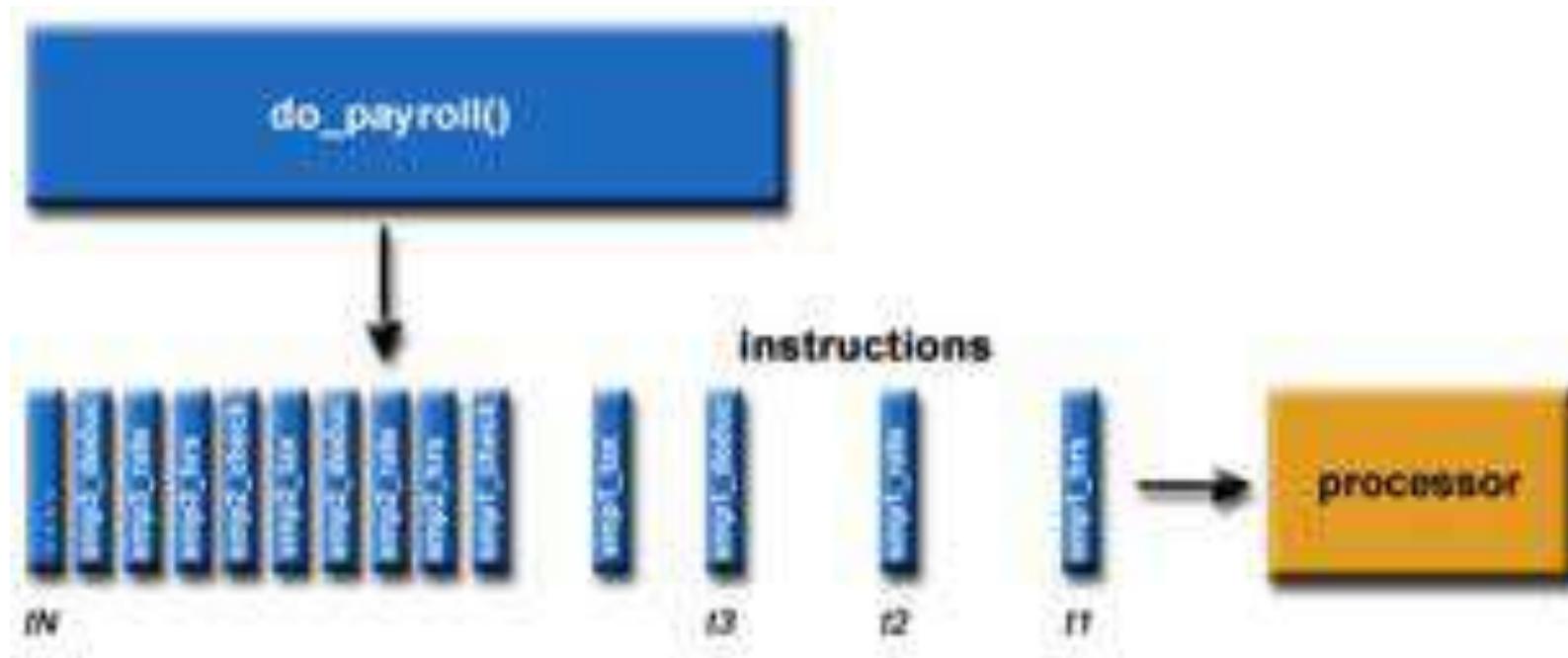
Basic idea is simple – using N computers/ processors collectively on a problem should lead to a faster solution.

What is Parallel Computing?

- Traditionally, software has been written for *serial* computation:
 - A problem is broken into a discrete series of instructions
 - Instructions are executed sequentially one after another
 - Executed on a single processor
 - Only one instruction may execute at any moment in time

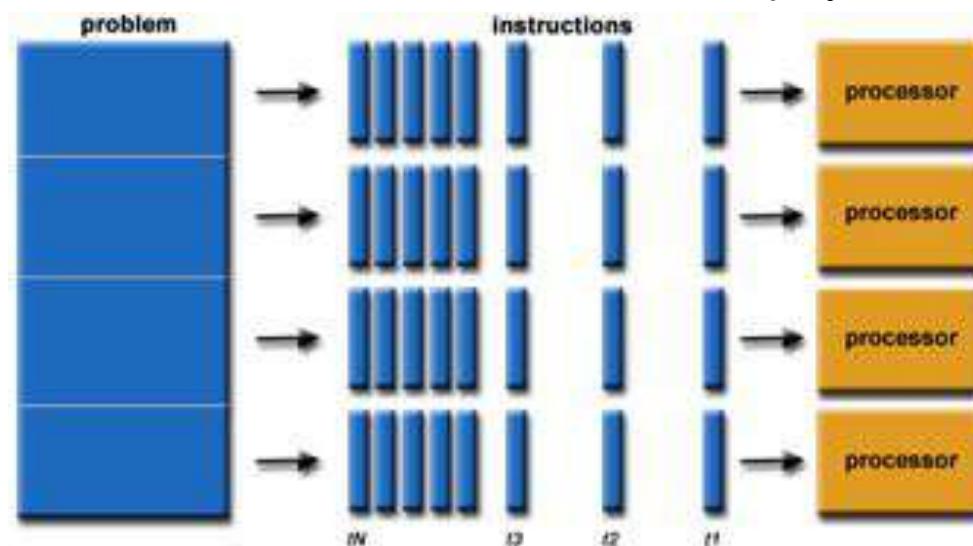


What is Parallel Computing?



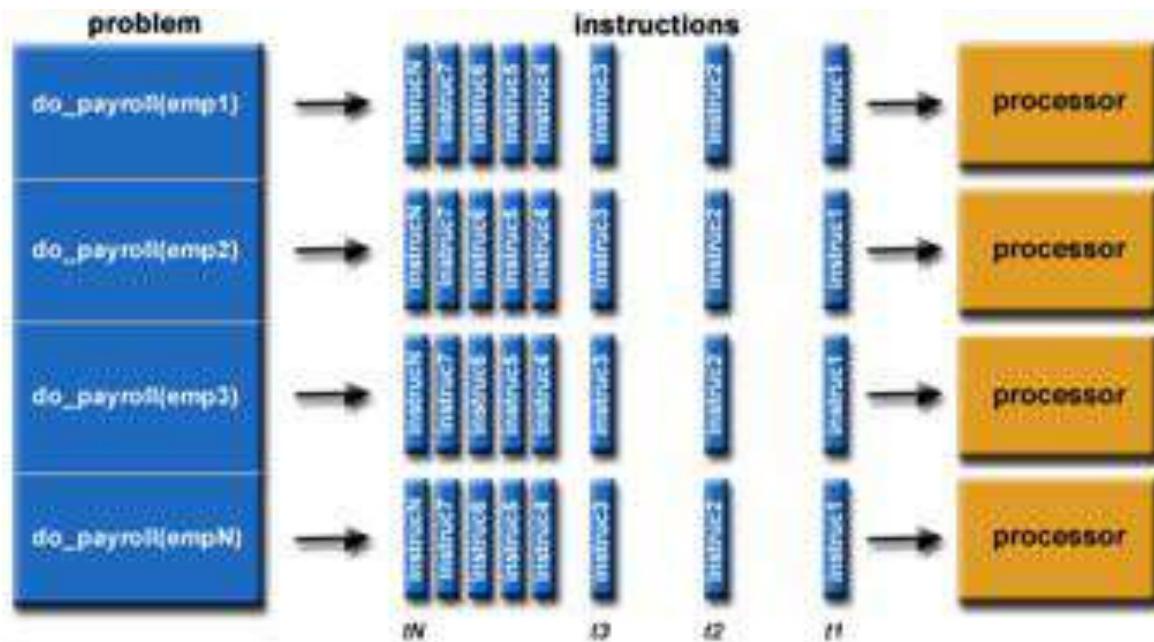
What is Parallel Computing?

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
 - A problem is broken into discrete parts that can be solved concurrently
 - Each part is further broken down to a series of instructions
 - Instructions from each part execute simultaneously on different processors
 - An overall control/coordination mechanism is employed



What is Parallel Computing?

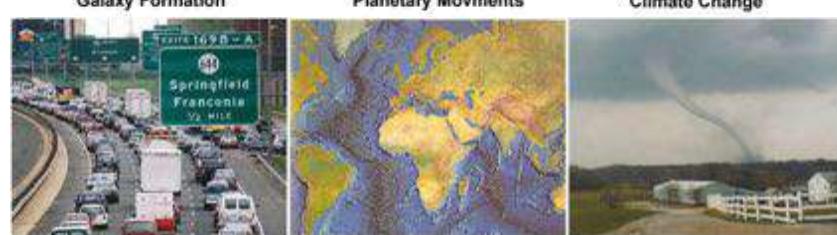
- *Parallel Computing* example:



- The computational problem should be able to:
 - Be broken apart into discrete pieces of work that can be solved simultaneously;
 - Execute multiple program instructions at any moment in time;
 - Be solved in less time with multiple compute resources than with a single compute resource.

Why Use Parallel Computing

- *The Real World is Massively Parallel:*
 - In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence.
 - Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex, real world phenomena.
 - For example, imagine modeling these serially:



Why Use Parallel Computing?

- SAVE TIME AND/OR MONEY:
 - In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings.
 - Parallel computers can be built from cheap, commodity components.



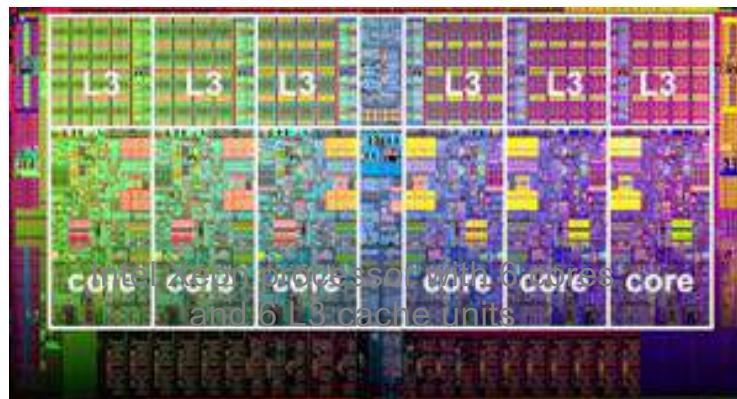
Why Use Parallel Computing?

- SOLVE LARGER / MORE COMPLEX PROBLEMS:
 - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
 - Example: "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.
 - Example: Web search engines/databases processing millions of transactions every second



Why Use Parallel Computing?

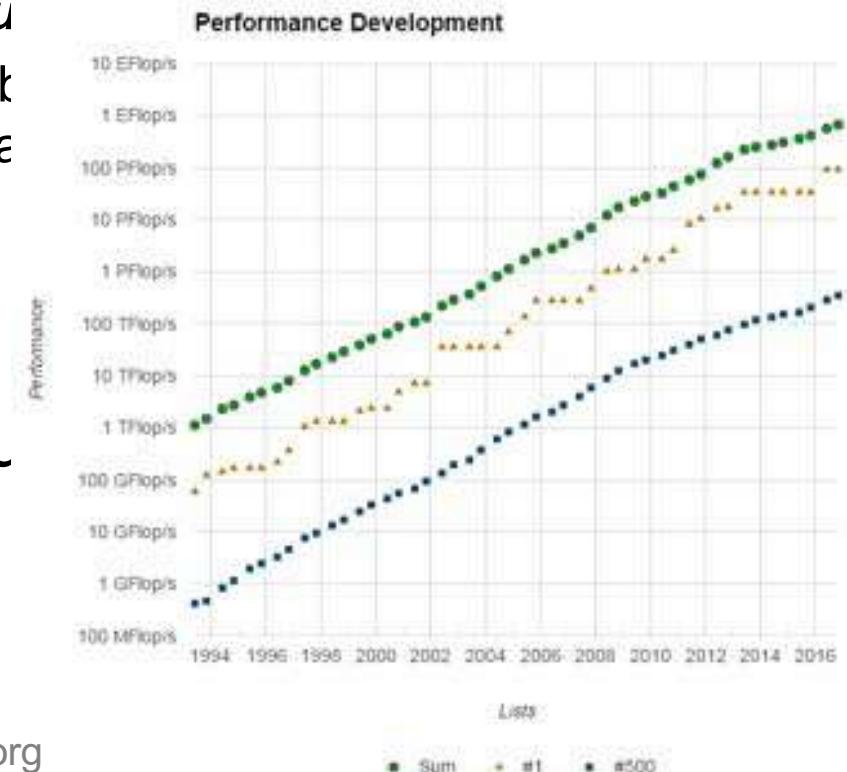
- MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE:
 - Modern computers, even laptops, are parallel in architecture with multiple processors/cores.
 - Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
 - In most cases, serial programs run on modern computers "waste" potential computing power.



Graphical Processing unit
(GPU)

The Future

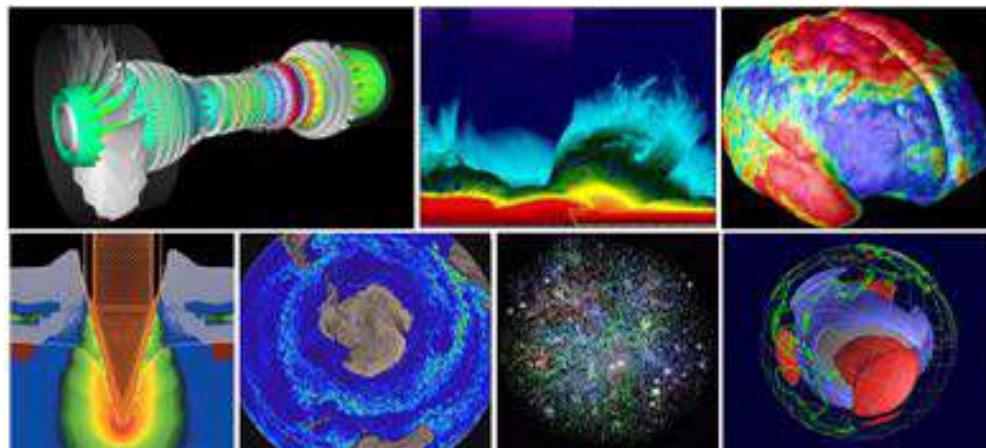
- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computation***
- In this same time period, there has been a dramatic increase in supercomputer performance, as shown in the graph below.
- ***The race is already on for Exascale Computing!***
 - Exaflop = 10^{18} calculations per second



Source: Top500.org

Who is Using Parallel Computing?

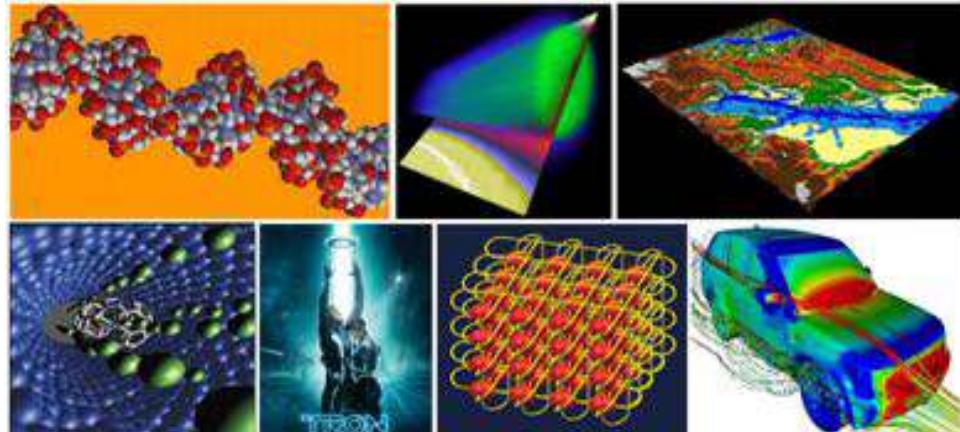
- **Science and Engineering:**
 - Atmosphere, Earth, Environment
 - Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
 - Bioscience, Biotechnology, Genetics
 - Chemistry, Molecular Sciences
 - Geology, Seismology
 - Mechanical Engineering - from prosthetics to spacecraft
 - Electrical Engineering, Circuit Design, Microelectronics
 - Computer Science, Mathematics
 - Defense, Weapons



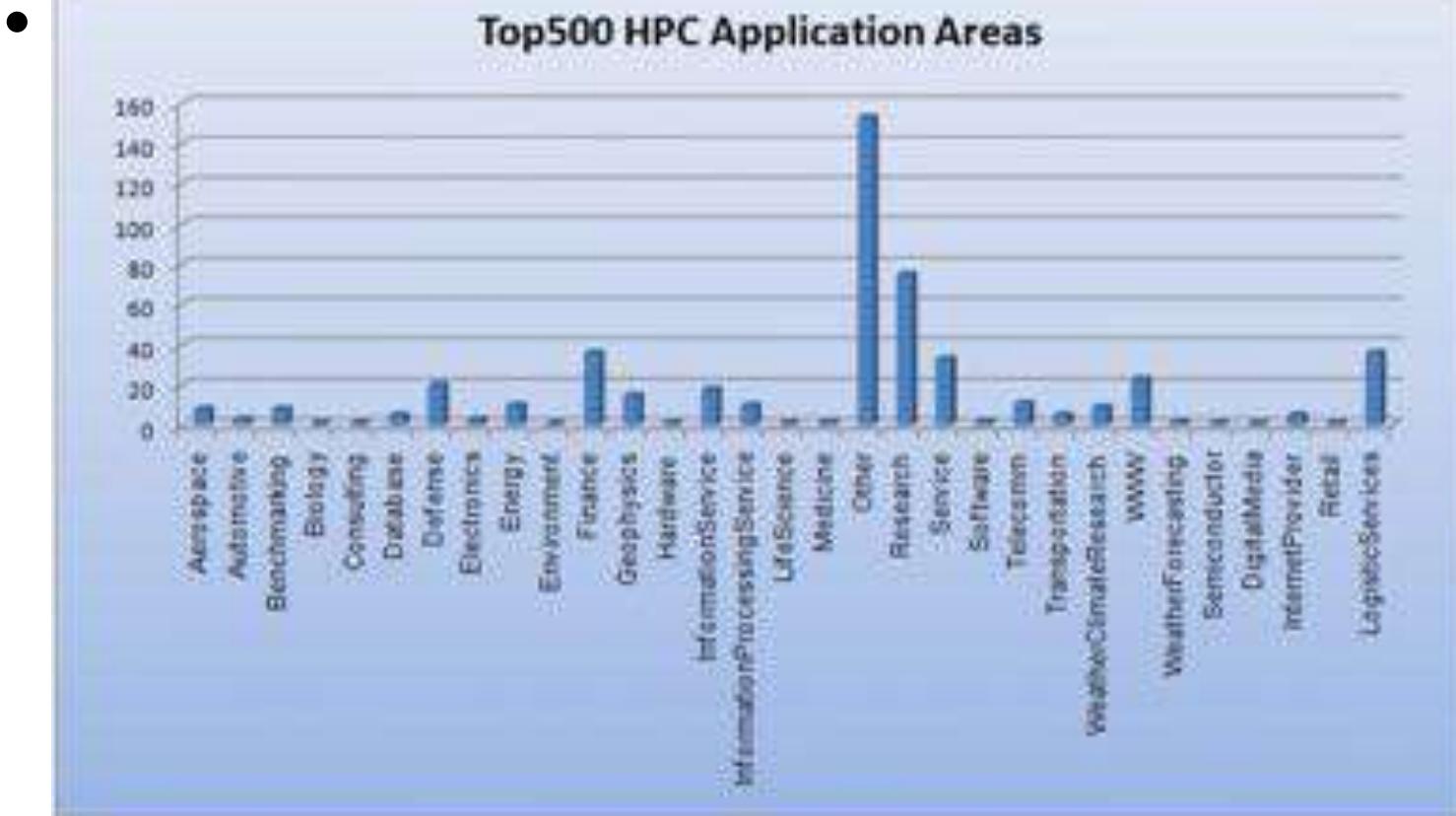
Who is Using Parallel Computing?

- **Industrial and Commercial:**

- "Big Data", databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies



Who is Using Parallel Computing?



Source: Top500.org

Computer platforms for parallel computing

1. Multiple interconnected computers
 - Cluster Computing, group of interconnected computers typically using Ethernet switch and physically all in one room.
2. A single computer system with multiple internal processors or cores
 - Usually shares a common main memory
3. Computer system with attached graphic processing unit (GPU)
 - GPUs have large number of execution cores, now used for high performance computing as well as original graphics application

Combination of above most likely.

Course Goals

- Designed to give insight into parallel computing
 - Motivation for using parallel computing
 - Available parallel computing architectures
- Learn how to program parallel computing systems
 - Design, analyze, and implement message-passing programming applications on a cluster of workstations
 - Design, analyze, and implement shared-memory programming applications on multicore systems
 - Design, analyze, and implement CUDA programming applications on the GPU
 - Analyze the efficiency of a given parallel algorithm
- Technical subjects
 - Parallel programming API, tools and techniques
 - Principles and patterns of parallel algorithms

Instructors details

Dr. Manal K. Jalloul

Email: manal.jalloul@lau.edu.lb

Online Office: <https://lau.webex.com/lau/j.php?MTID=mec8f6bb18b753d5d182ee4631dc45088>

Office Hours: Friday: 10:00-12:00 (or by appointment)

Course Load

- 3 credits
- Two 75-minute lectures per week
- Lab Sessions on Tuesdays
- Final project

Course Project

- The purpose of the project is to apply parallel programming tools and techniques to an open-ended problem
- Groups of three
- Deadline for forming groups and Proposal Submission is on **March 1st**
- Final project submission is at the end of semester (code, report, presentation)

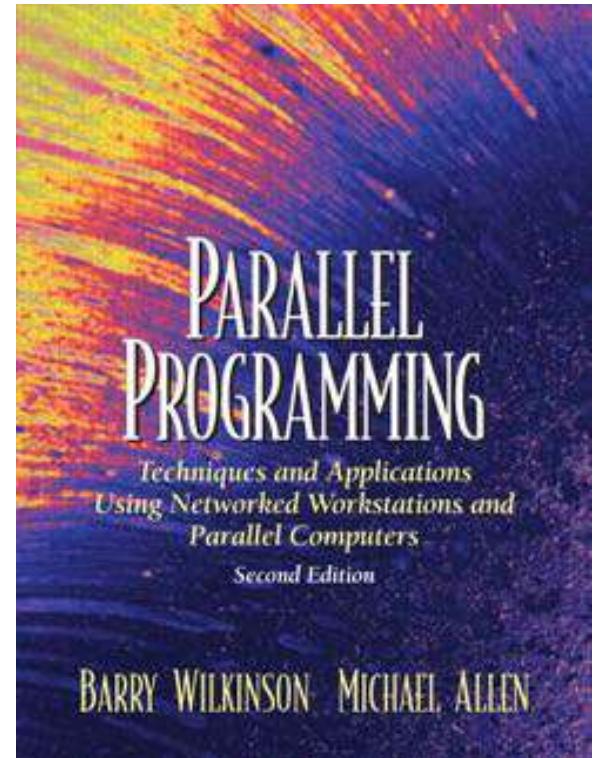
Course Prerequisites

- CSC310: Algorithms and Data Structures
CSC326: Operating Systems

What is needed -- basic skills in C.

Course References

- Wilkinson & Allen, Parallel Programming: Techniques & Applications using Networked Workstations & Parallel Computers, 2nd Edition, 2004.
https://webpages.uncc.edu/abw/parallel/parallel_prog/index.htm
- Kirk and Hwu, Prog. Massively Parallel Processors: A Hands-on Approach, Morgan Kaufman, 2017.
- M. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2003.



No Specific Text Book as detailed lecture notes will be provided

Grading Criteria

Tentative/ Subject to Change

- Labs: 15%
- Quizes/Assignments: 10%
- Project: 20%
- Midterm: 25%
- Final Exam: 30%

Course Policies

- Lectures will be delivered online via Webex. Online lectures will be recorded and recordings will be posted to Blackboard. Some lectures will be recorded and posted to Blackboard.
- Attendance to the online sessions is obligatory unless you have a valid excuse. Points will be deducted from total grade for unexcused absences.
- Short quizzes can be given during lectures unannounced.
- There are two types of assignments: exercises and lab assignments. The exercises consist of practice questions which are intended to assist the student in mastering the course content. Some of these assignments will be collected and graded, but you will be informed in advance when an assignment is to be handed in.
- The programming assignments require the solutions to problems using the computer. You will be instructed how to submit your lab assignments for grading. Typically you will be asked to submit an electronic version of your code, and test runs.

Course Policies

- Late Labs will be accepted, but will incur progressive penalties. No Labs containing Syntax errors will be graded.
- All Formal Homework Assignments (Including exercises and labs) and all Exams are to be treated as individual and not collective efforts. A severe penalty will be given to any assignment which indicates collusion or cheating. The usual penalty for cheating on lab assignments or an exam is failure in the course.
- You should carefully read the section on Academic Dishonesty. Your continued enrollment in this course implies that you have read it, and that you subscribe to the principles stated therein.

Course Policies

- The programming assignments should be considered as "open-book, take-home tests". If you need assistance with such an assignment, you may consult your professor, a CS TA designated to help CSC 447, your textbook, or any other textbook. You may not receive substantive assistance in any form from any other source (i.e., from another student, from paid or unpaid tutors, etc.). Any assistance you receive is to be documented in the comment section of your code.
- The only help you may receive from another student is with syntax errors or with questions regarding the computer system. Stealing another person's listing or having another person "ghost write" a lab will be considered cheating
- Complete lab code solutions will not be provided. Sample code will be provided to selective problems only.

Course Contents

Demand for computational speed: grand challenge problems.

Potential for speed-up using multiple process(or)s:
speed-up factor, max speed up, Amdahl's law,
Gustafson's law.

Parallel computers: architectural types, shared
memory, distributed memory, GPU systems.

Course Contents

(Continued)

Programming with shared memory programming I:
processes, fork, fork-join pattern, threads, Pthreads,
thread pool pattern.

OpenMP: thread team pattern, directives/constructs,
parallel, shared and local variables, work-sharing,
sections, for, loop scheduling, for reduction, single
master.

Course Contents

(Continued)

Message passing: MPI, point-to-point message passing, message tags, MPI communicator, blocking send/recv, command line compiling and executing MPI programs, instrumenting code for execution time

Message passing patterns: MPI collective routines, broadcast, scatter, gather, reduce, barrier, alltoall broadcast.

Implementation issues: barrier, local synchronization, safety and deadlock, safe MPI routines, synchronous message passing, asynchronous (non-blocking) message passing, changing to synchronous message passing.

Course Contents

(Continued)

Pattern programming concepts: problem addressed, low message-passing patterns, point to point data transfer, broadcast, scatter, gather, reduce, all-to-all broadcast, higher level message-passing patterns, workpool, pipeline, divide and conquer, all-to-all, iterative synchronous patterns, iterative synchronous all-to-all, stencil, advantages and disadvantages of patterns.

Course Contents

(Continued)

CPU-GPU systems: Introduction to heterogenous parallel computing, introduction to CUDA C, threads and kernel functions, memory allocation and data movement API functions, kernel-based SPMD parallel programming, memory model and locality, Tiled Algorithms, performance considerations, parallel computational patterns .

Questions?!



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Parallel Computing: Demand for High Performance

Parallel Computing

- Using more than one computer, or a computer with more than one processor, collectively to solve a problem.

Motives

- Usually faster computation.
- Very simple idea
 - n computers operating simultaneously can achieve the result faster
 - it will not be n times faster for various reasons
- Other motives include: fault tolerance, larger amount of memory available, ...

50

Parallel programming has been around for more than years. Gill writes in 1958*:

“... There is therefore nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers. It is not to be expected that the necessary programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming which seems at the present to be degenerating into a rather dull and routine occupation ...”

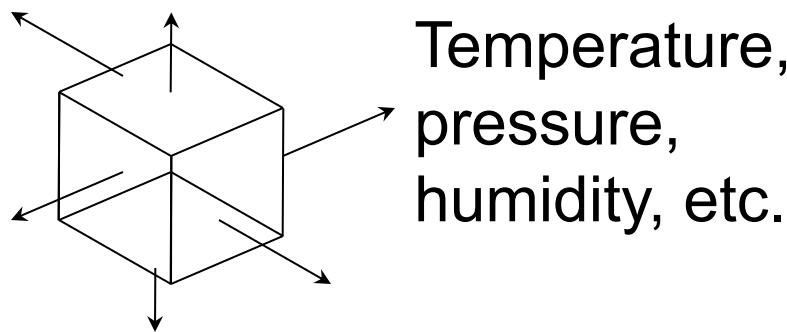
* Gill, S. (1958), “Parallel Programming,” *The Computer Journal*, vol. 1, April, pp. 2-10.

Problems needing multiple computers together to be solved

- The key aspect is the execution time.
- Sometimes there is a strict deadline for the solution, for example real time applications, flight control, ...
- Sometimes the deadline is less quantifiable, but still very important, for example:
 - CT scan image processing. A doctor needs that quickly within minutes or less to work effectively.
 - Computer assisted engineering design becomes ineffective if each step takes a long time.
- Sometimes present day computers simply cannot produce the result at all in a reasonable time -- so-called “Grand Challenge” problems, e.g. global weather forecasting, modeling large number of interacting bodies, large DNA structures ...

Weather Forecasting

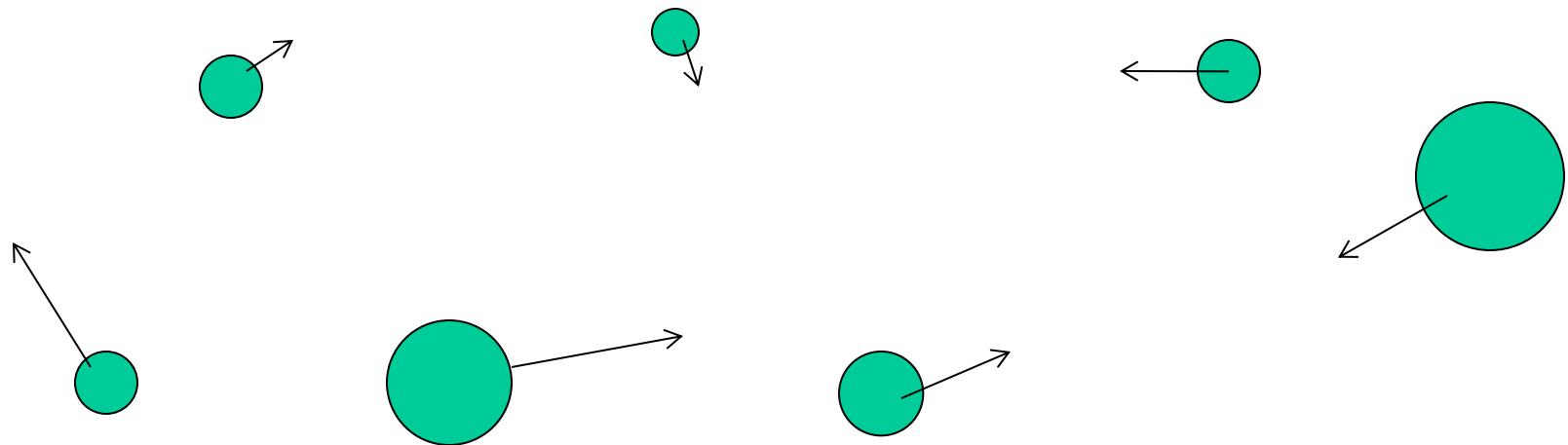
- Atmosphere modeled by dividing it into 3-dimensional cells.
- Calculations of each cell repeated many times to model passage of time.



Weather Forecasting

- Whole global atmosphere divided into cells of size 1 mile \times 1 mile \times 1 mile to a height of 10 miles (10 cells high) - about 5×10^8 cells.
- Suppose each calculation requires 200 floating point operations. In one time step, 10^{11} floating point operations necessary.
- To forecast the weather over 7 days using 1-minute intervals, a computer operating at 1Gflops (10^9 floating point operations/s) would take 106 seconds or over 10 days.
- To perform the calculation in 5 minutes would require a computer operating at 3.4 Tflops (3.4×10^{12} floating point operations/sec).

Modeling Motion of Astronomical Bodies



Each body attracted to each other body by gravitational forces.

Movement of each body predicted by calculating total force on each body and applying Newton's laws (in the simple case) to determine the movement of the bodies.

Modeling Motion of Astronomical Bodies

- Each body has $N-1$ forces on it from the $N-1$ other bodies - $O(N)$ calculation to determine the force on one body (three dimensional).
- With N bodies, approx. N^2 calculations, i.e. $O(N^2)$ *
- After determining new positions of bodies, calculations repeated, i.e. $N^2 \times T$ calculations where T is the number of time steps, i.e. the total complexity is $O(N^2T)$.

* There is an $O(N \log_2 N)$ algorithm, which we will cover in the course

- A galaxy might have, say, 10^{11} stars.
 - Even if each calculation done in 1 ms (extremely optimistic figure), it takes:
 - 10^9 years for one iteration using N^2 algorithm
- or
- Almost a year for one iteration using the $N \log_2 N$ algorithm assuming the calculations take the same time (which may not be true).
 - Then multiple the time by the number of time periods!

We may set the N -body problem as an assignment using the basic $O(N^2)$ algorithm. However, you do not have 10^9 years to get the solution and turn it in.



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems

Spring 2021

Potential for parallel computers/parallel programming

Before we embark on using a parallel computer, we need to establish whether we can obtain increased execution speed with an application and what the constraints are.

Speedup Factor

$$S(p) = \frac{\text{Execution time using one processor (best sequential algorithm)}}{\text{Execution time using a multiprocessor with } p \text{ processors}} = \frac{t_s}{t_p}$$

where t_s is execution time on a single processor, and
 t_p is execution time on a multiprocessor.

$S(p)$ gives increase in speed by using multiprocessor.

Typically use best sequential algorithm for single processor system.

Underlying algorithm for parallel implementation might be (and is usually) different.

Parallel time complexity

Can extend sequential time complexity to parallel computations.
Speedup factor can also be cast in terms of computational steps:

$$S(p) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } p \text{ processors}}$$

although several factors may complicate this matter, including:

- Individual processors usually do not operate in synchronism and do not perform their steps in unison
- Additional message passing overhead

Maximum Speedup

Maximum speedup usually p with p processors (**linear speedup**). $S(p) \leq p$

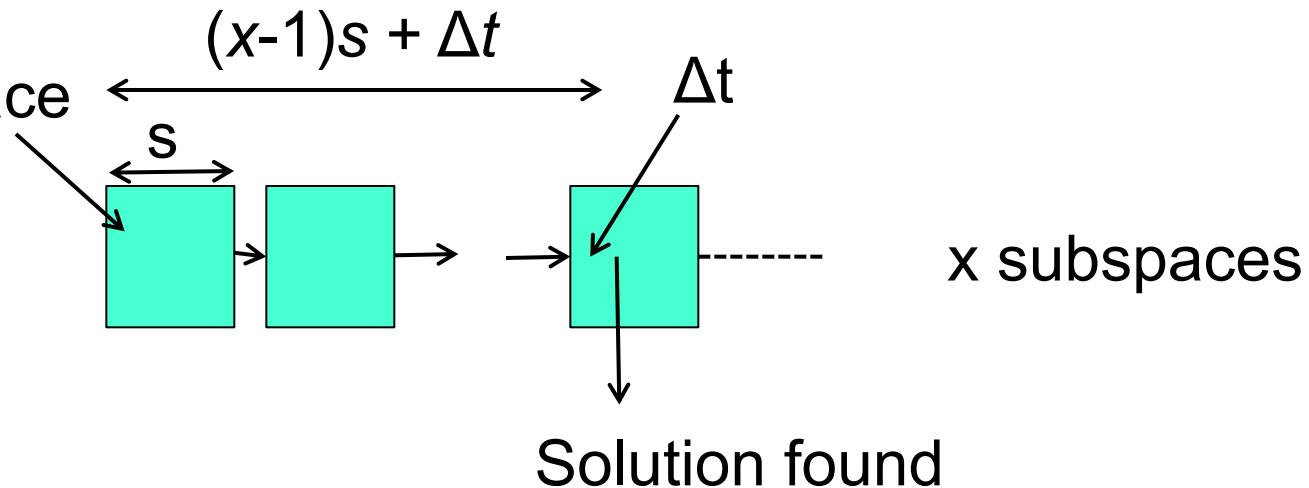
Possible to get **superlinear** speedup (greater than p) but usually a specific reason such as:

- Extra memory in multiprocessor system
- Nondeterministic algorithm
- Sequential algorithm not the fastest known sequential algorithm
- Sequential execution on a different machine (slower)

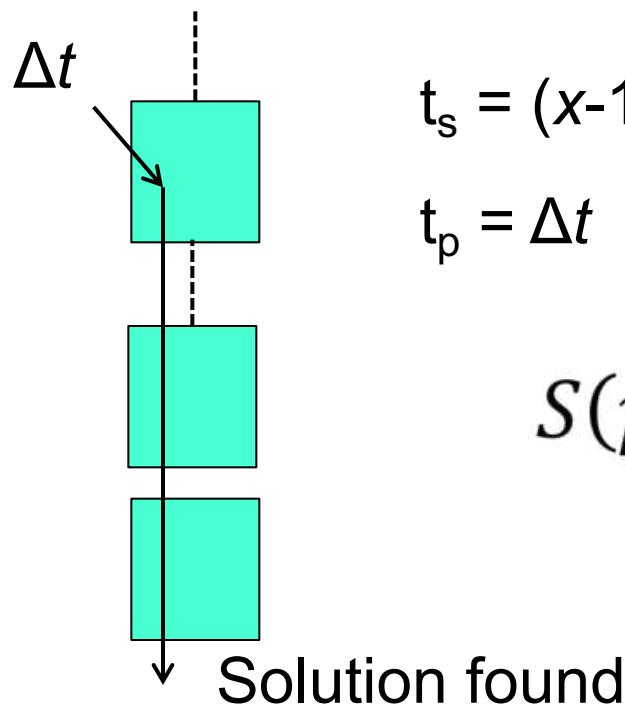
Superlinear speedup example with a nondeterministic algorithm

Searching

(a) Searching each sub-space sequentially



(b) Searching each sub-space in parallel



$$t_s = (x-1)s + \Delta t$$

$$t_p = \Delta t$$

$$S(p) = \frac{(x - 1)s + \Delta t}{\Delta t}$$

x indeterminate

Speed-up

- Solution found in last sub-space. Worst case for sequential search.

Greatest benefit for parallel version -

- Solution found in first sub-space search of sequential search. Least advantage for parallel version search -

$$S(p) = \frac{(x-1)s + \Delta t}{\Delta t}$$

$$S(p) = \frac{(p-1)s + \Delta t}{\Delta t}$$

$S(p) \rightarrow \infty$ as Δt reduces

$$S(p) = \frac{\Delta t}{\Delta t} = 1$$

$S(p) \rightarrow 1$ as x reduces

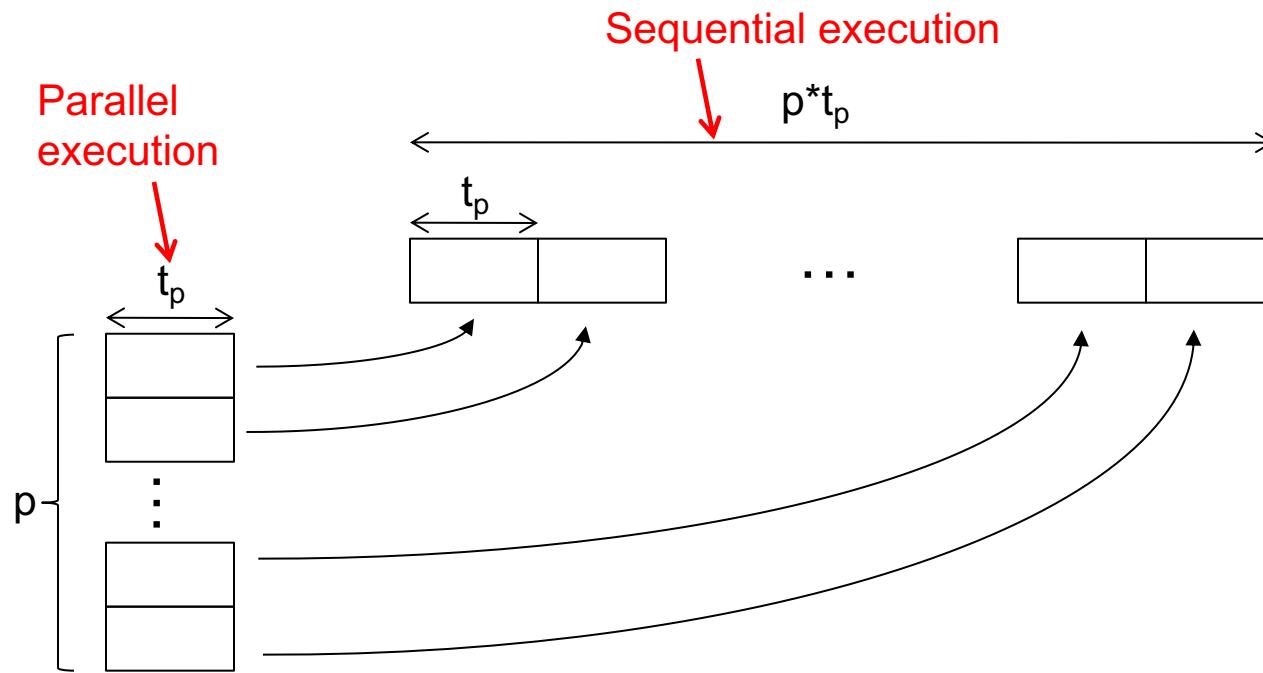
Actual speed-up depends upon which subspace holds solution but could be extremely large.

Why superlinear speedup should not be possible generally?

Suppose $t_p < p * t_s$, i.e. superlinear speedup.

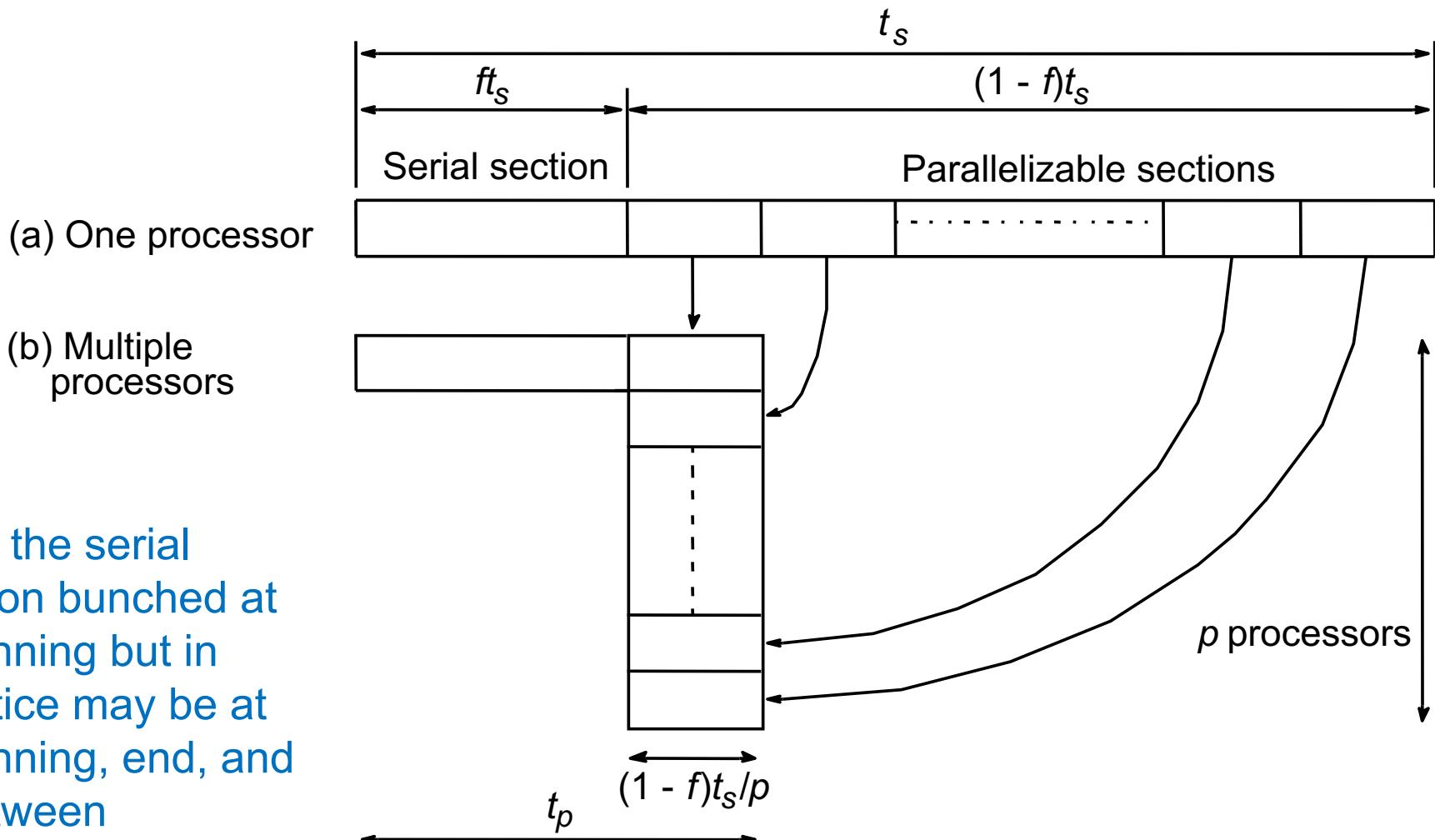
We could create a new sequential execution that simply executes the parts of each parallel processor one at a time:

- a sequential algorithm faster than the existing sequential algorithm!



Maximum speedup with parts not parallelizable (usual case)

Amdahl's law (circa 1967)

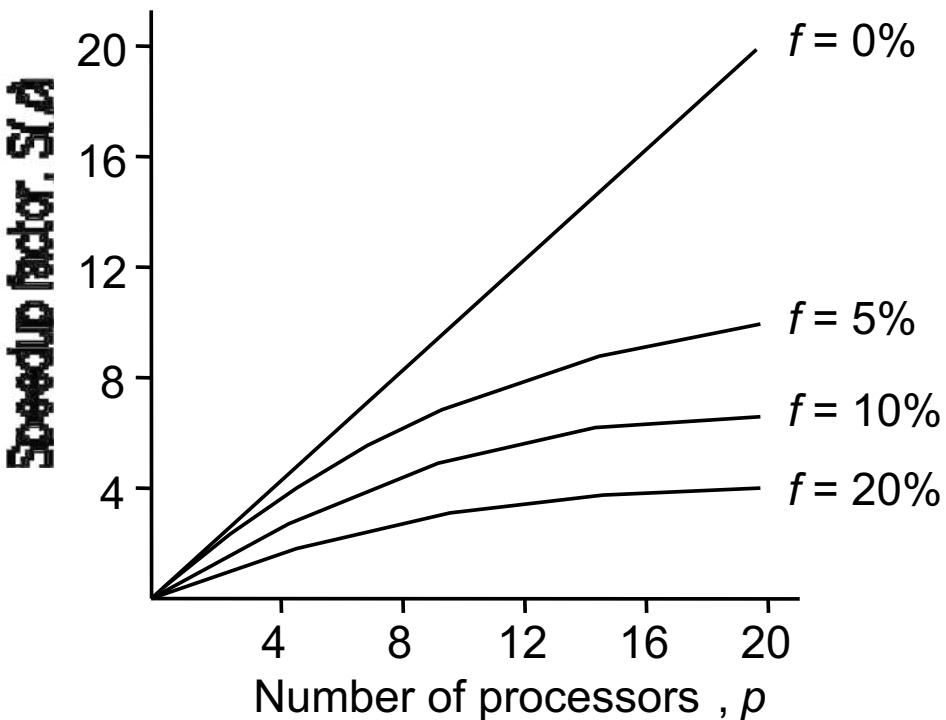


Speedup factor is given by:

$$S(p) = \frac{t_s}{ft_s + (1 - f)t_s/p} = \frac{p}{1 + (p - 1)f}$$

This equation is known as **Amdahl's law**.

Speedup against number of processors



Even with infinite number of processors, maximum speedup limited to $1/f$.

Example

With only 5% of computation serial, max. speedup is 20, irrespective of number of processors.

This is a very discouraging result.

Amdahl used this argument to support the design of ultra-high speed single processor systems in the 1960s.

Gustafson's law

Later, Gustafson (1988) described how the conclusion of Amdahl's law might be overcome by considering the effect of increasing the problem size.

He argued that when a problem is ported onto a multiprocessor system, larger problem sizes can be considered, that is, the same problem but with a larger number of data values.

Gustafson's law

Starting point for Gustafson's law is the computation on the multiprocessor rather than on the single computer.

In Gustafson's analysis, **parallel execution time kept constant**, which we assume to be some acceptable time for waiting for the solution.

Gustafson's law*

Using same approach as for Amdahl's law:

$$S'(p) = \frac{f't_p + (1 - f')pt_p}{t_p} = p - (p - 1)f'$$

f' is the fraction of parallel computation that must be done sequentially – not the same as f previously

Conclusion - almost linear increase in speedup with increasing number of processors, but fractional sequential part f' needs to diminish with increasing problem size.

Example: if f' is 5%, scaled speedup is 19.05 with 20 processors whereas with Amdahl's law and $f = 5\%$ speedup is 10.26. Gustafson quotes results obtained in practice of very high speedup close to linear on a 1024-processor hypercube.

See Wikipedia http://en.wikipedia.org/wiki/Gustafson%27s_law for original derivation of Gustafson's law.

Next question to answer is how does one construct a computer system with multiple processors to achieve the speed-up?





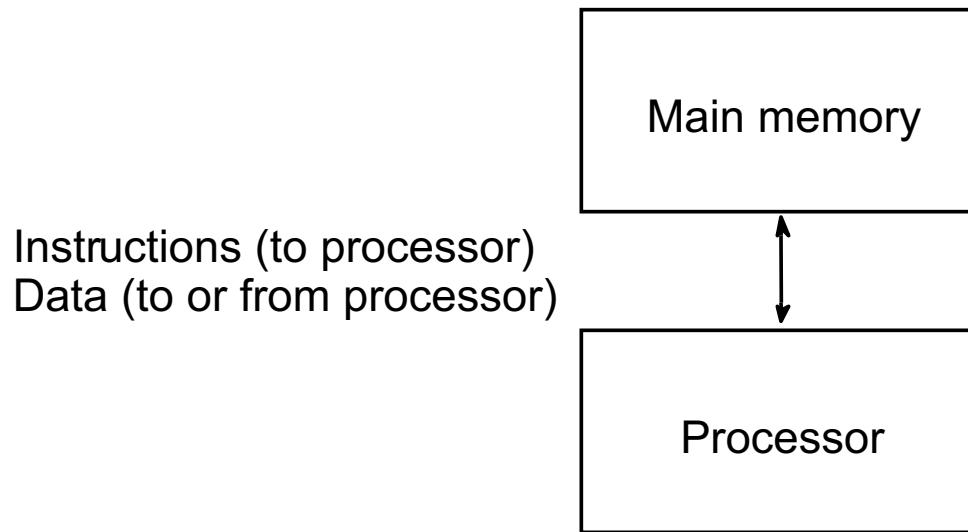
Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

**Constructing a system with
multiple computers or processors**

Conventional Computer

Consists of a processor executing a program stored in a (main) memory:



Each main memory location located by its address.
Addresses start at 0 and extend to $2^b - 1$ when there are b bits (binary digits) in address.

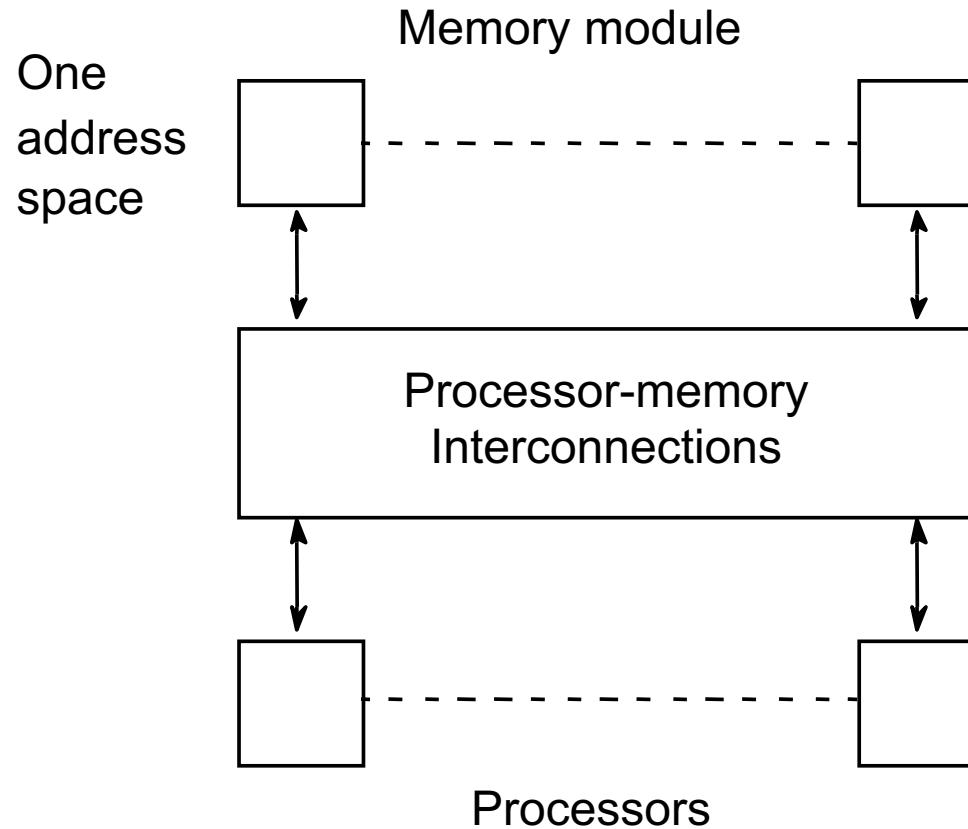
Types of Parallel Computers

Two principal approaches:

- Shared memory multiprocessor
- Distributed memory multicomputer

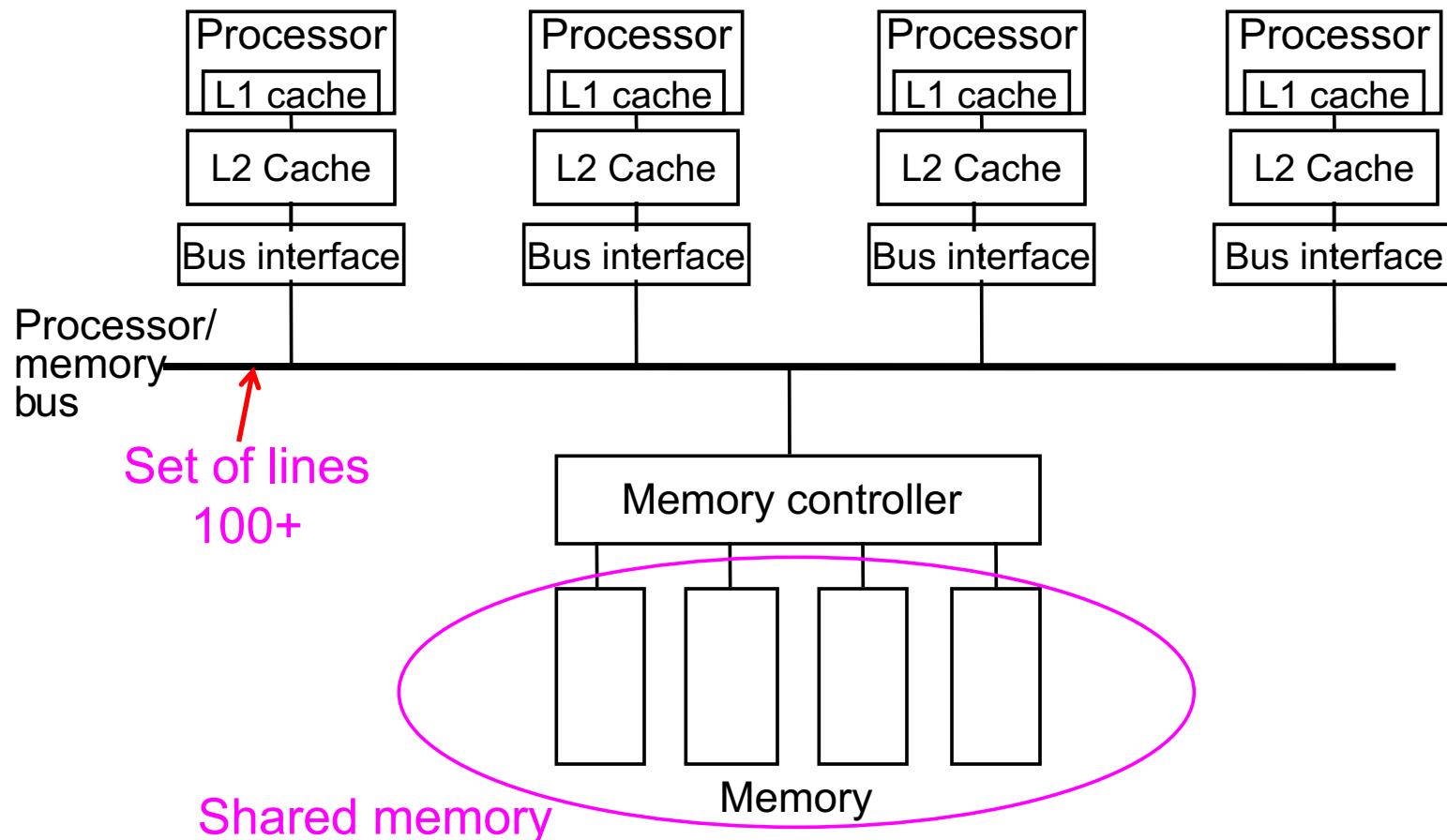
1. Shared Memory Multiprocessor System

Natural way to extend single processor model - have multiple processors connected to multiple memory modules, such that each processor can access any memory module:



Using a processor-memory bus as the interconnection network

Example – Dual and Quad Processor Shared Memory Multiprocessors

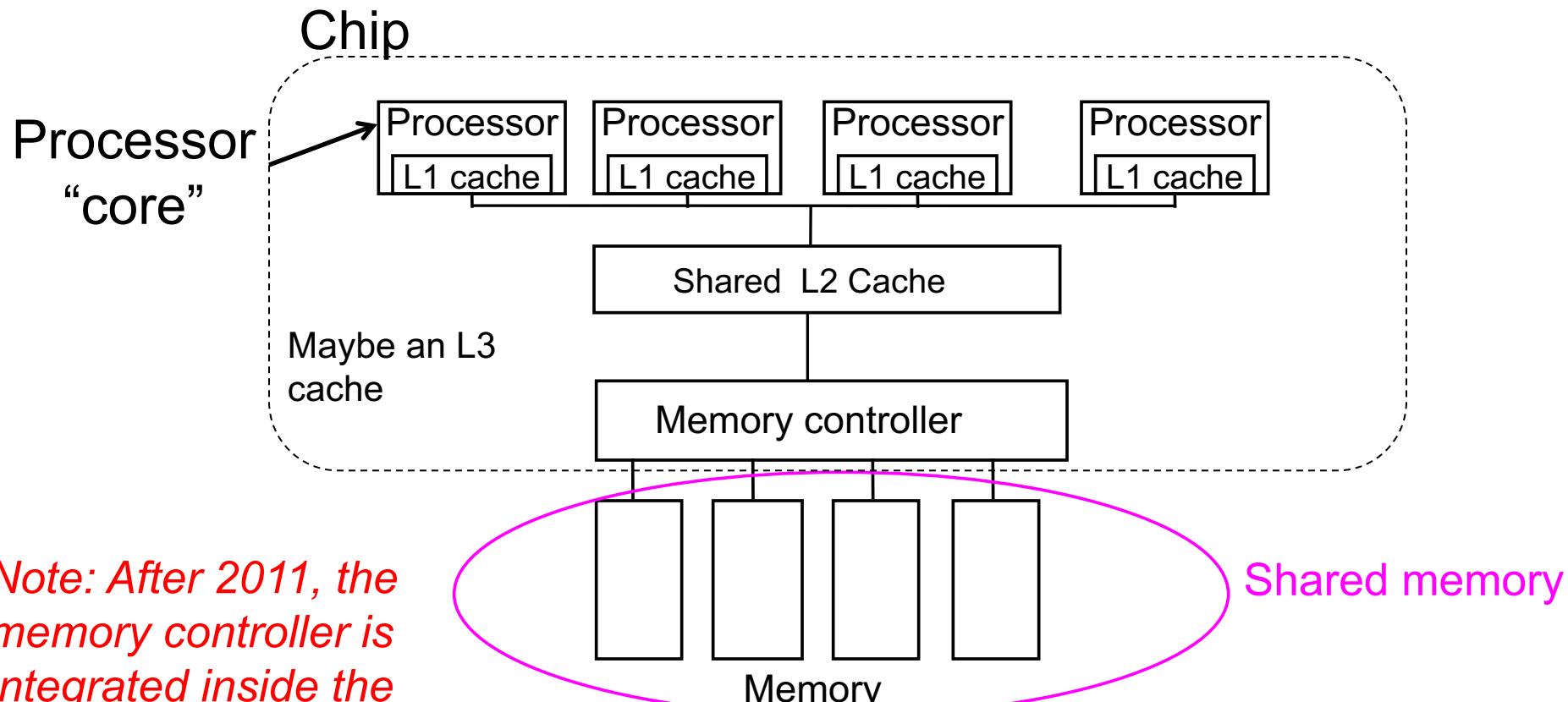


Typically nowadays individual connections are used instead of a bus.

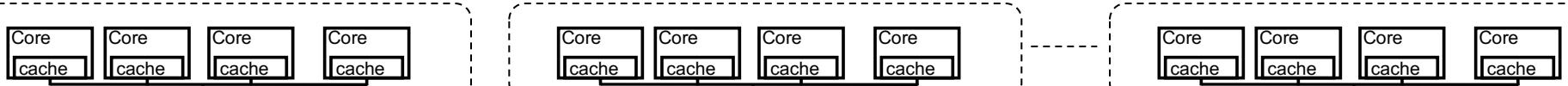
“Recent” innovation(since 2005)

- Dual-core and multi-core processors
- Two or more independent processors in one package
- Actually an old idea but not put into wide practice until recently because limits of making single processors faster principally caused by:
 - Power dissipation (power wall) and clock frequency limitations
 - Limits in parallelism within a single instruction stream
 - Memory speed limitations (memory wall)

Single quad core shared memory multiprocessor

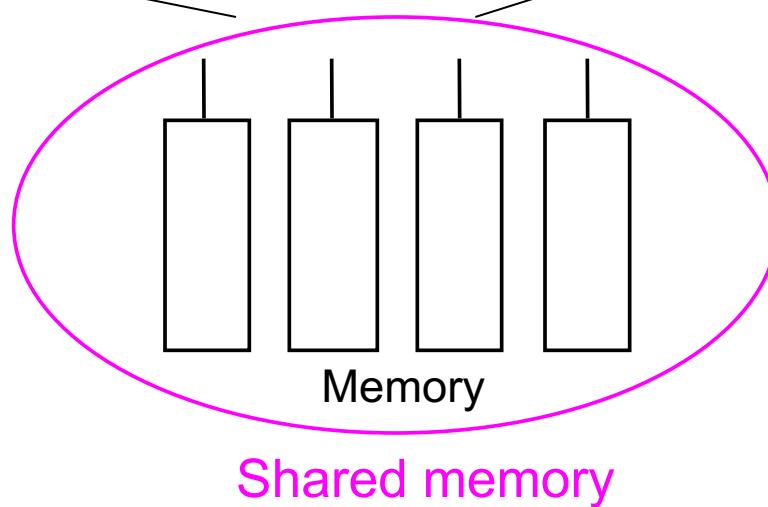


Multiple quad-core multiprocessors



Examples

Four processors each quad core. All 16 cores have access to 64 GB shared main memory (thro multilevel caches)



Programming Shared Memory Multiprocessors

Several possible ways – Usual approach is to use threads

Threads - individual parallel sequences (threads), each thread having their own local variables but being able to access shared variables declared outside threads.

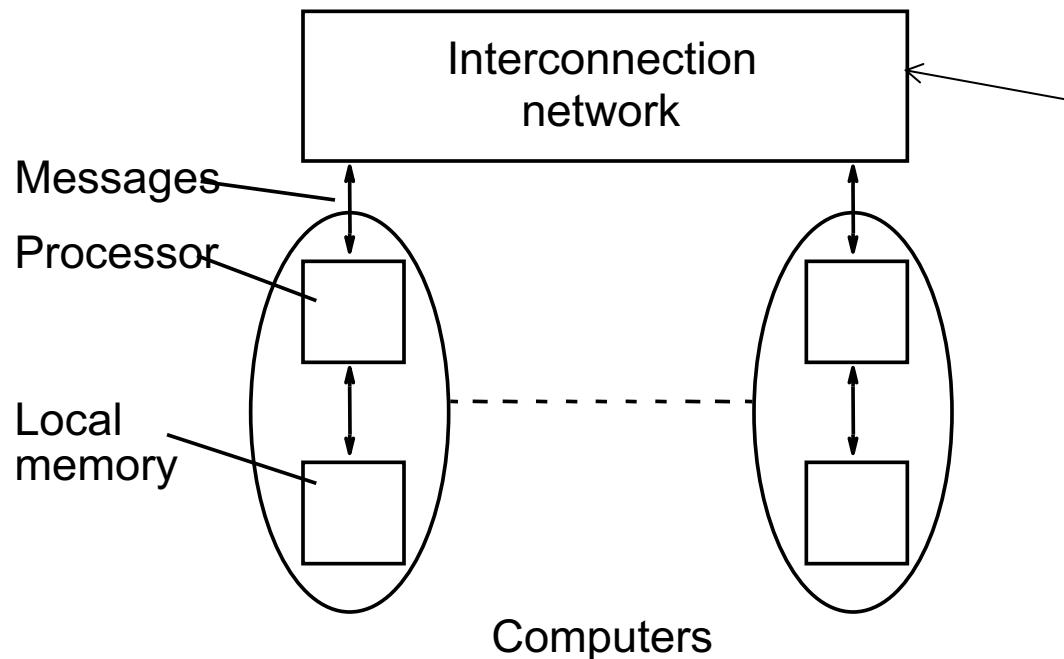
1. **Low-level thread libraries** - programmer calls thread routines to create and control the threads.
Examples: Pthreads, Java threads.
2. **Higher level library functions and preprocessor compiler directives.**
Example: OpenMP – an industry standard.

Other programming alternatives

- Sequential programming language with added syntax to declare shared variables and specify parallelism.
Example UPC (Unified Parallel C) - needs a UPC compiler.
- Parallel programming language with syntax to express parallelism - compiler creates executable code for each processor (not now common)
- Sequential programming language and ask parallelizing compiler to convert it into parallel executable code. - also not now common

2. Distributed Memory Multicomputer

Complete computers connected through an interconnection network:



Many interconnection networks explored including 2- and 3-dimensional meshes, hypercubes, and multistage interconnection networks

Networked Computers as a Computing Platform

- Became a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing in early 1990s.
- Several early projects. Notable: NASA Beowulf project.
- “Beowulf” cluster -- A group of interconnected commodity computers achieving high performance with low cost. Typically using commodity interconnects - high speed Ethernet, and Linux OS.

Key advantages of using commodity networked computers:

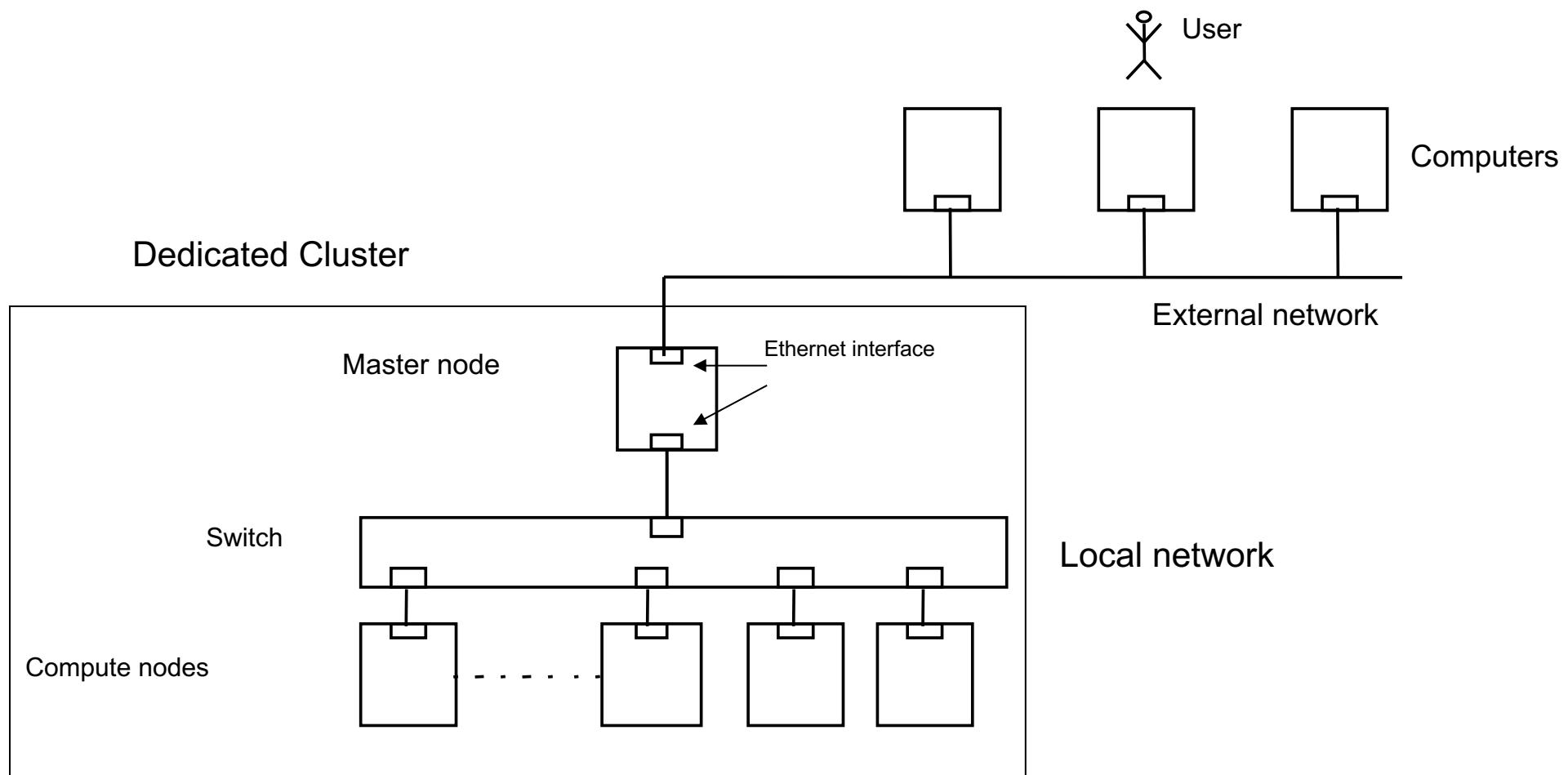
- Very high performance workstations and PCs readily available at low cost.
- Latest processors can easily be incorporated into the system as they become available.
- Existing software can be used or modified.

Cluster Interconnects

- Originally fast Ethernet on low cost clusters
- Gigabit Ethernet - easy upgrade path

More specialized/higher performance interconnects available including Infiniband.

Dedicated cluster with a master node and compute nodes



Software Tools for Clusters

- Based upon message passing programming model
- User-level libraries provided for explicitly specifying messages to be sent between executing processes on each computer .
- Use with regular programming languages (C, C++, ...).
- Can be quite difficult to program correctly as we shall see.

Flynn's Classifications

Flynn (1966) created a classification for computers based upon instruction streams and data streams:

- *Single instruction stream-single data stream (SISD) computer*

Single processor computer - single stream of instructions generated from program. Instructions operate upon a single stream of data items.

Multiple Instruction Stream-Multiple Data Stream (MIMD) Computer

General-purpose multiprocessor system - each processor has a separate program and one instruction stream is generated from each program for each processor. Each instruction operates upon different data.

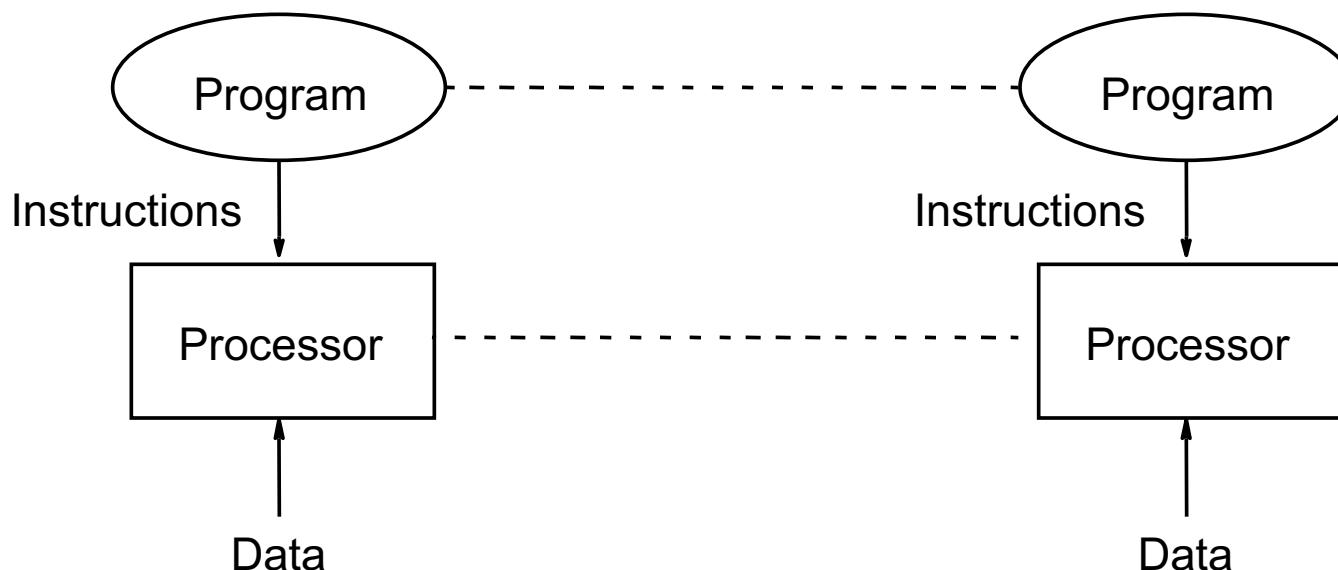
Both the shared memory and the message-passing multiprocessors so far described are in the MIMD classification.

Single Instruction Stream-Multiple Data Stream (SIMD) Computer

- A specially designed computer - a single instruction stream from a single program, but multiple data streams exist. Instructions from program broadcast to more than one processor. Each processor executes same instruction in synchronism, but using different data.
- Developed because a number of important applications that mostly operate upon arrays of data.

Multiple Program Multiple Data (MPMD) Structure

Within the MIMD classification, each processor will have its own program to execute:



Single Program Multiple Data (SPMD) Structure

Single source program written and each processor executes its personal copy of this program, although independently and not in synchronism.

Source program can be constructed so that parts of the program are executed by certain computers and not others depending upon the identity of the computer.

Using GPUs for high performance computing

- GPUs (graphics processing units) originally designed to speed up and support graphics operations
- Now also used for high performance computing.
- GPUs now have 100's or 1000's of processing cores and provide orders of magnitude increase in execution speed.

We will look at GPU devices and how to program them in the second half of the course

GPU clusters

- Recent trend for clusters – incorporating GPUs for high performance.
- Several of the fastest computers in the world are GPU clusters

GPU clusters

According to Top500 List June 2020:

<https://www.top500.org/lists/top500/2020/06/>

- The new top system, Fugaku, turned in a High Performance Linpack (HPL) result of 415.5 petaflops, besting the now second-place Summit system by a factor of 2.8x. Fugaku, is powered by Fujitsu's 48-core A64FX SoC, becoming the first number one system on the list to be powered by ARM processors. In single or further reduced precision, which are often used in machine learning and AI applications, Fugaku's peak performance is over 1,000 petaflops (1 exaflops).

GPU clusters

According to Top500 List June 2020:

<https://www.top500.org/lists/top500/2020/06/>

- Number two on the list is Summit, an IBM-built supercomputer that delivers 148.8 petaflops on HPL. The system has 4,356 nodes, each equipped with two 22-core Power9 CPUs, and six NVIDIA Tesla V100 GPUs. The nodes are connected with a Mellanox dual-rail EDR InfiniBand network. Summit is running at Oak Ridge National Laboratory (ORNL) in Tennessee and remains the fastest supercomputer in the US.

GPU clusters

According to Top500 List June 2020:

<https://www.top500.org/lists/top500/2020/06/>

- At number three is Sierra, a system at the Lawrence Livermore National Laboratory (LLNL) in California achieving 94.6 petaflops on HPL. Its architecture is very similar to Summit, equipped with two Power9 CPUs and four NVIDIA Tesla V100 GPUs in each of its 4,320 nodes. Sierra employs the same Mellanox EDR InfiniBand as the system interconnect.

GPU clusters

According to Top500 List November 2020:

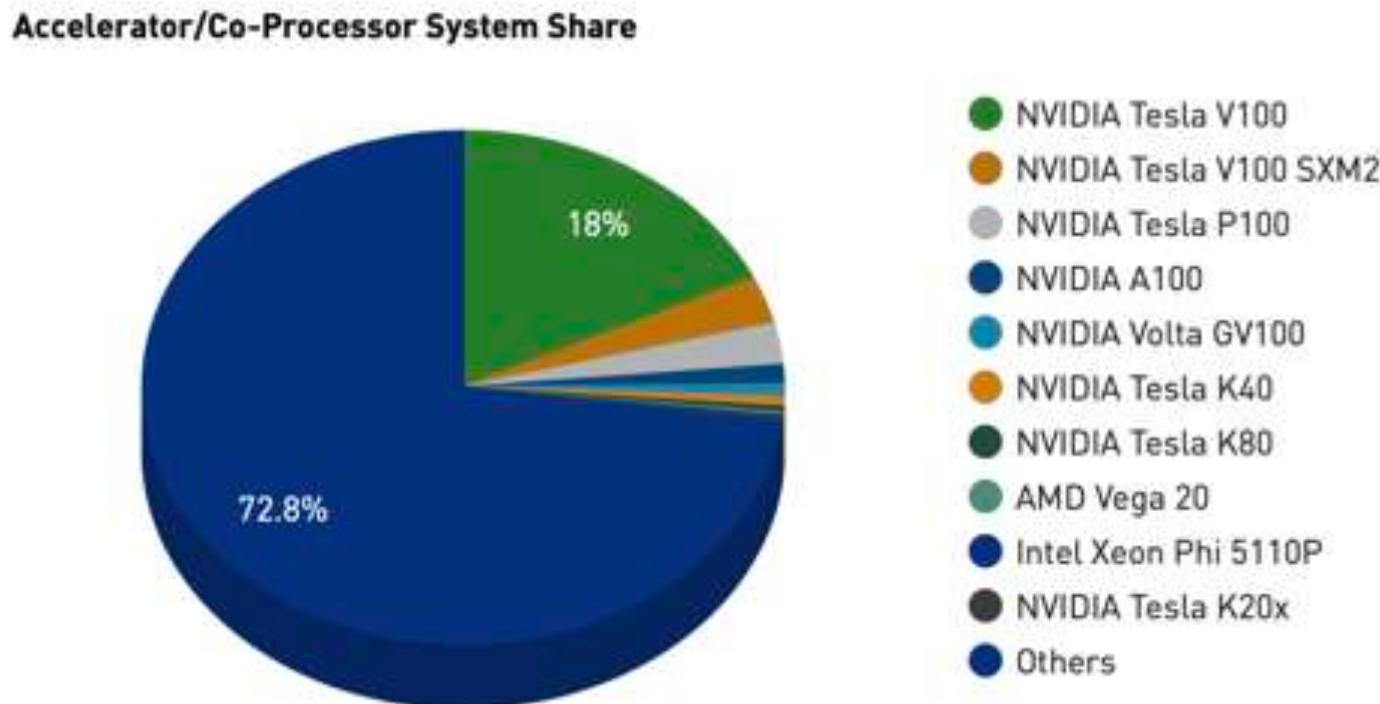
<https://www.top500.org/lists/top500/2020/11/>

- At number five is Selene, an NVIDIA DGX A100 SuperPOD installed in-house at NVIDIA Corp. It was listed as number seven in June but has doubled in size, allowing it to move up the list by two positions. The system is based on AMD EPYC processors with NVIDIA's new A100 GPUs for acceleration. Selene achieved 63.4 petaflops on HPL as a result of the upgrade.

GPU clusters

According to Top500 List November 2020:

<https://www.top500.org/lists/top500/2020/11/>



Next step

- Learn how to program multi-computer systems
- We will start with distributed memory programming and MPI.





Department of Computer Science & Mathematics

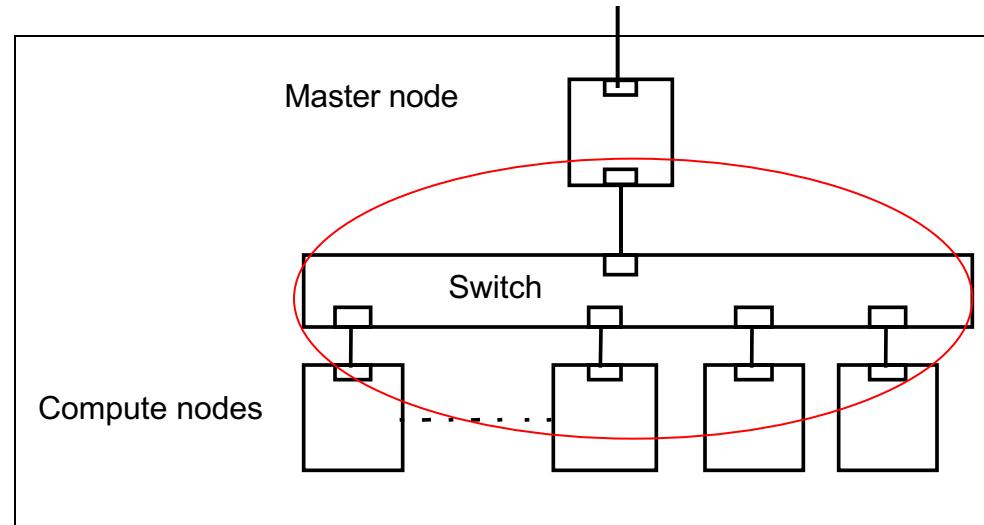
CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Programming distributed memory systems

Clusters
Distributed computers

Computer Cluster

- Complete computers connected together through an interconnection network, often Ethernet switch.
- Memory of each computer not directly accessible from other computers (distributed memory system)



Programming model – separate processes running on each system communicating through explicit messages to exchange data and synchronization.

MPI (Message Passing Interface)

- Widely adopted message passing library standard. MPI-1 finalized in 1994, MPI-2 in 1996, MPI-3 in 2012
- Process-based -- processes communicate between themselves with messages. Point-to-point and collectively.
- ***A specification, not an implementation.***
- Several free implementations exist, OpenMPI, MPICH,
- Large number of routines: MPI-1 128 routines, MPI-2 287 routines, MPI-3 440+ routines, but typically only a few used.
- C and Fortran bindings (C++ removed from MPI-3)
- Originally for distributed systems but now used for all types, clusters, shared memory, hybrid.

Some common MPI Routines

We will look into the use of these routines shortly

Environment

[**MPI_Init\(\)**](#)

- Initialize MPI (No MPI routines before this)

[**MPI_Comm_size\(\)**](#)

- Get number of processes
(in a communicating domain)

[**MPI_Comm_rank\(\)**](#)

- Get process ID (rank)

[**MPI_Finalize\(\)**](#)

- Terminate MPI (No MPI routines after this)

Pont-to-point message passing

[**MPI_Send\(\)**](#)

- Send a message, locally blocking

[**MPI_Recv\(\)**](#)

- Receive a message, locally blocking

[**MPI_SSend\(\)**](#)

- Send a message, synchronous

[**MPI_Isend\(\)**](#)

- Send a message, non blocking

Collective message passing

[**MPI_Gather\(\)**](#)

- All to one, collect elements of an array

[**MPI_Scatter\(\)**](#)

- One to all, send elements of array

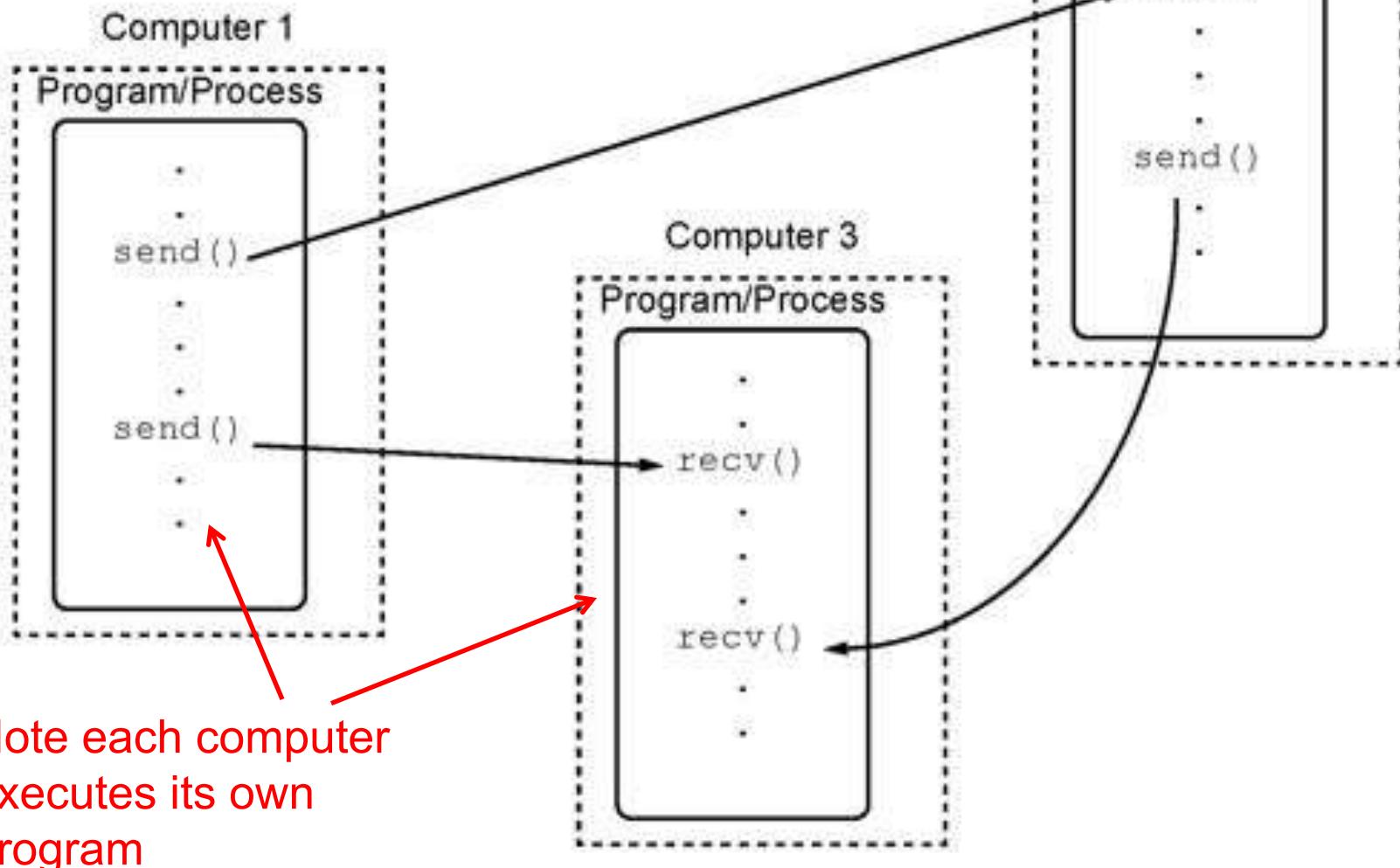
[**MPI_Reduce\(\)**](#)

- Collective computation (sum,min,max, ...)

[**MPI_Barrier\(\)**](#)

- Synchronize processes

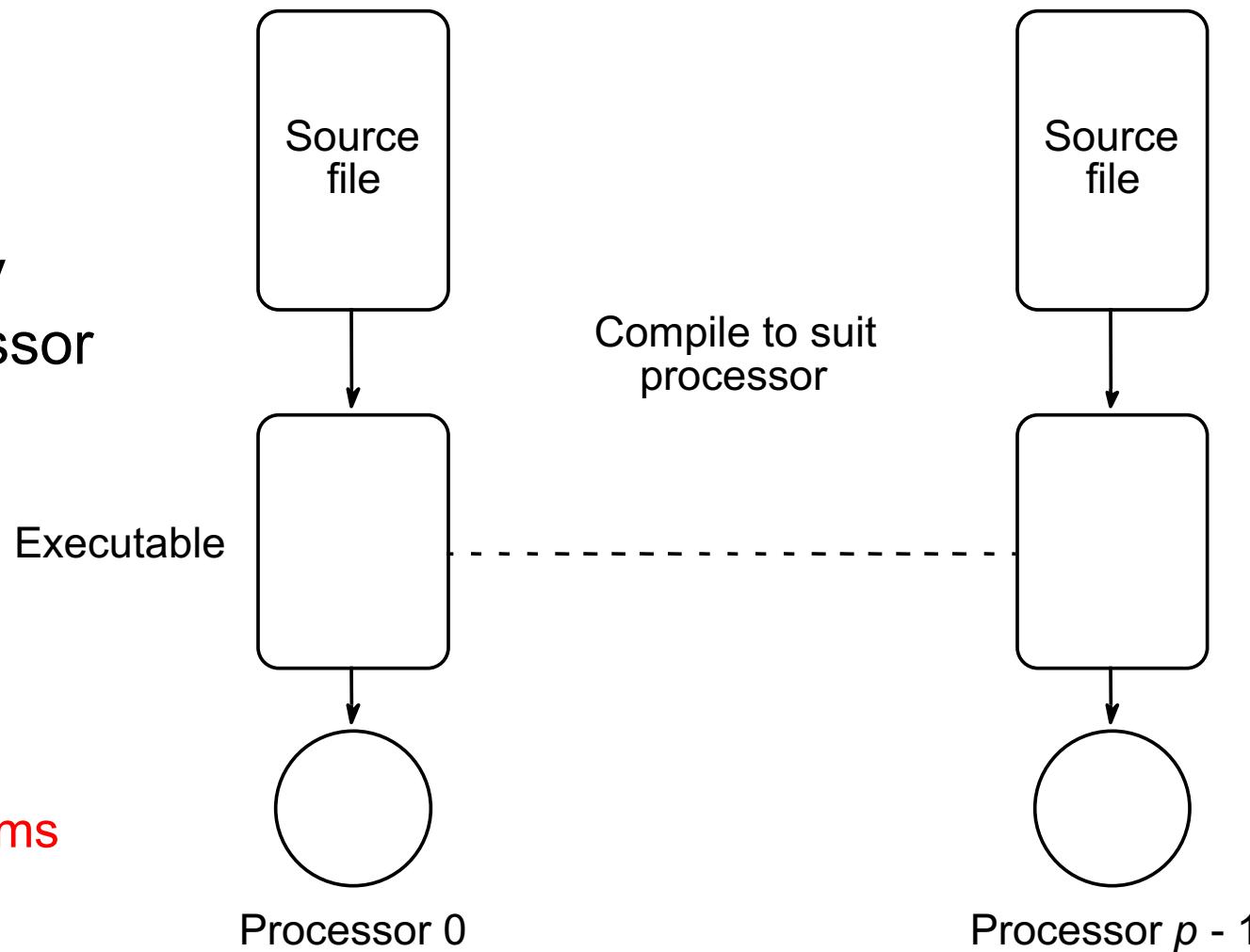
Message passing concept using library routines



Creating processes for execution on different computers

1. Multiple program, multiple data (MPMD) model

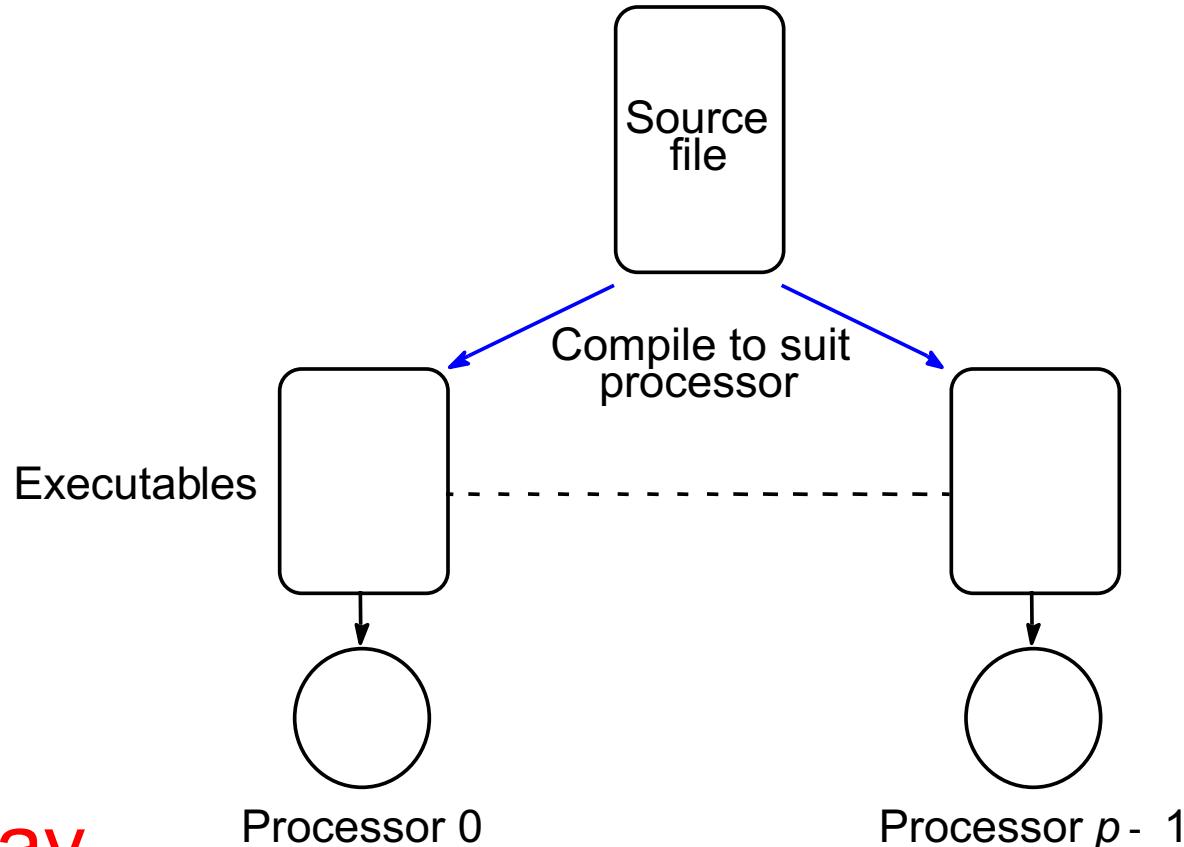
- Different programs executed by each processor



2. Single Program Multiple Data (SPMD) model

- Same program executed by each processor
- Control statements select different parts for each processor to execute.

Usual MPI way



Starting processes

Static process creation: All processes are specified before execution and system will execute a fixed number of processes. All executables started together.

Dynamic process creation: Processes created from within an executing process (fork)

Static process creation the normal MPI way.

Possible to dynamically start processes from within an executing process (fork) in MPI-2, which might find applicability if do not initially know how many processes needed.

MPI program structure

```
int main(int argc, char **argv) {
```

```
    MPI_Init(&argc, &argv);
```

```
    // Code executed by all processes
```

```
    MPI_Finalize();
```

```
}
```

Takes command line arguments, which includes the number of processes to use, see later.

In MPI, processes within a defined “**communicating group**” given a number called a **rank** starting from zero onwards.

Program uses control constructs, typically IF statements, to direct processes to perform specific actions.

Example

```
if (rank == 0) ... /* do this */;  
if (rank == 1) ... /* do this */;  
.  
:  
.
```

Master-Slave approach

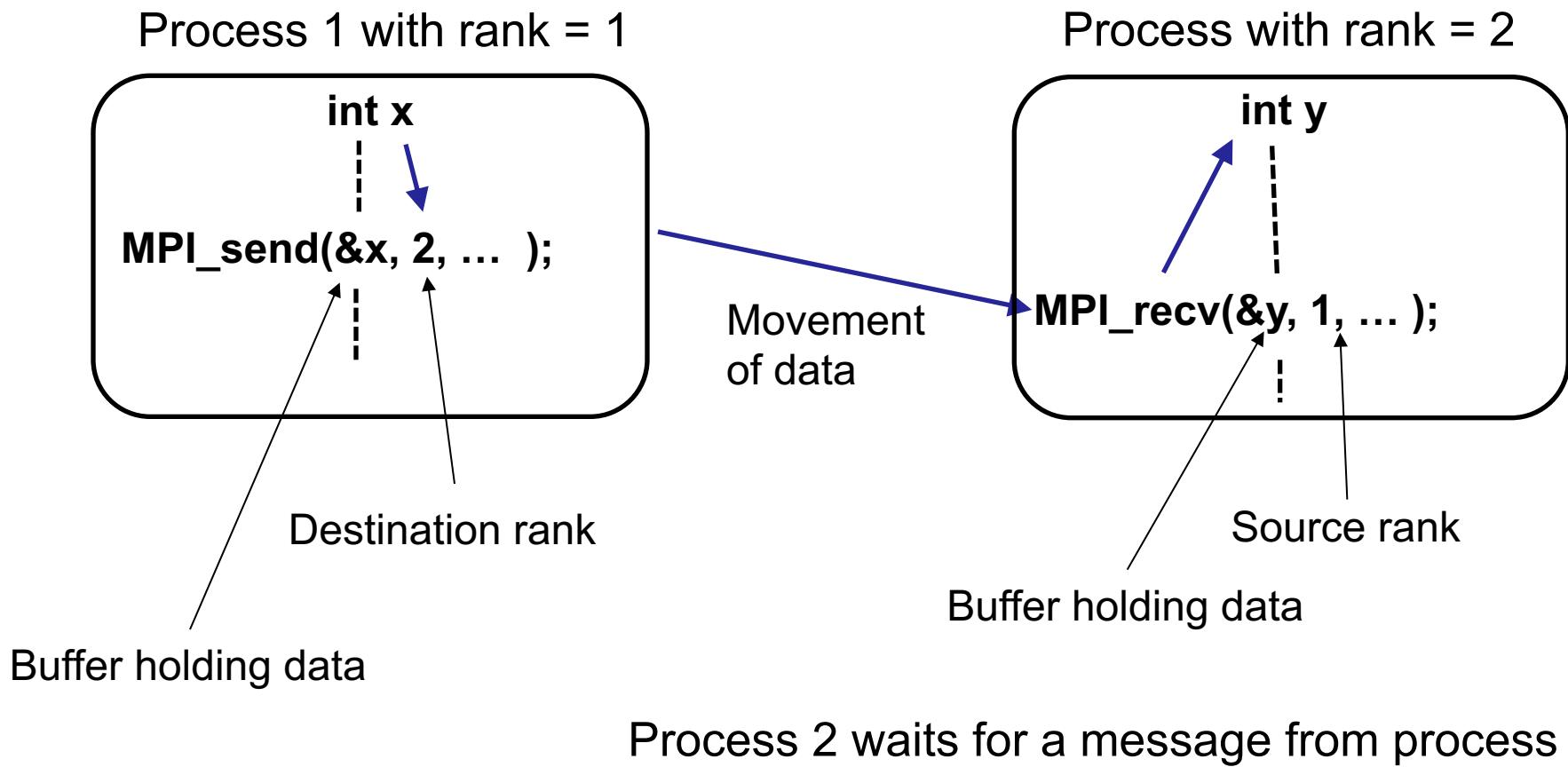
Usually computation constructed as a master-slave model

One process (the master), performs one set of actions and all the other processes (the slaves) perform identical actions although on different data, i.e.

```
if (rank == 0) ... /* master do this */;  
else ...           /* all slaves do this */;
```

MPI point-to-point message passing using MPI_send() and MPI_recv() library calls

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



Semantics of MPI_Send() and MPI_Recv()

Called **blocking**, which means in MPI that routine waits until all its local actions within process have taken place before returning.

After returning, any local variables used can be altered without affecting message transfer but not before.

MPI_Send() – When returns, message may not reached its destination but process can continue in the knowledge that message safely on its way.

MPI_Recv() – Returns when message received and data collected. Will cause process to stall until message received.

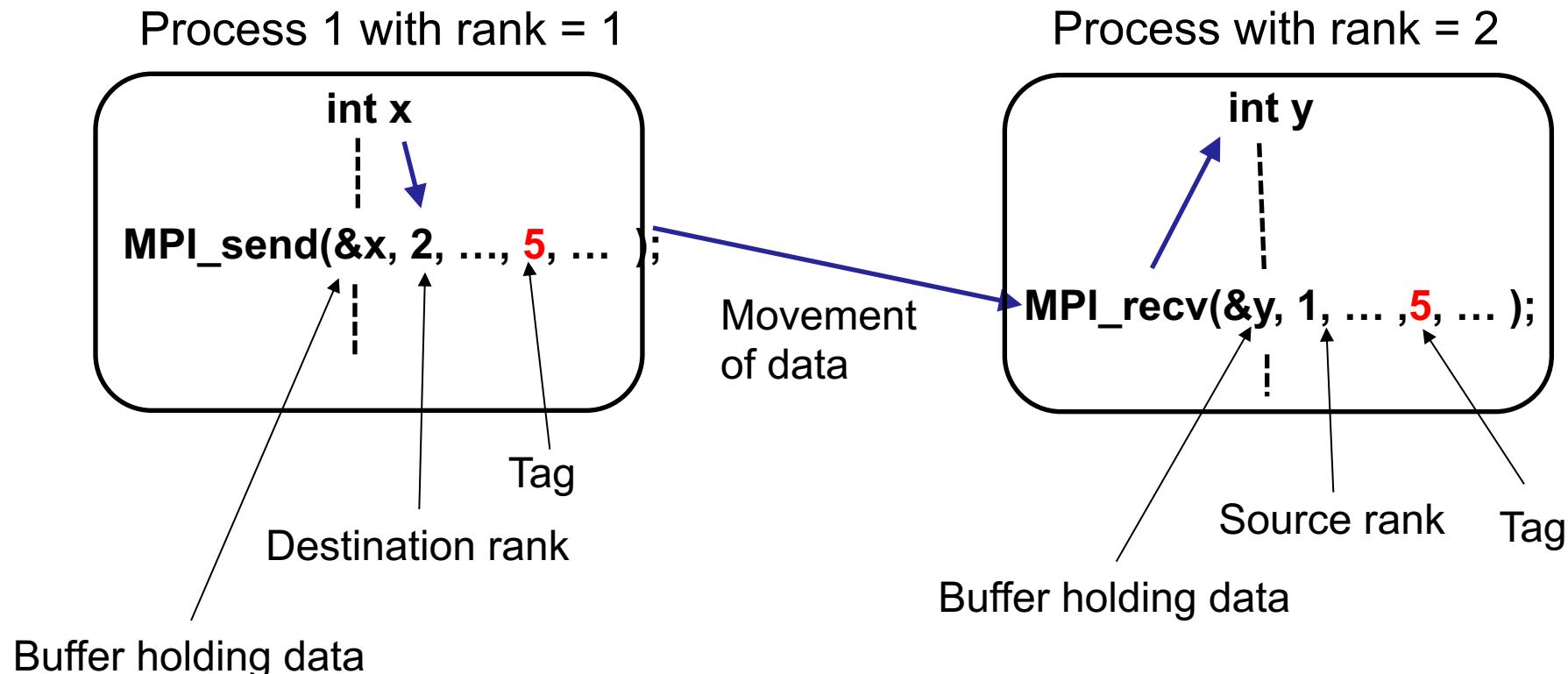
Other versions of MPI_Send() and MPI_Recv() have different semantics.

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 used. Then recv() will match with any send().

Message Tag Example

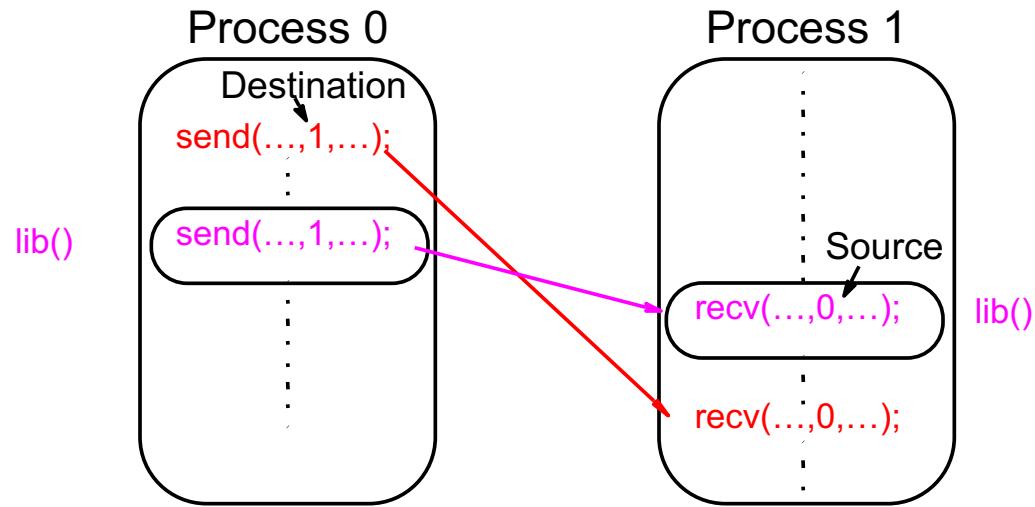
To send a message, x, from a source process, 1, with message tag 5 to a destination process, 2, and assign to y:



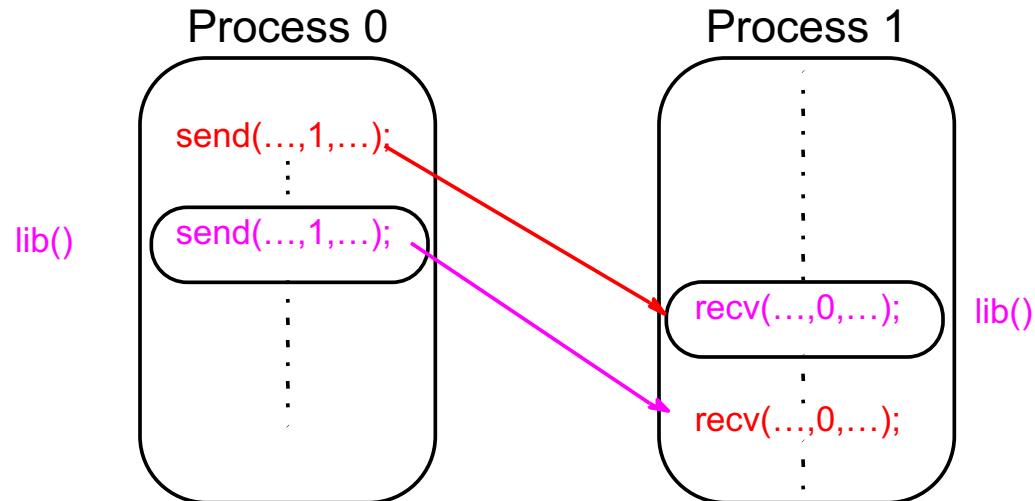
Process 2 waits for a message from process 1 with a tag of 5

Unsafe message passing - Example

(a) Intended behavior



(b) Possible behavior



Tags alone will not fix this as the same tag numbers might be used.

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.
- ***Process rank is a “rank” in a particular communicator.***

Note: **Intracommunicator** – for communicating within a single group of processes.

Intercommunicator - for communicating within two or more groups of processes

Default Communicator

MPI_COMM_WORLD

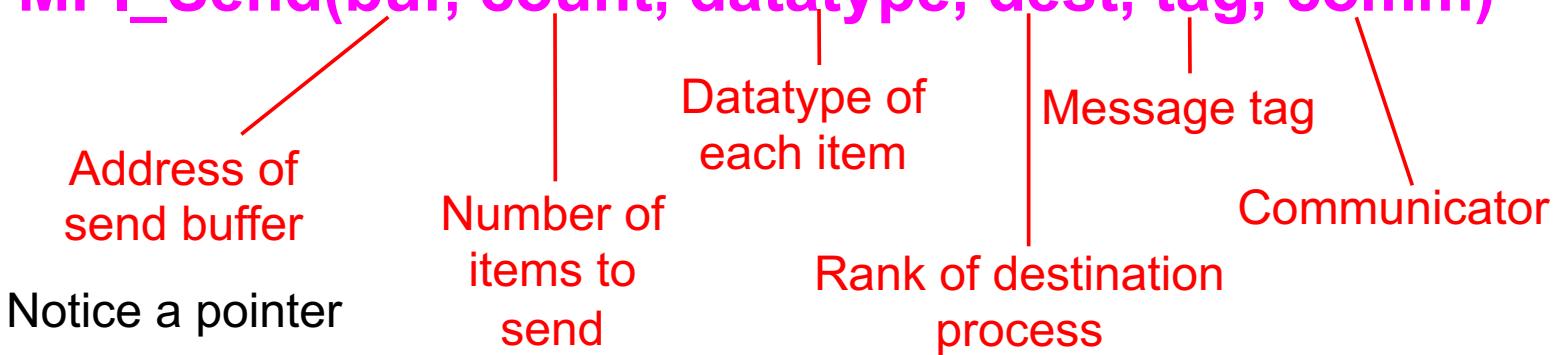
- Exists as first communicator for all processes existing in the application.
- Process rank in MPI_COMM_World obtained from:

```
MPI_Comm_rank (MPI_COMM_WORLD , &myrank) ;
```

- A set of MPI routines exists for forming additional communicators although we will not use them.

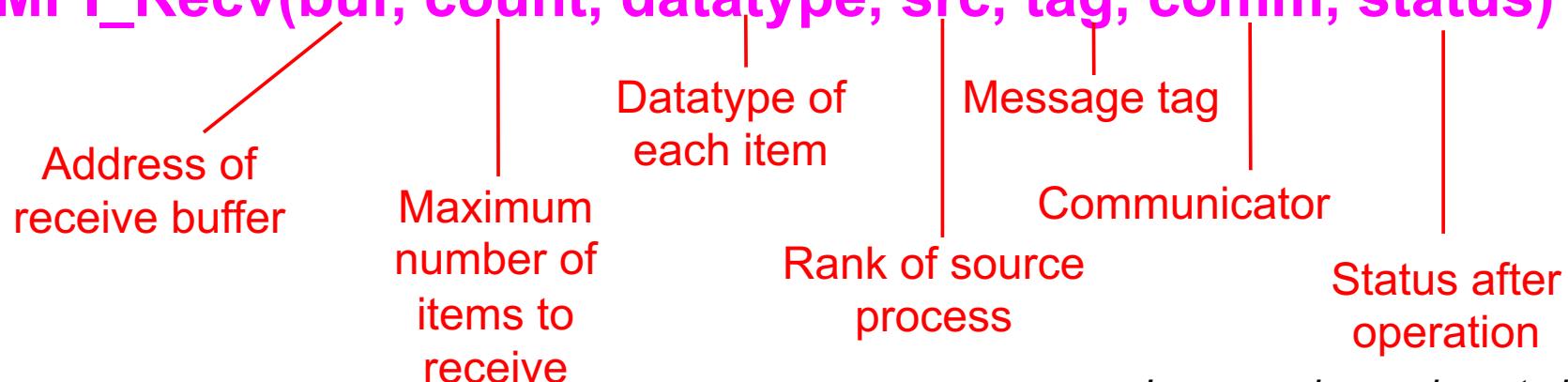
Parameters of blocking send

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



Parameters of blocking receive

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



Usually send and recv counts are the same.

In our code we do not check status but good programming practice to do so.

MPI Datatypes

(defined in mpi.h)

MPI datatypes
MPI_BYTE
MPI_PACKED
MPI_CHAR
MPI_SHORT
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
MPI_LONG_DOUBLE
MPI_UNSIGNED_CHAR

Wild cards -- any source or tag

- In MPI_Recv(), source can be **MPI_ANY_SOURCE** and tag can be **MPI_ANY_TAG**
- Cause MPI_Recv() to take any message destined for current process regardless of source and/or tag.

Example

```
MPI_Recv(message,256,MPI_CHAR, MPI_ANY_SOURCE,  
        MPI_ANY_TAG,MPI_COMM_WORLD, &status);
```

Program Examples

To send an integer x from process 0 to process 1 and assign to y.

```
int x, y; //all processes have their own copies of x and y

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);      // find rank

if (myrank == 0) {
    MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
    MPI_Recv(&y, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Another version

To send an integer x from process 0 to process 1 and assign to y.

```
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 y;
    MPI_Recv(&y, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

What is the difference?

Sample MPI Hello World program

```
#include <stddef.h>
#include <stdlib.h>
#include "mpi.h"
main(int argc, char **argv ) {
    char message[20];
    int i,rank, size, type=99;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    if(rank == 0) {
        strcpy(message, "Hello, world");
        for (i=1; i<size; i++)
            MPI_Send(message,13,MPI_CHAR,i,type,MPI_COMM_WORLD);
    } else
        MPI_Recv(message,20,MPI_CHAR,0,type,MPI_COMM_WORLD,&status);
    printf( "Message from process =%d : %.13s\n", rank,message);
    MPI_Finalize();
}
```

Program sends message “Hello World” from master process (rank = 0) to each of the other processes (rank != 0). Then, all processes execute a `println` statement.

In MPI, standard output automatically redirected from remote computers to the user’s console (thankfully!) so final result on console will be

Message from process =1 : Hello, world

Message from process =0 : Hello, world

Message from process =2 : Hello, world

Message from process =3 : Hello, world

...

except that the order of messages might be different but is unlikely to be in ascending order of process ID; it will depend upon how the processes are scheduled.

Another Example (array)

```
int array[100];
```

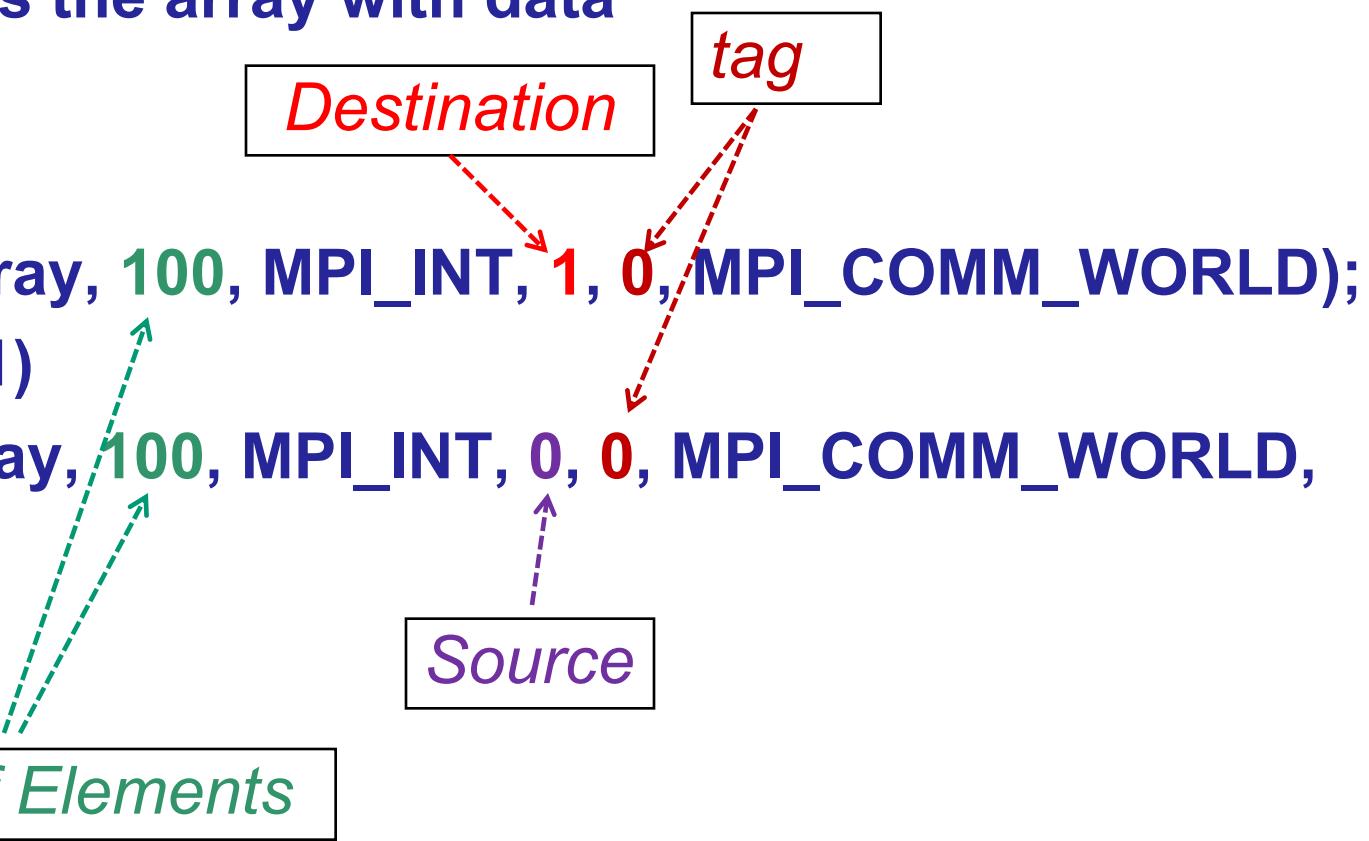
... // rank 0 fills the array with data

```
if (rank == 0)
```

```
    MPI_Send (array, 100, MPI_INT, 1, 0, MPI_COMM_WORLD);
```

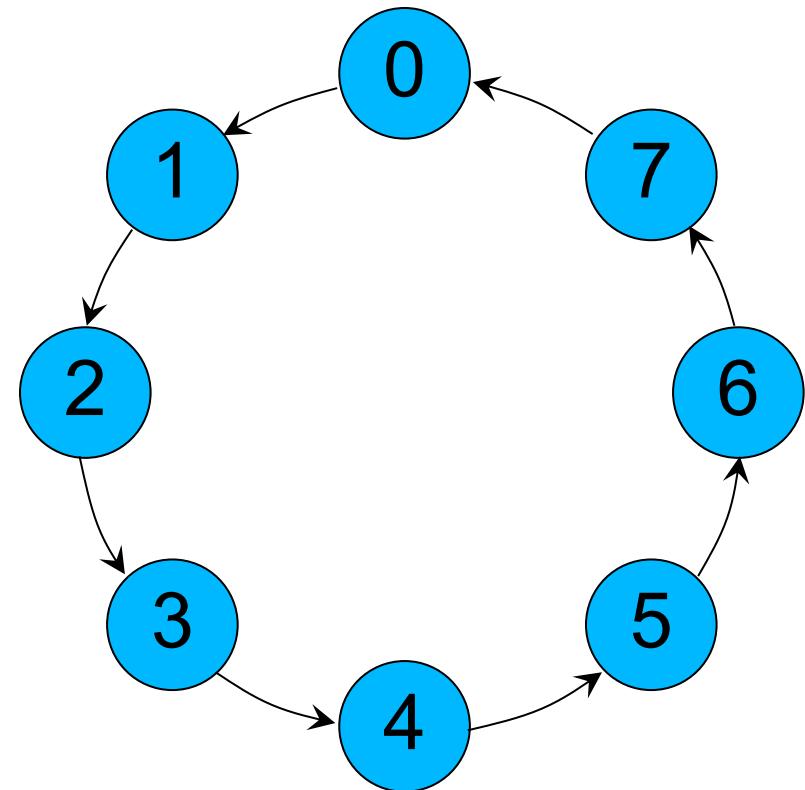
```
else if (rank == 1)
```

```
    MPI_Recv(array, 100, MPI_INT, 0, 0, MPI_COMM_WORLD,  
&status);
```



Another Example (Ring)

- Each process (excepts the master) receives a token from the process with rank 1 less than its own rank.
- Then each process increments the token by 2 and sends it to the next process (with rank 1 more than its own).
- The last process sends the token to the master



Question: Do we have pattern for this?

Ring Example

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

int main (int argc, char *argv[]) {
    int token, NP, myrank;
    MPI_Status status;

    MPI_Init (&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &NP);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

Ring Example continued

```
if (myrank == 0) {  
  
    token = -1; // Master sets initial value before sending.  
  
else {  
  
    // Everyone except master receives from process 1 less  
    // than its own rank.  
  
    MPI_Recv(&token, 1, MPI_INT, myrank - 1, 0,  
             MPI_COMM_WORLD, &status);  
    printf("Process %d received token %d from process %d\n",  
          myrank, token, myrank - 1);  
}
```

Ring Example continued

```
// all processes

token += 2;                      // add 2 to token before sending it
MPI_Send(&token, 1, MPI_INT, (myrank + 1) % NP, 0,
         MPI_COMM_WORLD);

// Now process 0 can receive from the last process.
if (myrank == 0) {
    MPI_Recv(&token, 1, MPI_INT, NP - 1, 0,
              MPI_COMM_WORLD, &status);
    printf("Process %d received token %d from process %d\n",
           myrank, token, NP - 1);
}
MPI_Finalize();
```

Results (Ring)

Process 1 received token 1 from process 0

Process 2 received token 3 from process 1

Process 3 received token 5 from process 2

Process 4 received token 7 from process 3

Process 5 received token 9 from process 4

Process 6 received token 11 from process 5

Process 7 received token 13 from process 6

Process 0 received token 15 from process 7

Matching up sends and recvs

- Notice in code how you have to be very careful matching up send's and recv's.
- Every send must have matching recv.
- The sends return after local actions complete but the recv will wait for the message so easy to get deadlock if written wrong
- Pre-implemented patterns are designed to avoid deadlock. We will look at deadlock again

Measuring Execution Time

MPI provides the routine **`MPI_Wtime()`** for returning time (in seconds) from some point in the past.

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

```
double start_time, end_time, exe_time;  
L1: start_time = MPI_Wtime(); // record time  
:  
:  
L2: end_time = MPI_Wtime(); // record time  
  
exe_time = end_time - start_time;
```

Using C time routines

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

```
L1: time(&t1);           // record time  
.  
. .  
L2: time(&t2);           // record time  
. .  
elapsed_Time = difftime(t2, t1); /*time=t2-t1*/  
  
printf("Elapsed time=%5.2f secs", elapsed_Time);
```

gettimeofday()

```
#include <sys/time.h>
double elapsed_time;
struct timeval tv1, tv2;
gettimeofday(&tv1, NULL);
...
gettimeofday(&tv2, NULL); } Measure time to execute
elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
               ((tv2.tv_usec - tv1.tv_usec) / 1000000.0); this section
```

Using `time()` or `gettimeofday()` routines may be useful if you want to compare with a sequential C version of the program with same libraries.

Compiling and executing MPI programs on the command line (without a scheduler)

Compiling/executing MPI program

MPI implementations provide scripts, **mpicc** and **mpiexec** for compiling and executing code (not part of original standard but now universal)

To compile MPI C programs:

```
mpicc -o prog prog.c
```

-o option to specify name of output file. Can be before or after program name. Many prefer after.

mpicc uses the gcc compiler adding the MPI libraries so all options with the gcc can be used.

To execute MPI program:

```
mpirun -n no_procs prog
```

A positive integer specifying number of processes

Notice number of processes determined at execution time, so same code can be run with different numbers of processes



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Message-Passing Computing

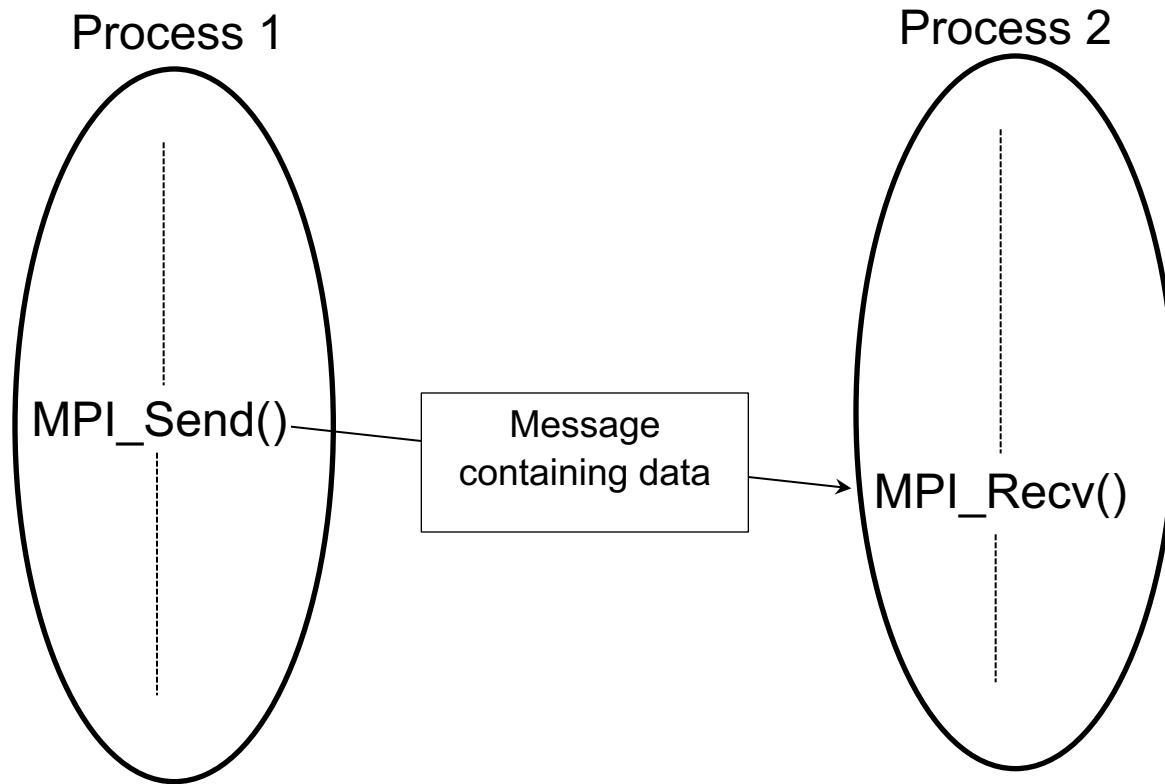
Message passing patterns and collective
MPI routines

Message passing patterns

Point-to-point Send-Receive



**Implementation of
send-receive
pattern with explicit
MPI send and
receive routines**



Collective message-passing routines

Involve multiple processes.

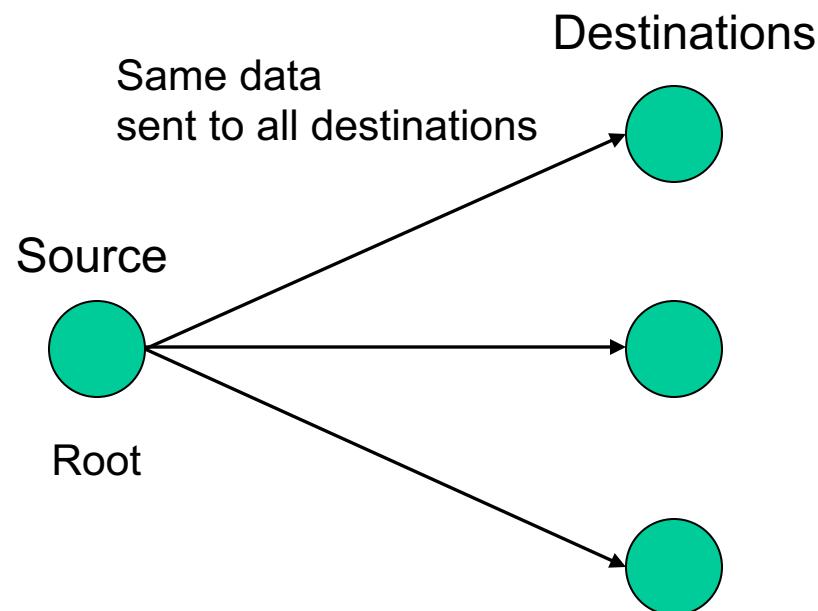
Implements commonly appearing groups of send-receive patterns efficiently

One process (the **root**) is the source of data sent to other processes, or destination of data sent from other processes.

Broadcast pattern

Sends same data to each of a group of processes

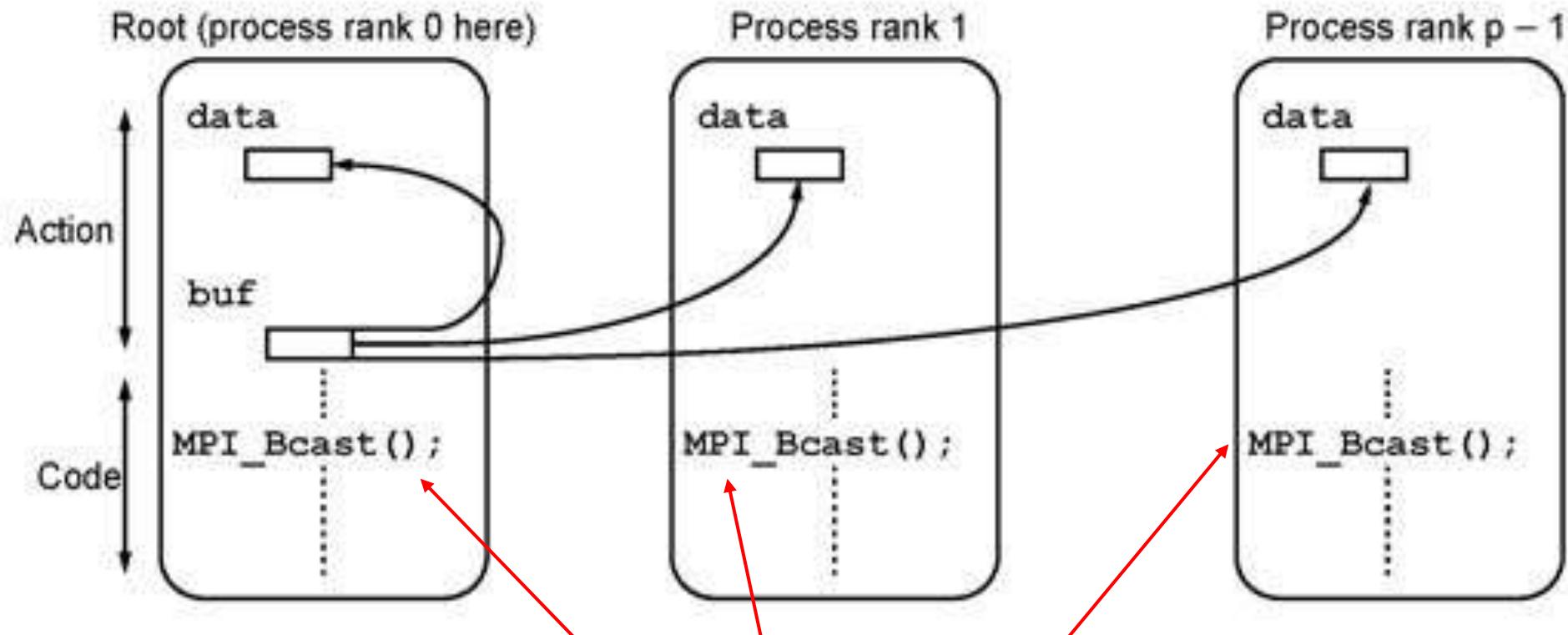
A common pattern to get same data to all processes, especially at the beginning of a computation



Note: Patterns given do not mean the implementation does them as shown. Only the final result is the same in any parallel implementation. Patterns do not describe the implementation.

MPI broadcast operation

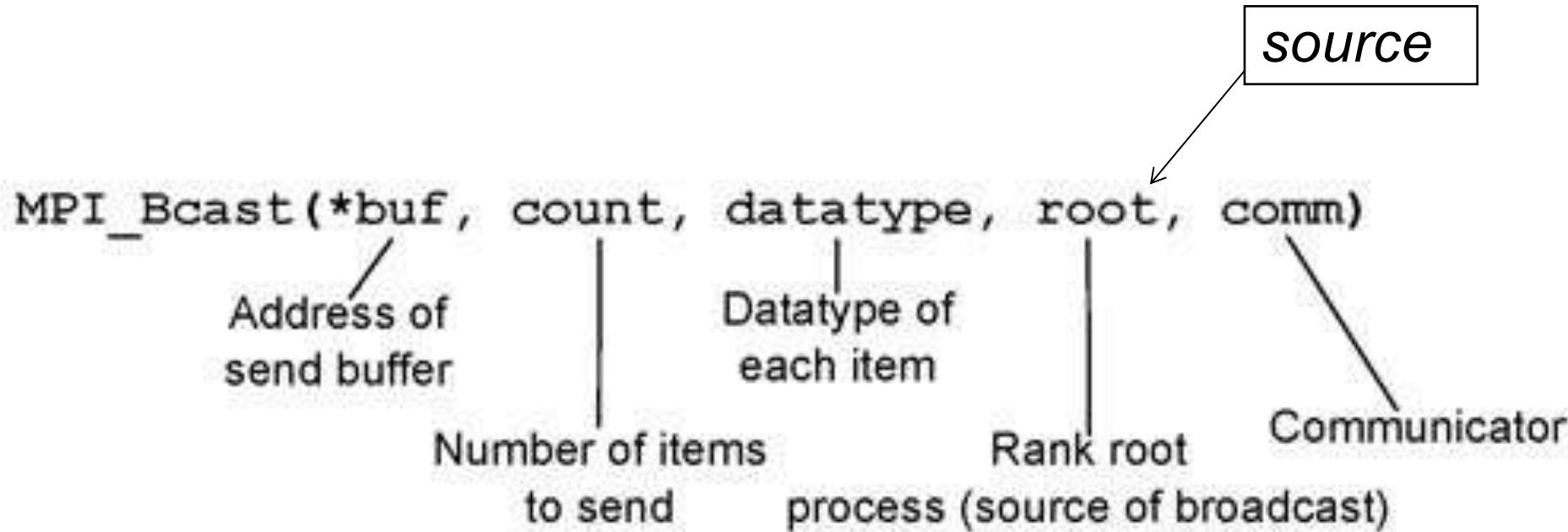
Sending same message to all processes in communicator



Notice same routine called by each process, with same parameters.

MPI processes usually execute the same program so this is a handy construction.

MPI_Bcast parameters



All processes in the Communicator must call the `MPI_Bcast` with the same parameters

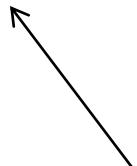
Notice that there is no tag.

Broadcast example

Suppose we wanted to broadcast an array to all processes

```
int main(int argc, char *argv[]) {  
    MPI_Status status; // MPI variables  
    int rank;  
    double A[N][N];  
    MPI_Init(&argc, &argv);      // Start MPI  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    if (rank == 0) {            // initialize A  
        for (i = 0; i < N; i++)  
            for (j = 0; j < N; j++)  
                A[i][j] = i + j;  
    }  
    ...  
    MPI_Bcast(A, N*N,MPI_DOUBLE,0,MPI_COMM_WORLD); // Broadcast A  
    ...  
    MPI_Finalize();  
    return 0;  
}
```

Number of elements in total

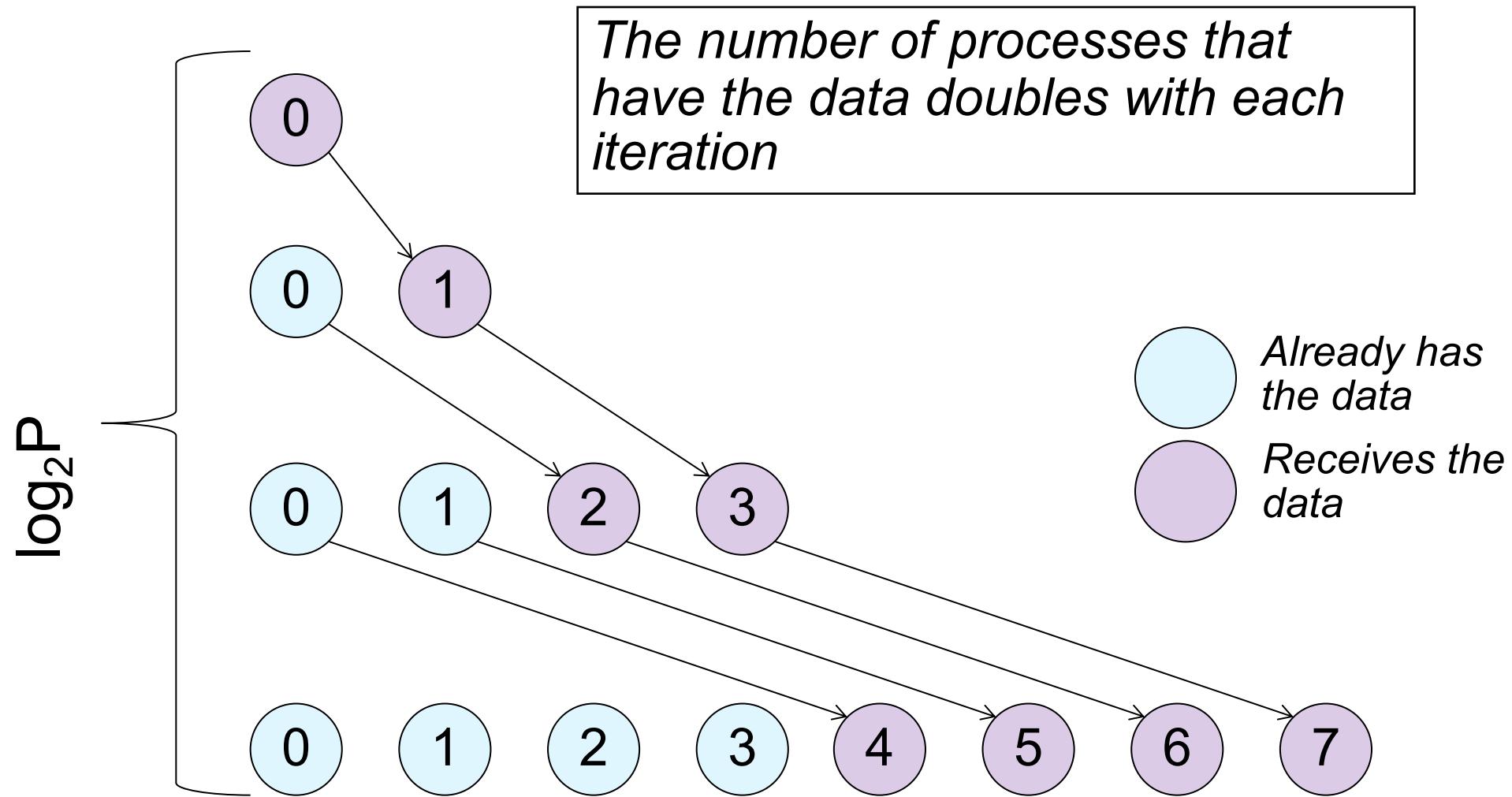


Creating a broadcast with individual sends and receives

```
if(rank == 0) {  
    for (i=1; i < P; i++)  
        MPI_Send(buf,N,MPI_INT,i,tag, MPI_COMM_WORLD);  
} else  
    MPI_Recv(buf,N,MPI_INT,0,tag,MPI_COMM_WORLD,&status);
```

Complexity of doing that is $O(N * P)$, where the number of bytes in the message is N and there are P processors.

Likely MPI_Bcast implementation



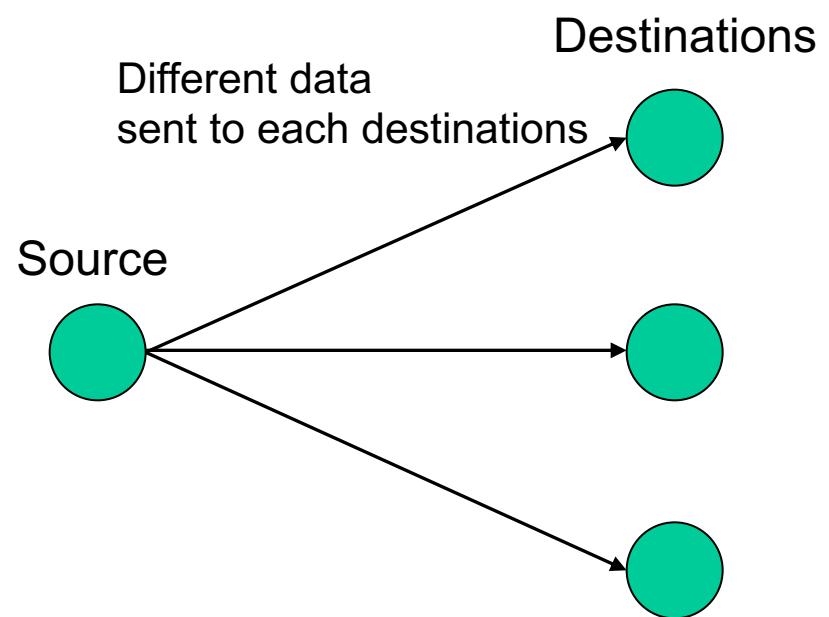
Complexity of broadcast is $O(N * \log_2(P))$.

Scatter Pattern

Distributes a collection of data items to a group of processes

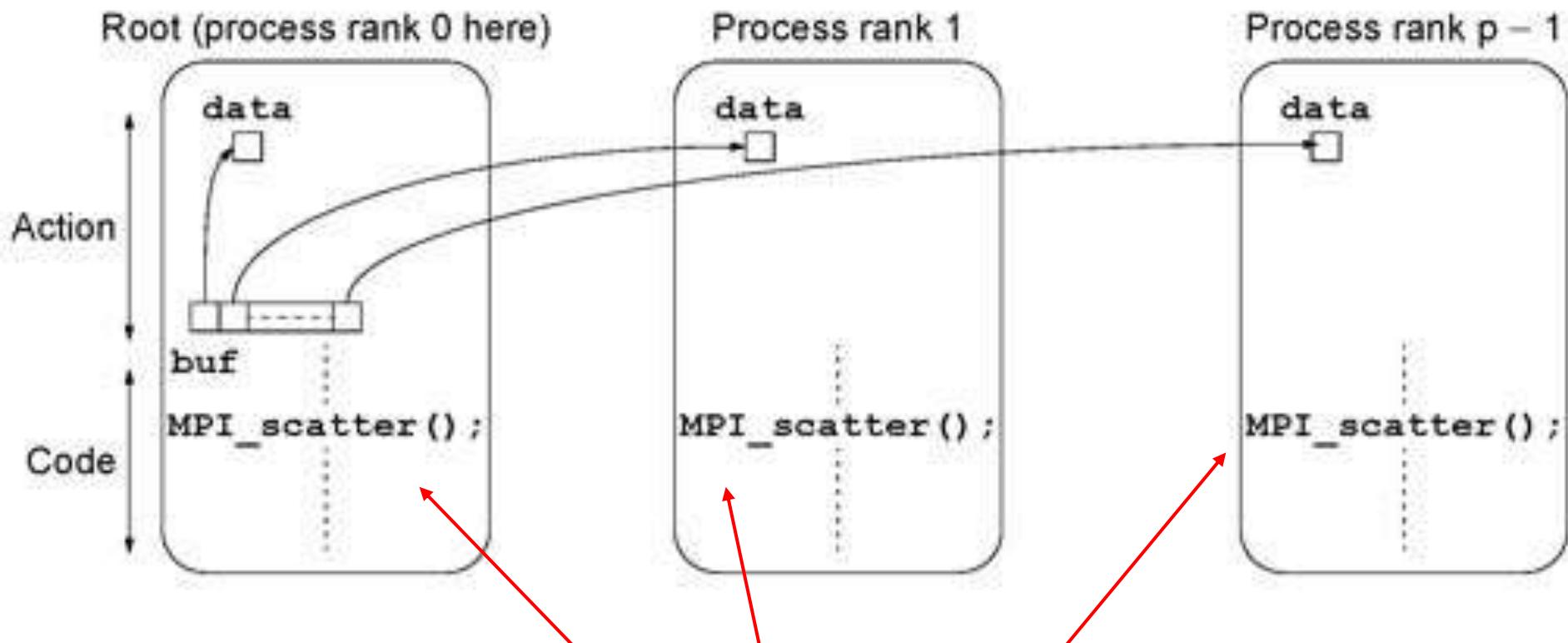
A common pattern to get data to all processes

Usually data sent are parts of an array



Basic MPI scatter operation

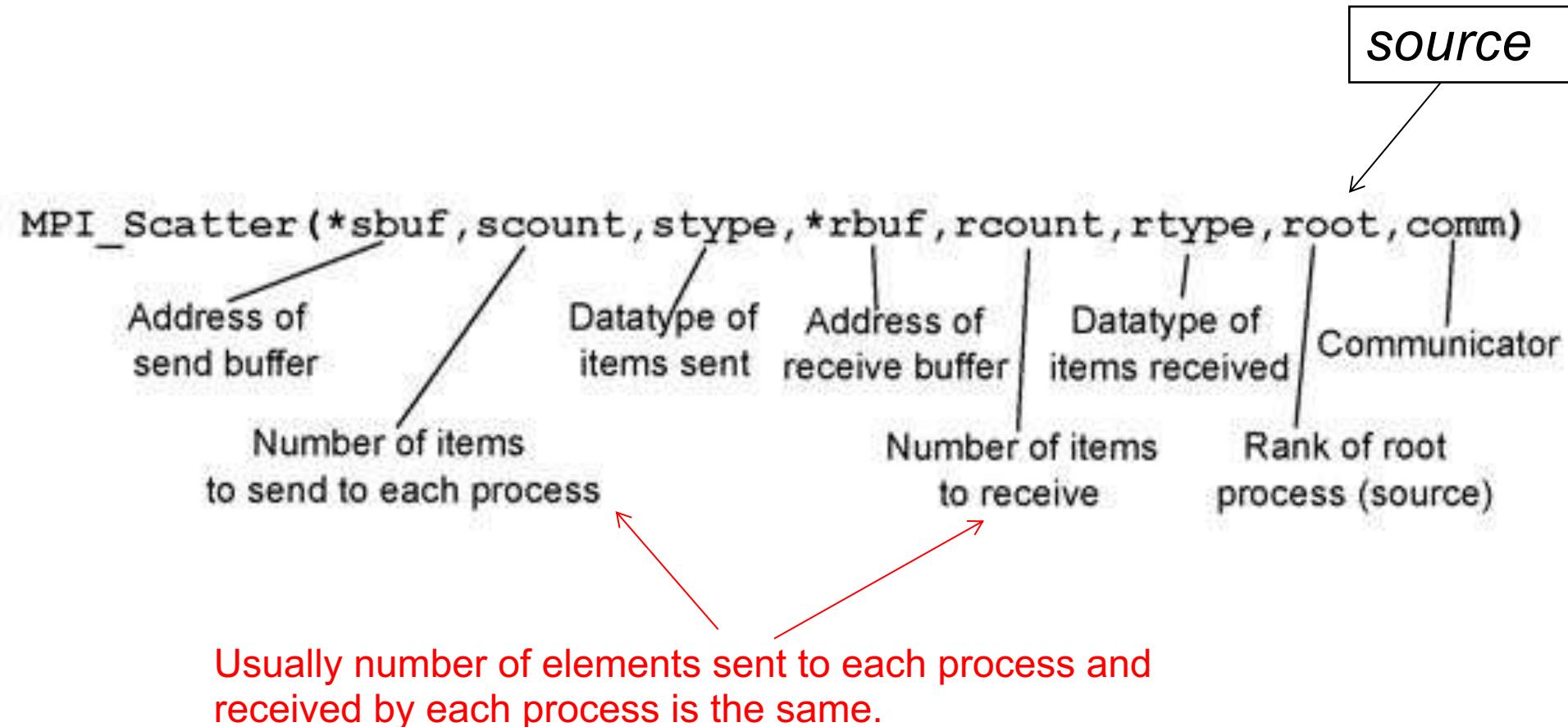
Sending one or more contiguous elements of an array in root process to a separate process.



Notice same routine called by each process, with same parameters.

MPI processes usually execute the same program so this is a handy construction.

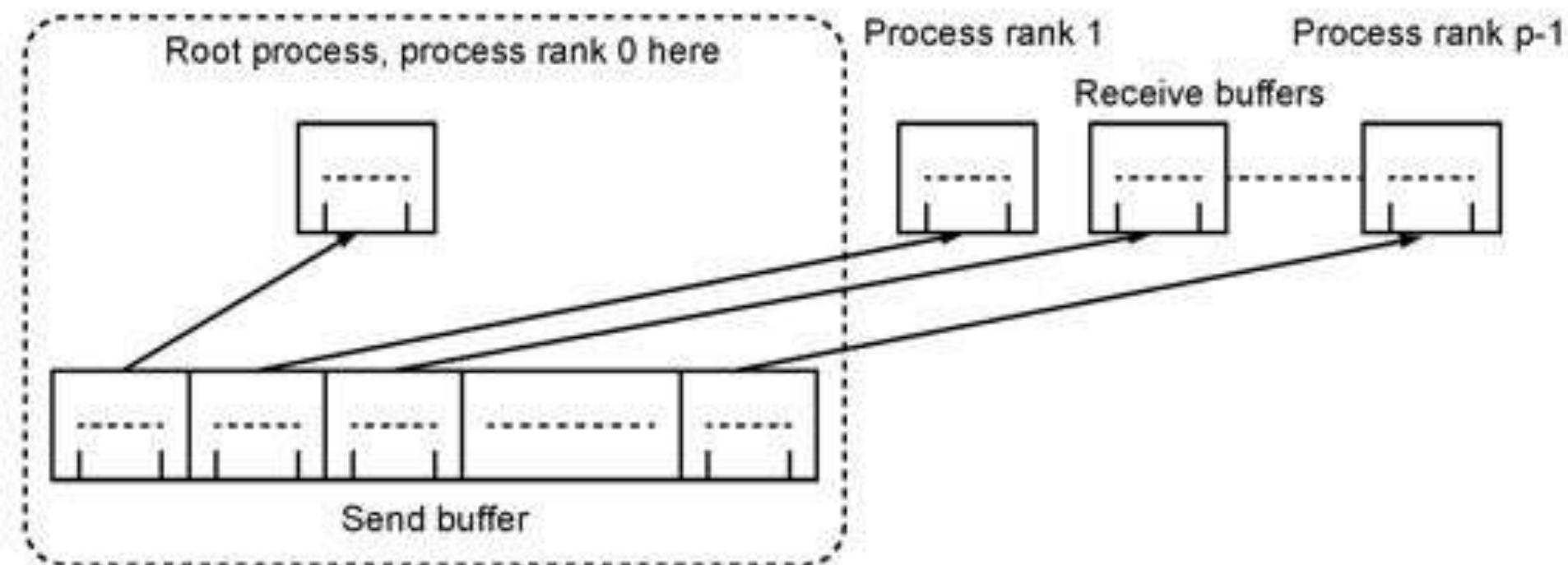
MPI scatter parameters



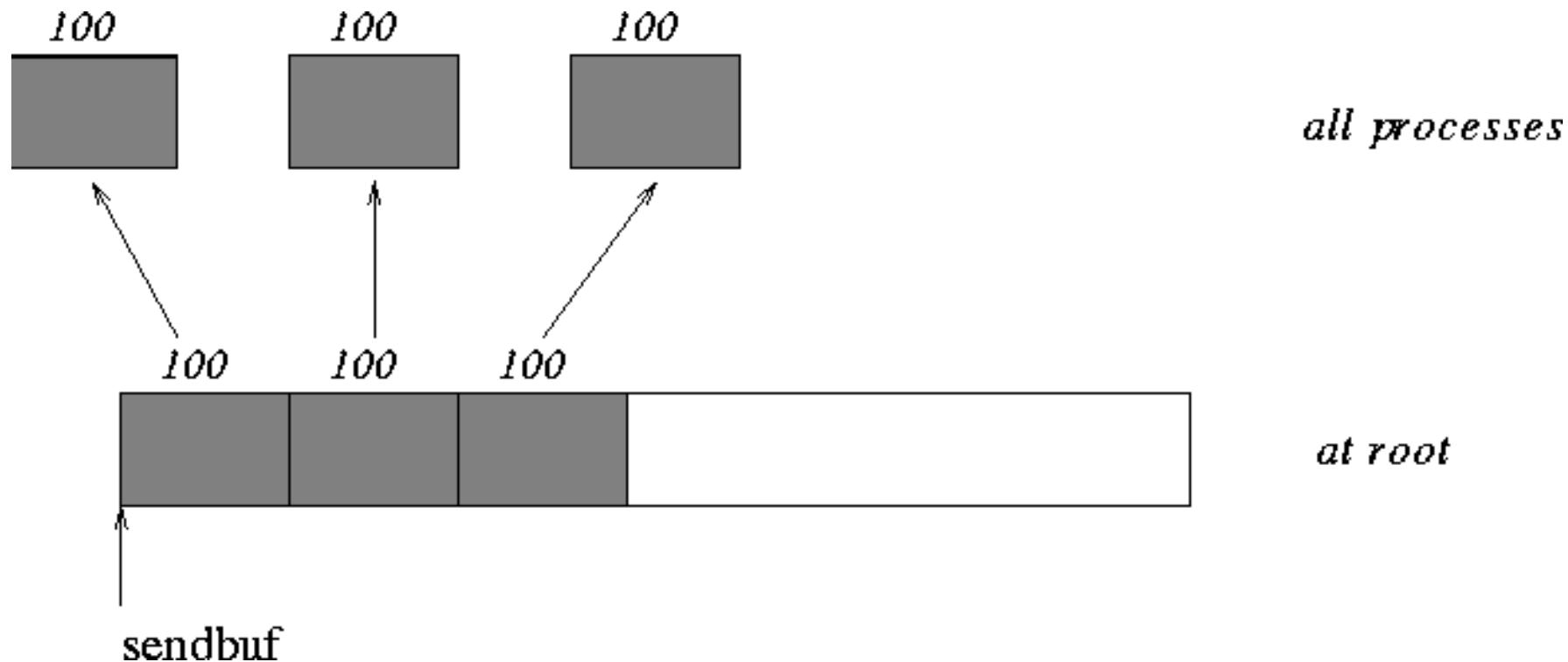
All processes in the Communicator must call the MPI_Scatter with the same parameters

Notice that there is no tag.

Scattering contiguous groups of elements to each process



Scatter Example



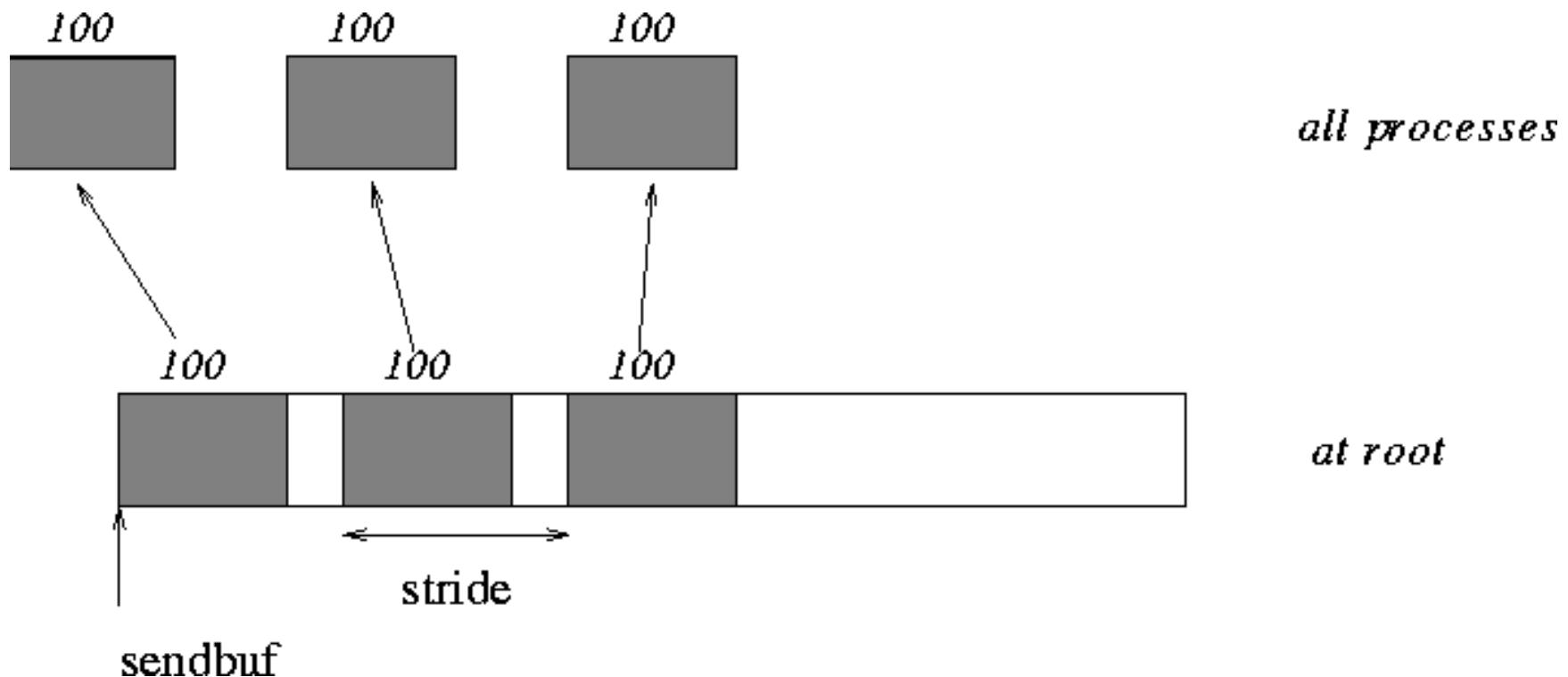
Example

In the following code, size of send buffer is given by 100 * <number of processes> and 100 contiguous elements are send to each process:

```
main (int argc, char *argv[]) {  
    int size, *sendbuf, recvbuf[100];          /* for each process */  
    MPI_Init(&argc, &argv);                  /* initialize MPI */  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    sendbuf = (int *)malloc(size*100*sizeof(int));  
  
    .  
  
    MPI_Scatter(sendbuf,100,MPI_INT,recvbuf,100,MPI_INT,0,  
               MPI_COMM_WORLD);  
  
    .  
  
    MPI_Finalize();                          /* terminate MPI */  
}
```

There is a version scatter called `MPI_Scatterv`, that can jump over parts of the array:

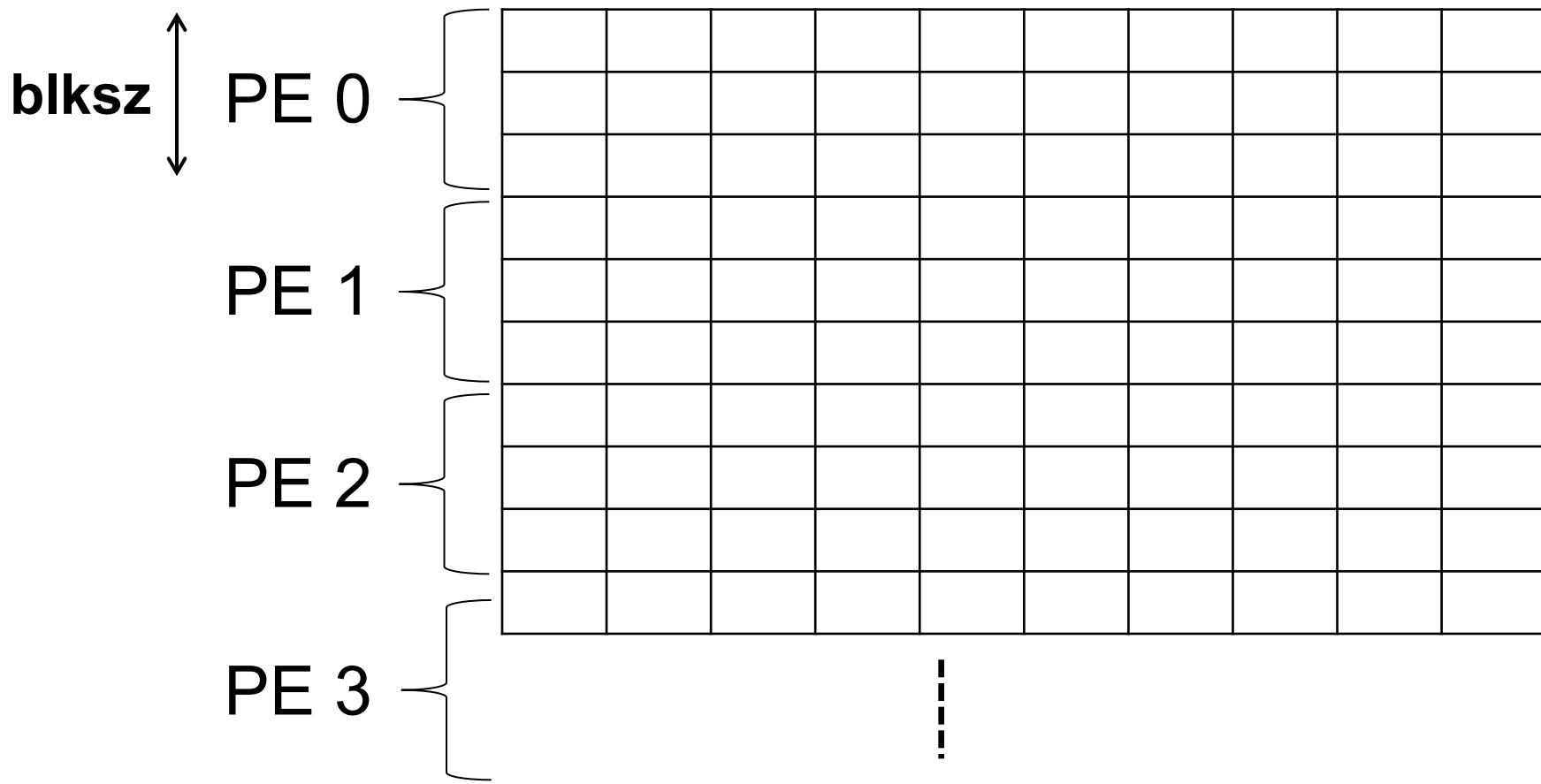
MPI_Scatterv Example



(source: <http://www mpi-forum.org>)

Scattering Rows of a Matrix

Since C stores multi-dimensional arrays in row-major order, scattering groups of rows of a matrix is easy



Scatter example

Scatter blksz number of rows of an array table[N][N] to each of P processes, where blksz = N/P.

...

```
#define N 8
```

```
int main( int argc, char **argv ) {
    int rank, P, i, j, blksz;
    int table[N][N], row[N][N];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &P );
    blksz = N/P;                                // N/P must be an integer
```

here

```
if (rank == 0) {
    ... Initialize table[][] with numbers (multiplication table)
}
```

Number of rows sent and received same

```
// All processors do this
```

```
MPI_Scatter (table, blksz*N, MPI_INT, row, blksz*N, MPI_INT, 0,
MPI_COMM_WORLD);
```

```
... // All processes print what they get
```

```
MPI_Finalize();
```

Output

Number of Processes= 8

```
<rank 0 initial>: N = 8 blksz = 1
<rank 0 initial>:   0   0   0   0   0   0   0   0
<rank 0 initial>:   0   1   2   3   4   5   6   7
<rank 0 initial>:   0   2   4   6   8   10  12  14
<rank 0 initial>:   0   3   6   9  12  15  18  21
<rank 0 initial>:   0   4   8  12  16  20  24  28
<rank 0 initial>:   0   5  10  15  20  25  30  35
<rank 0 initial>:   0   6  12  18  24  30  36  42
<rank 0 initial>:   0   7  14  21  28  35  42  49

<rank 0>:   0   0   0   0   0   0   0   0
<rank 1>:   0   1   2   3   4   5   6   7
<rank 2>:   0   2   4   6   8  10  12  14
<rank 3>:   0   3   6   9  12  15  18  21
<rank 4>:   0   4   8  12  16  20  24  28
<rank 5>:   0   5  10  15  20  25  30  35
<rank 6>:   0   6  12  18  24  30  36  42
<rank 7>:   0   7  14  21  28  35  42  49
```

Output

Number of Processes= 4

```
<rank 0 initial>: N = 8 blksz = 2
<rank 0 initial>:  0   0   0   0   0   0   0   0
<rank 0 initial>:  0   1   2   3   4   5   6   7
<rank 0 initial>:  0   2   4   6   8  10  12  14
<rank 0 initial>:  0   3   6   9  12  15  18  21
<rank 0 initial>:  0   4   8  12  16  20  24  28
<rank 0 initial>:  0   5  10  15  20  25  30  35
<rank 0 initial>:  0   6  12  18  24  30  36  42
<rank 0 initial>:  0   7  14  21  28  35  42  49

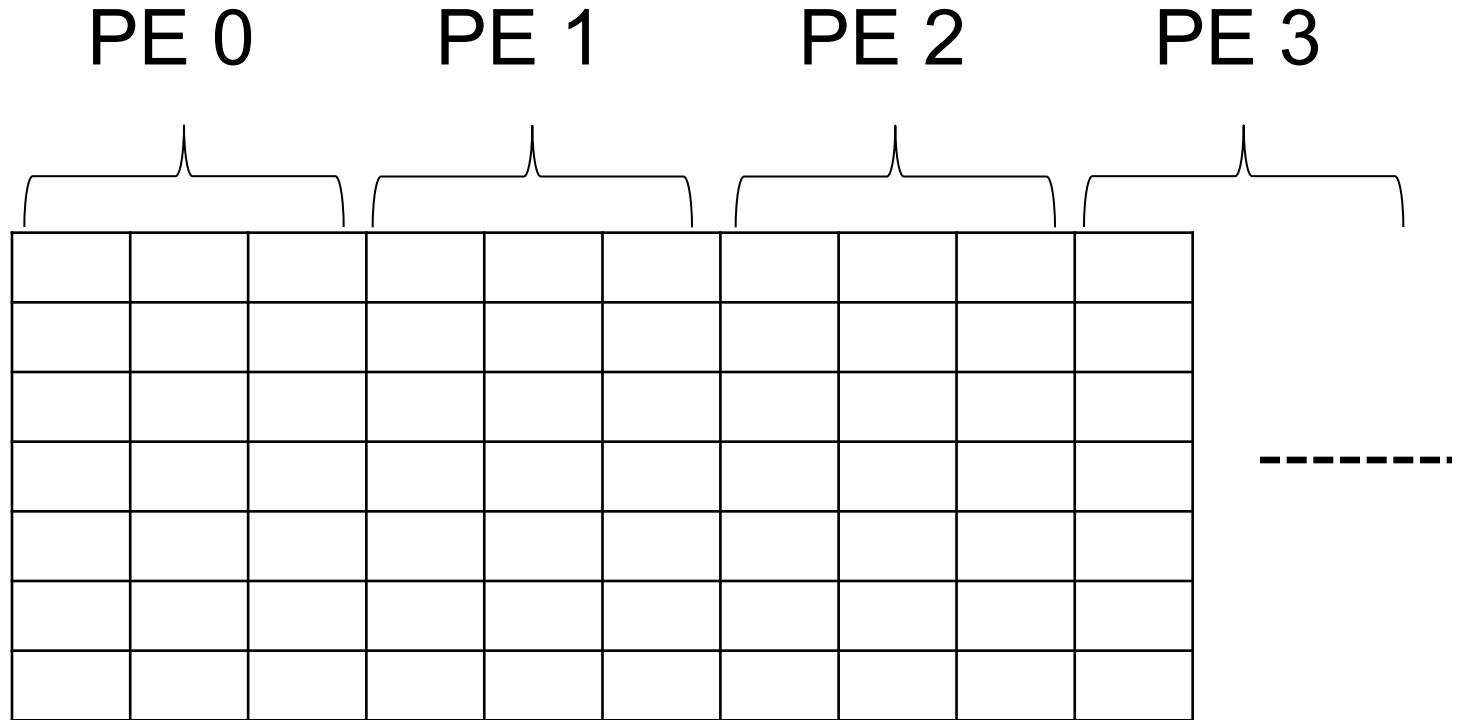
<rank 0>:  0   0   0   0   0   0   0   0
<rank 0>:  0   1   2   3   4   5   6   7
<rank 1>:  0   2   4   6   8  10  12  14
<rank 1>:  0   3   6   9  12  15  18  21
<rank 2>:  0   4   8  12  16  20  24  28
<rank 2>:  0   5  10  15  20  25  30  35
<rank 3>:  0   6  12  18  24  30  36  42
<rank 3>:  0   7  14  21  28  35  42  49
```

Generating readable output

- Can be tricky as cannot guarantee order of printf statements executed by different processes unless force order that processes execute.
- Usually have to add sleep statements (stdout flush and barriers alone do not necessarily work).

Scattering Columns of a Matrix

- What if we want to scatter columns?



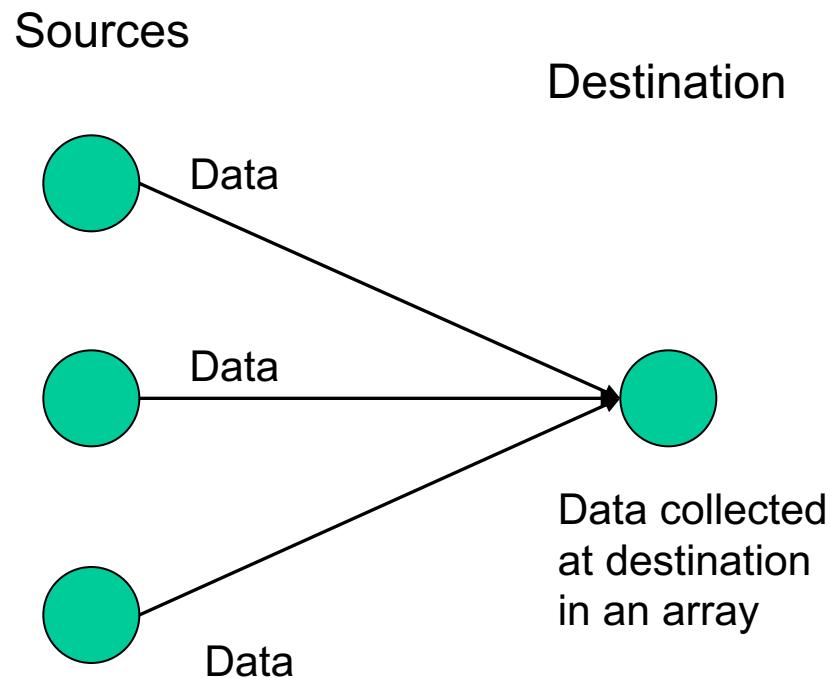
Scattering Columns of a Matrix

- The easiest solution would be to transpose the matrix, then scatter the rows (although transpose incurs an overhead especially for large matrices).

Gather Pattern

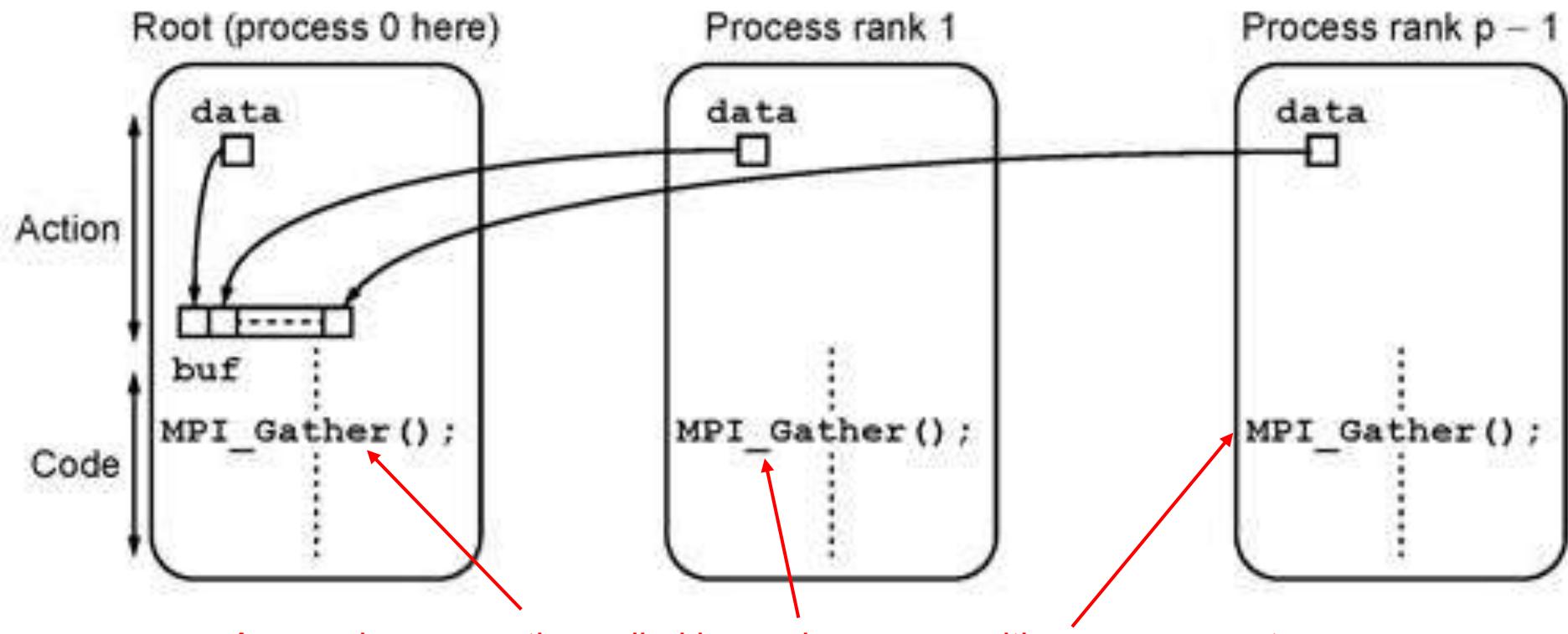
Essentially the reverse of a scatter. It receives data items from a group of processes

A common pattern especially at the end of a computation to collect results



MPI Gather

Having one process collect individual values from set of processes (includes itself).



Gather parameters

```
MPI_Gather(*sbuf, scount, stype, *rbuf, rcount, rtype, root, comm)
```

Address of send buffer Datatype of items sent Address of receive buffer Datatype of items received Communicator
Number of items to send to root process (from each process) Number of items to receive (in any single receive) Rank of root process (destination)

Red text note: Usually number of elements sent to each process and received by each process is the same.

All processes in the Communicator must call the MPI_Gather with the same parameters

Gather Example

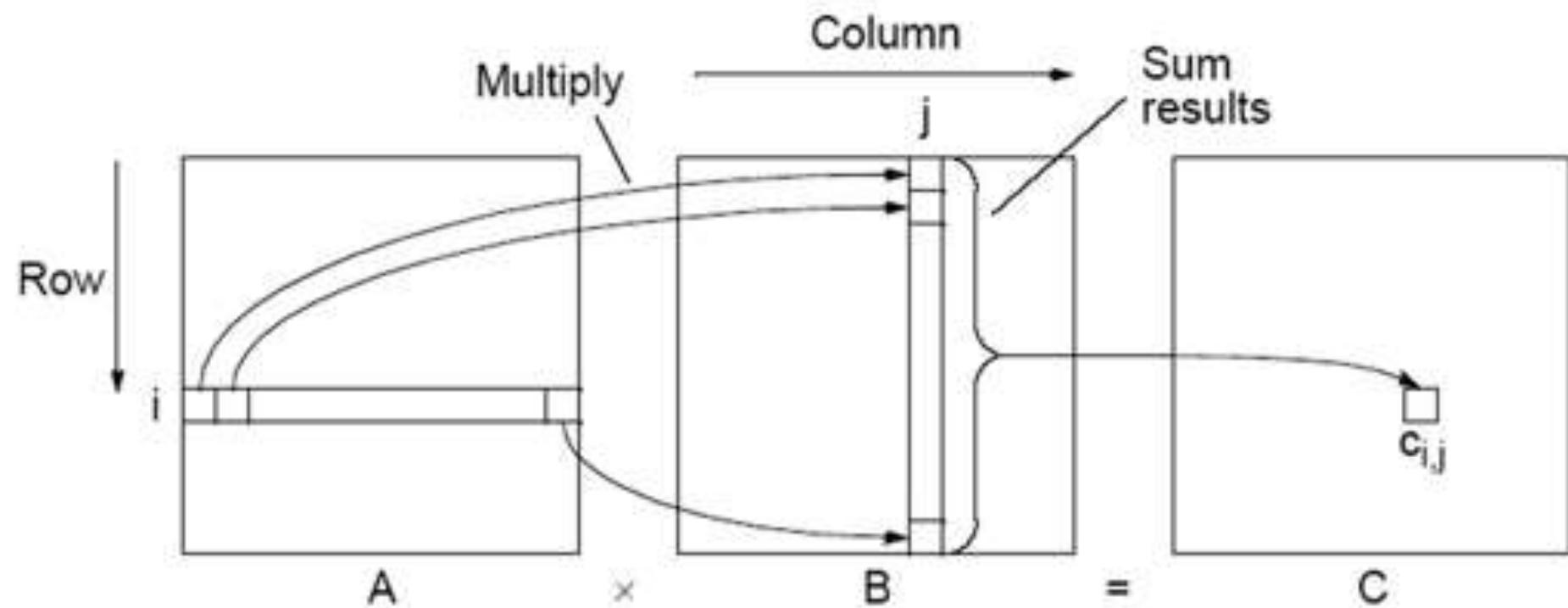
To gather 10 data elements from each process into process 0, using dynamically allocated memory in root process:

```
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)); /*alloc. mem*/  
}  
  
MPI_Gather(data,10,MPI_INT,buf,10,MPI_INT,0,MPI_COMM_WORLD) ;  
...  
  
MPI_Gather() gathers from all processes, including root.
```

Example using scatter, broadcast and gather

Matrix Multiplication, $C = A * B$

Multiplication of two matrices, **A** and **B**, produces matrix **C**



Parallelizing Matrix Multiplication

Assume throughout that matrices square ($n \times n$ matrices).

Sequential code to compute $\mathbf{A} \times \mathbf{B}$ could simply be

```
for (i = 0; i < n; i++)          // for each row of A
    for (j = 0; j < n; j++) {      // for each column of B
        c[i][j] = 0;
        for (k = 0; k < n; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

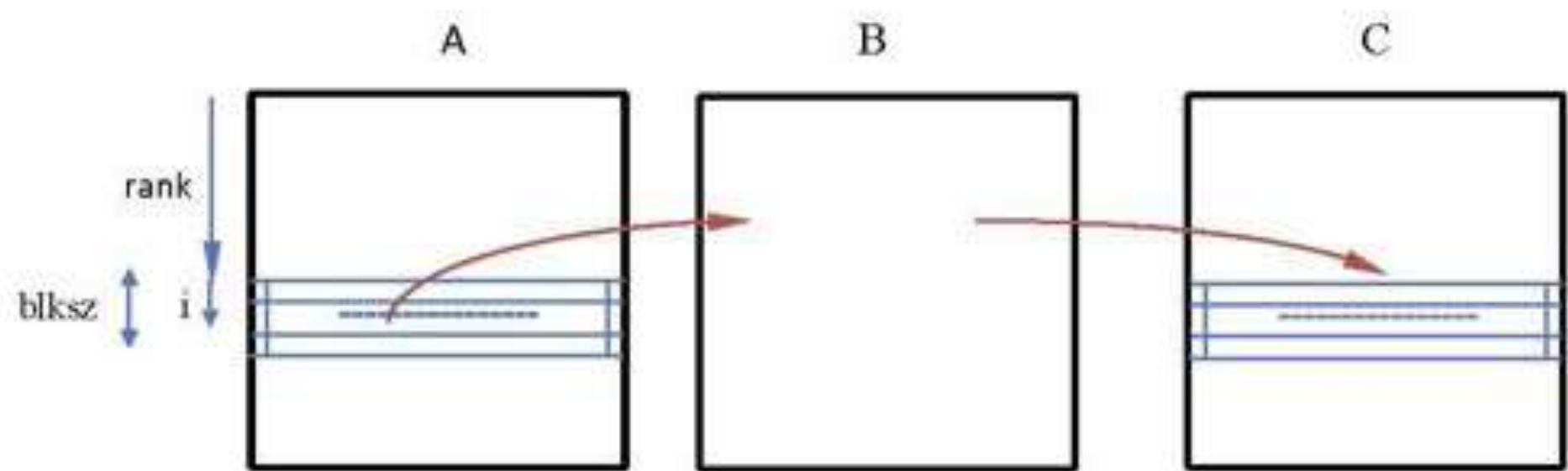
Requires n^3 multiplications and n^3 additions. Sequential time complexity of $O(n^3)$.

Very easy to parallelize as each result independent. Each processor computes one C element or group of C elements

Matrix multiplication

Often matrix size (N) much larger than number of processes (P). Rather than having one process for each result, have each process compute a group of result elements.

In MPI, convenient arrangement is to take a group of rows of **A** and multiply that with whole of **B** to create a groups of rows of **C**:



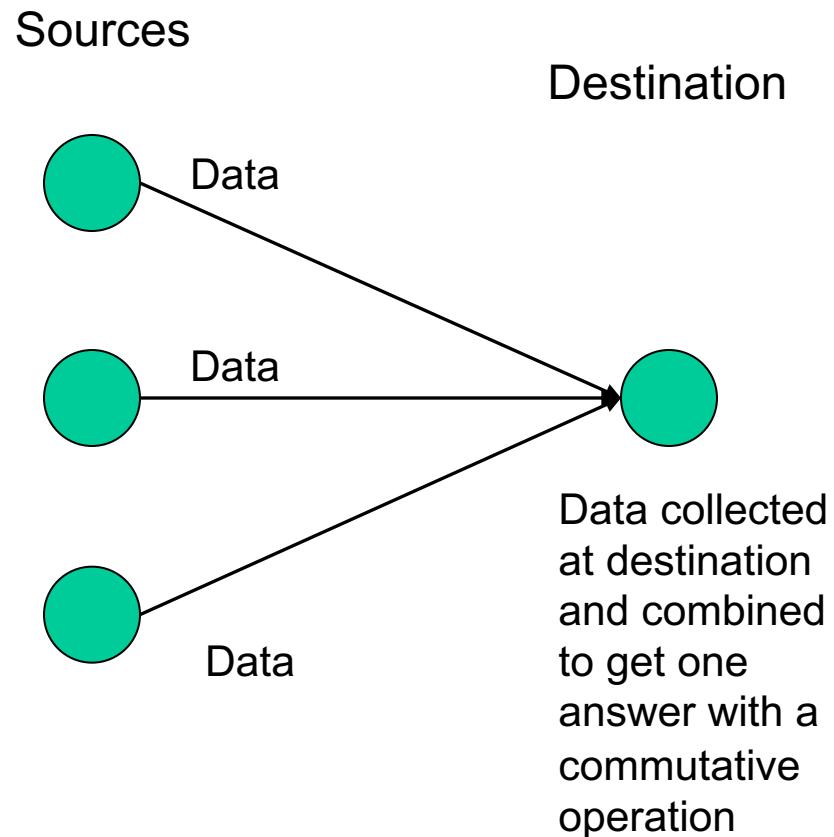
Matrix multiplication

```
MPI_Scatter(A, blksz*N,MPI_DOUBLE,A,blksz*N,  
            MPI_DOUBLE,0, MPI_COMM_WORLD); // Scatter A  
  
MPI_Bcast(B, N*N, MPI_DOUBLE, 0, MPI_COMM_WORLD); // Broadcast B  
  
for(i = 0 ; i < blksz; i++) {  
    for(j = 0 ; j < N ; j++) {  
        C[i][j] = 0;  
        for(k = 0 ; k < N ; k++) {  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}  
  
MPI_Gather(C, blksz*N,MPI_DOUBLE,C,blksz*N,MPI_DOUBLE,0,  
            MPI_COMM_WORLD);
```

Reduce Pattern

Common pattern to get data back to master from all processes and then aggregate it by combining collected data into one answer.

Reduction operation must be a binary operation that is commutative (changing the order of the operands does not change the result)

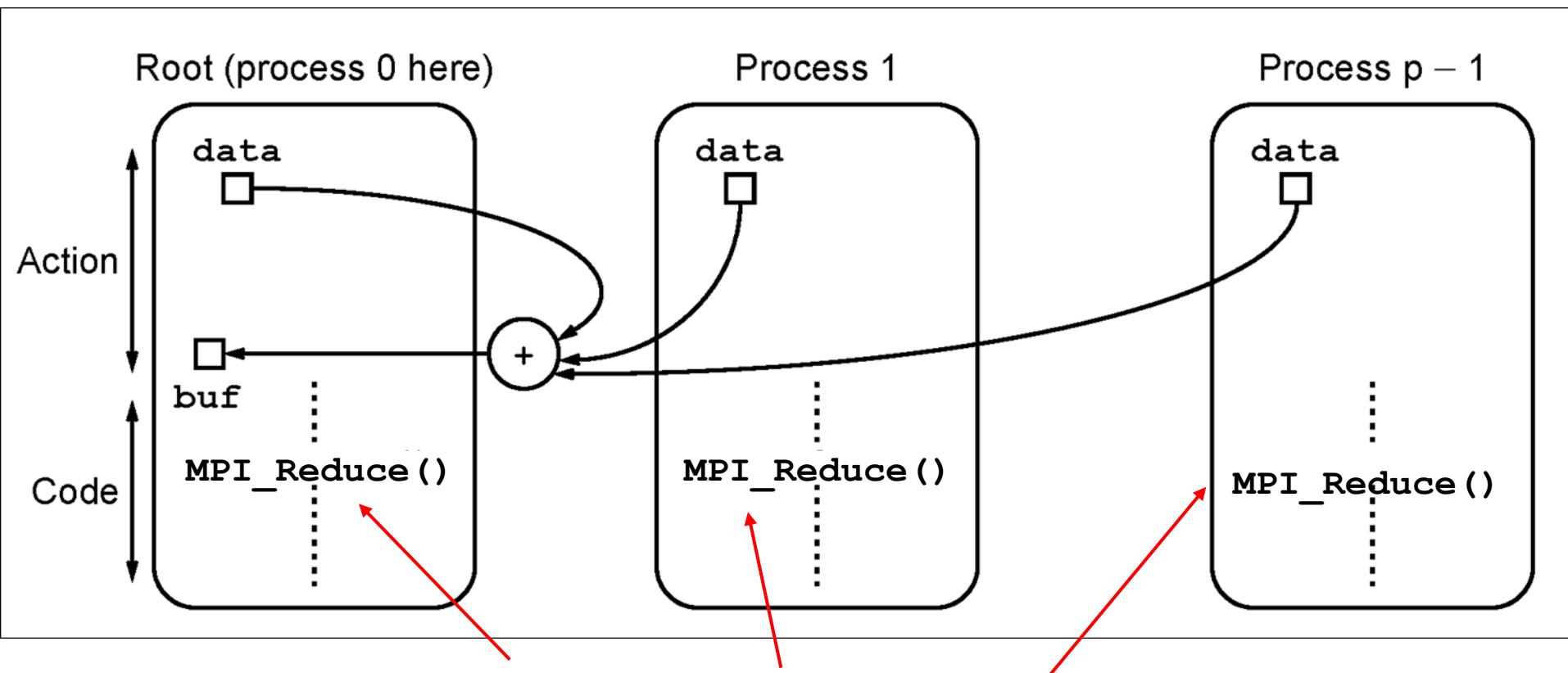


Needs to be commutative operation to allow the implementation to do the operations in any order, see later.

MPI Reduce

Gather operation combined with specified arithmetic/logical operation.

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



As usual same routine called by each process, with same parameters.

Reduce parameters

```
MPI_Reduce(*sendbuf, *recvbuf, count, datatype, op root, comm)
           /         /           |           |           |           |
           Address of   Address of   Datatype of   Operation   Communicator
           send buffer  receive buffer  each item    |           |
                                         |           |           |           |
                                         Number of items
                                         to send      |           |
                                         |           |           |           |
                                         Rank of root
                                         process (destination)
```

All processes in the Communicator must call the MPI_Reduce with the same parameters

Reduce - operations

MPI_Reduce(*sendbuf,*recvbuf,count,datatype,op,root,comm)

Parameters:

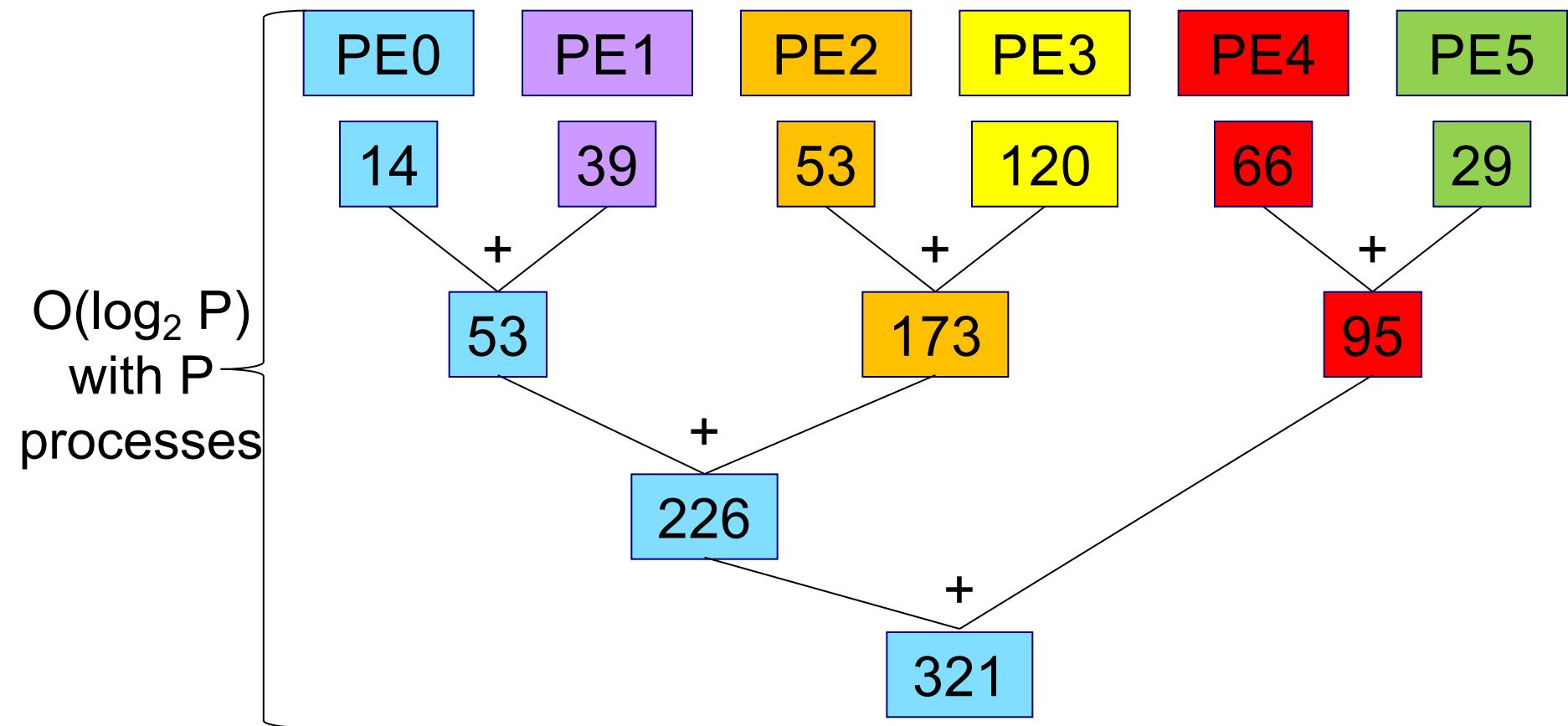
*sendbuf	send buffer address
*recvbuf	receive buffer address
count	number of send buffer elements
datatype	data type of send elements
op	reduce operation.

Several operations, including

MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product

root	root process rank for result
comm	communicator

Implementation of reduction using a tree construction



MPI_Reduce Example

```
#define N 8
int main( int argc, char **argv ) {
    int rank, P, i, j;
    int table[N], result[N];
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &P );
    srand((unsigned int) rank+1);
    for (i = 0; i < N; i++) {
        table[i] = ((float) random()) / RAND_MAX * 100;
    }
    for (j = 0; j < N; j++) {
        if (rank == j) {
            printf ("<rank %d>:", rank);
            for (i = 0; i < N; i++) {
                printf ("%4d", table[i]);
            }
            printf ("\n");
        }
        sleep(1);
    }
    MPI_Reduce (table, result, N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf ("\nAnswer:\n");
        printf ("<rank %d>:", rank);
        for (i = 0; i < N; i++)
            printf ("%4d", result[i]);
        printf ("\n");
    }
    MPI_Finalize();
    return 0;
}
```

Diagram annotations:

- An arrow points from the line `MPI_Reduce (table, result, N, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);` to the text "Array from each source".
- An arrow points from the line `printf ("%4d", result[i]);` to the text "Final result in destination".

Output

mpicc test3.c –o test3

mpirun –n 7 test3

```
<rank 0>: 84 39 78 79 91 19 33 76
<rank 1>: 76 80 8 12 34 42 69 6
<rank 2>: 56 22 39 44 28 14 56 86
<rank 3>: 91 13 19 26 21 36 41 65
<rank 4>: 27 4 99 8 14 7 28 44
<rank 5>: 13 96 79 40 8 80 64 24
<rank 6>: 48 86 59 21 1 51 99 3
```

Answer:

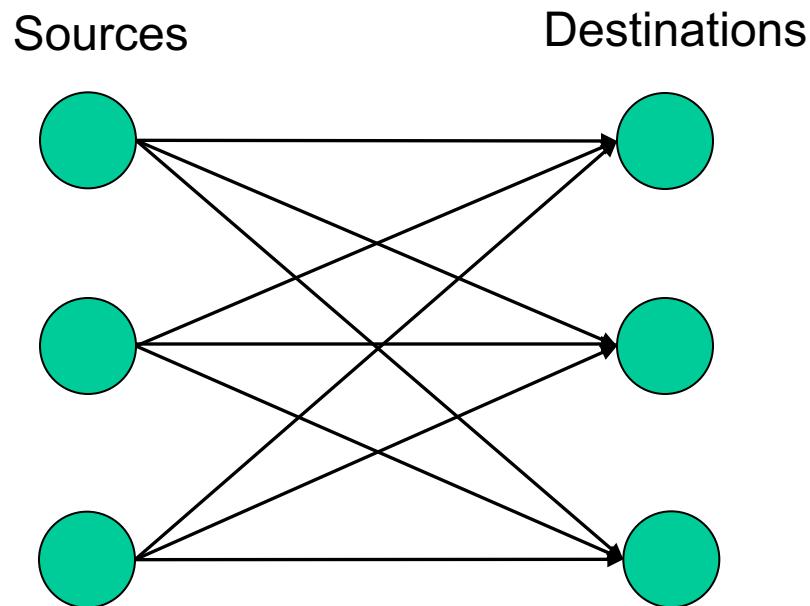


```
<rank 0>: 389 340 381 230 197 249 350 384
```

Combined Patterns

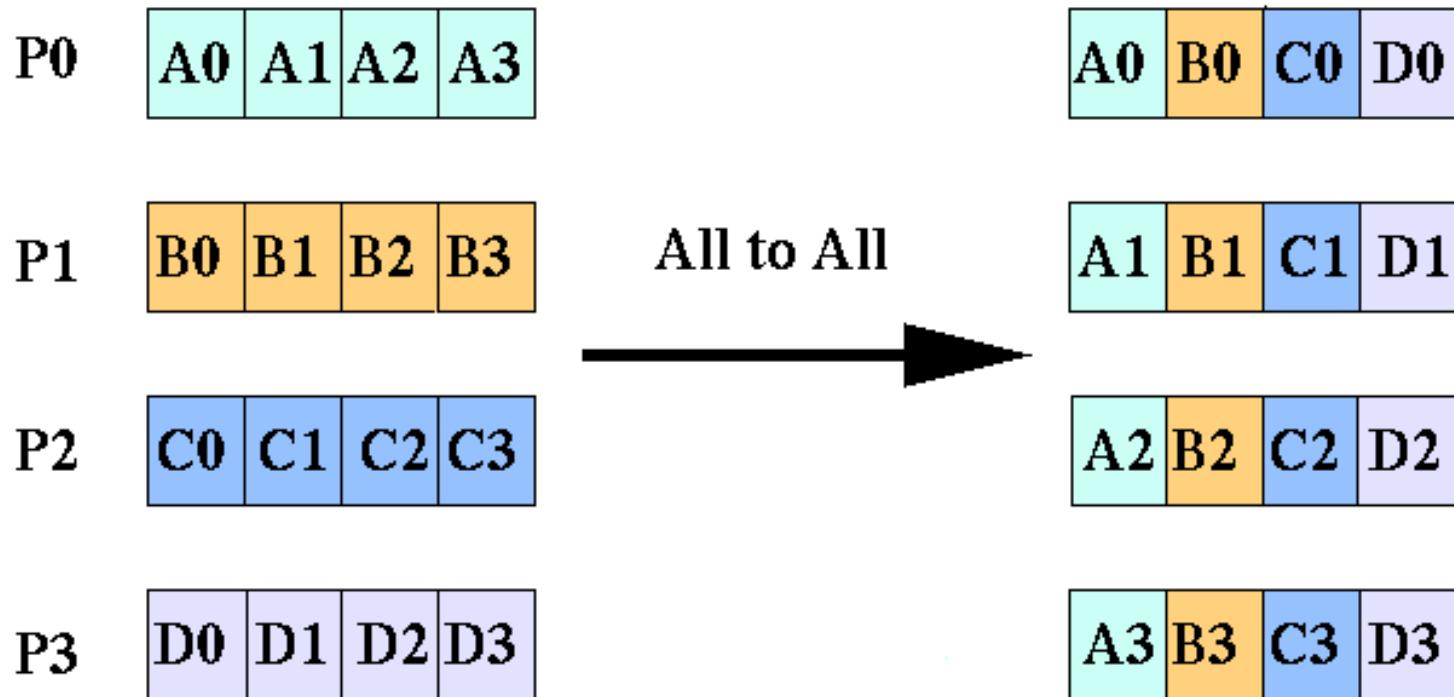
Collective all-to-all broadcast

A common pattern to send data from all processes to all processes often within a computation



MPI_AlltoAll

Combines multiple scatters:



This is essentially matrix transposition

MPI_AlltoAll parameters

```
int MPI_Alltoall (  
    void *sendbuf,  
    int sendcount,  
    MPI_Datatype sendtype,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    MPI_Comm comm );
```

Some other combined patterns in MPI

- **`MPI_Reduce_scatter()`** Combines values and scatters the results
- **`MPI_Allreduce()`** Combines values from all processes and distributes the results back to all processes
- **`MPI_Sendrecv()`** Sends and receives a message (without deadlock, see later)

MPI Collective data transfer routines

General features

- Performed on a group of processes, identified by a communicator
- Substitutes for a sequence of point-to-point calls
- Communications are **locally blocking**
- Synchronization is *not guaranteed* (implementation dependent)
- Most routines use a **root** process to originate or receive all data (broadcast, scatter, gather, reduce, ...)
- Data amounts must exactly match
- Many variations to basic categories
- No message tags needed

Data transfer collective routines

Synchronization

- Generally the collective operations have the same semantics as if individual MPI_send()’s and MPI_recv()’s were used according to the MPI standard.
- i.e. both sends and recv’s are locally blocking, sends will return after sending message, recv’s will wait for messages.
- However we need to know the exact implementation to really figure out when each process will return.



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Collective patterns
and MPI routines

Synchronizing Computations

Recap on synchronization

- Our *collective data transfer patterns* *do not specify whether synchronized* (an implementation detail)
- MPI routines that implement these patterns *do not generally synchronize all the processes.*
- Collective data transfer MPI routines have same semantics as if individual MPI_send()s and MPI_recv()'s were used (according to the MPI standard).
- Unfortunate different implementations may do things differently and with different network configurations.
- All we can really say is the destination returns when the message is received.

Synchronizing processes

Needed in many applications to ensure parallel processes start at same point and data is available to work on by processes.

Synchronization should be avoided where possible as it delays processes but sometimes it is unavoidable.

Some algorithms such as iterative computations require previous iteration values to compute next iteration values. So previous values need to be computed first.

(This constraint can be relaxed for increased performance in some cases, see later in course)

Synchronous Message Passing

Routines that return when message transfer completed.

Synchronous send routine

- Returns only after message received (matching receive posted) -- in MPI, `MPI_SSend()` routine.

Synchronous receive routine

- Waits until the message it is expecting arrives -- in MPI, actually the regular `MPI_recv()` routine.

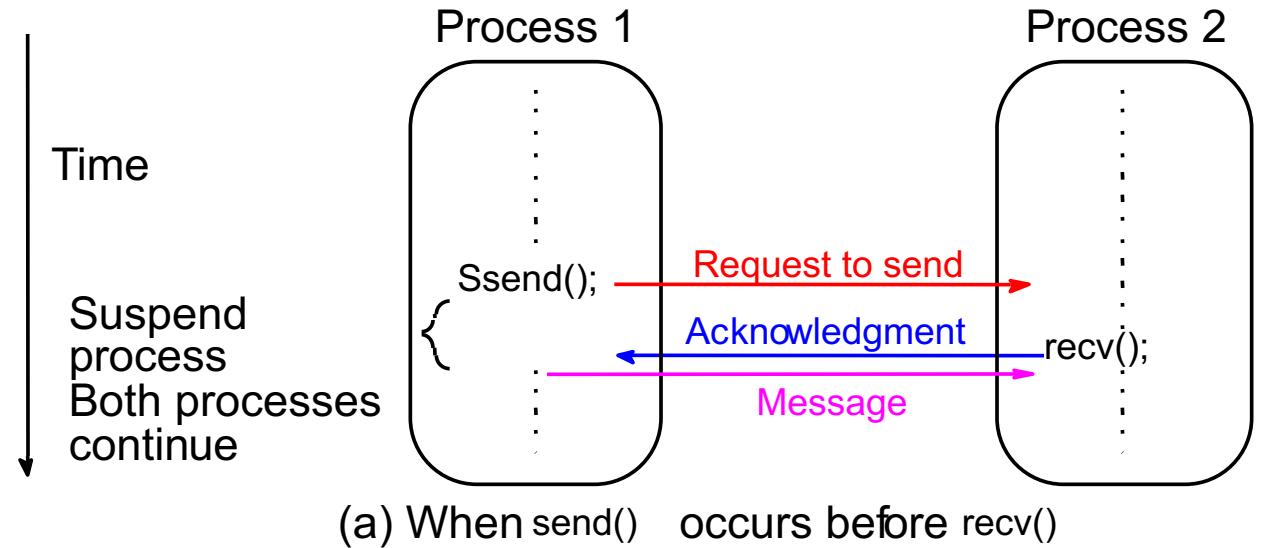
Synchronous Message Passing

Synchronous message-passing routines intrinsically perform two actions:

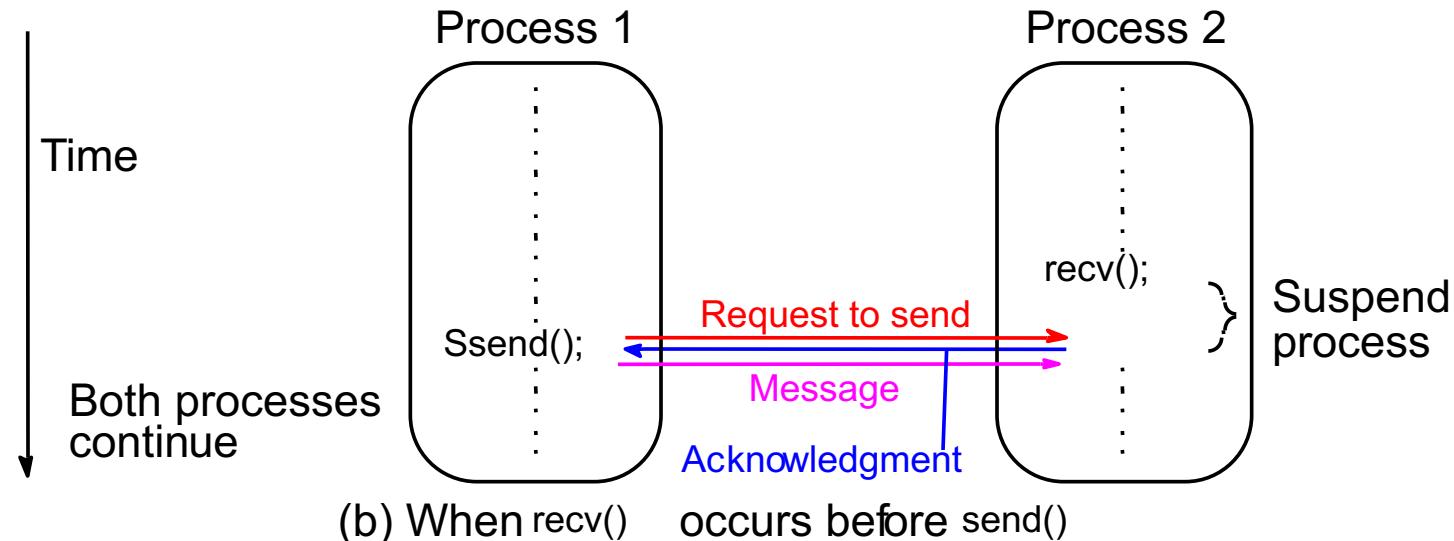
- They transfer data and
- They synchronize processes.

Possible implementation of synchronous Ssend() and recv() using 3-way protocol

In this case, send waits until complete message can be accepted by receiving process before sending message.



Then does not need a external message buffer.

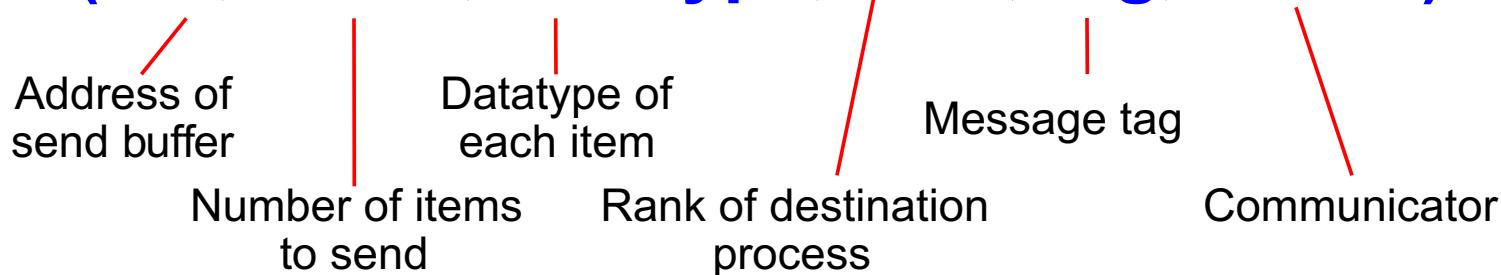


Again in MPI actual implementation not specified in standard

Parameters of synchronous send, **MPI_Ssend()**

(actually the same as blocking send)

MPI_Ssend(buf, count, datatype, dest, tag, comm)



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.
Sometimes called **locally blocking**.
- **Non-blocking** - return immediately (*asynchronous*)

Non-blocking 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.**

Blocking/non-blocking terms may have different interpretations in other systems.

MPI blocking routines

Block until local actions complete

- **Blocking send** - `MPI_send()` - blocks only until message is on its way. User can modify buffer after it returns.
- **Blocking receive** - `MPI_recv()` - blocks until message arrives

MPI Nonblocking Routines

- **Non-blocking send** - `MPI_Isend()` - will return “immediately” even before source location is safe to be altered.

User ***should not*** modify the send buffer until the communication completes.

- **Non-blocking receive** - `MPI_Irecv()` - will return even if no message to accept.

User ***should not*** modify the recv buffer until the communication completes.

Nonblocking Routine Formats

MPI_Isend(buf, count, datatype, dest, tag, comm, req)

MPI_Irecv(buf, count, datatype, source, tag, comm, req)

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

MPI_Wait(req, status) waits until operation completed and returns then.

MPI_Test(req, flag, status) returns with flag set indicating whether operation completed at that time.

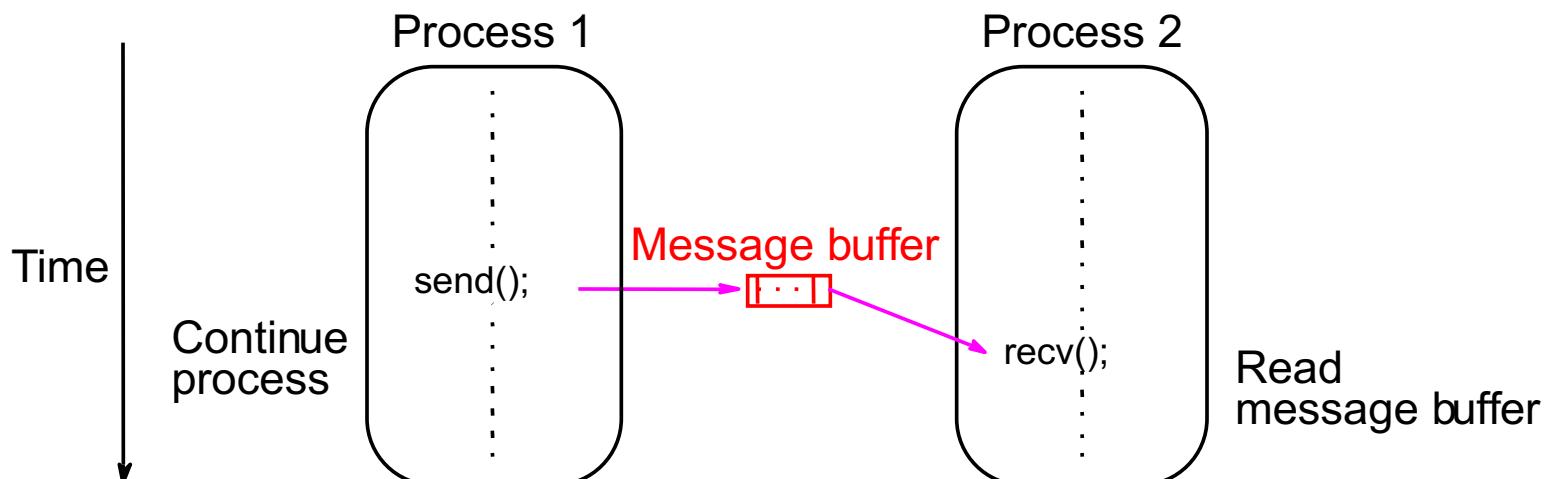
Example

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);
}
```

How message-passing routines return before message transfer completed

Message buffer needed between source and destination to hold message:



Asynchronous (blocking) routines

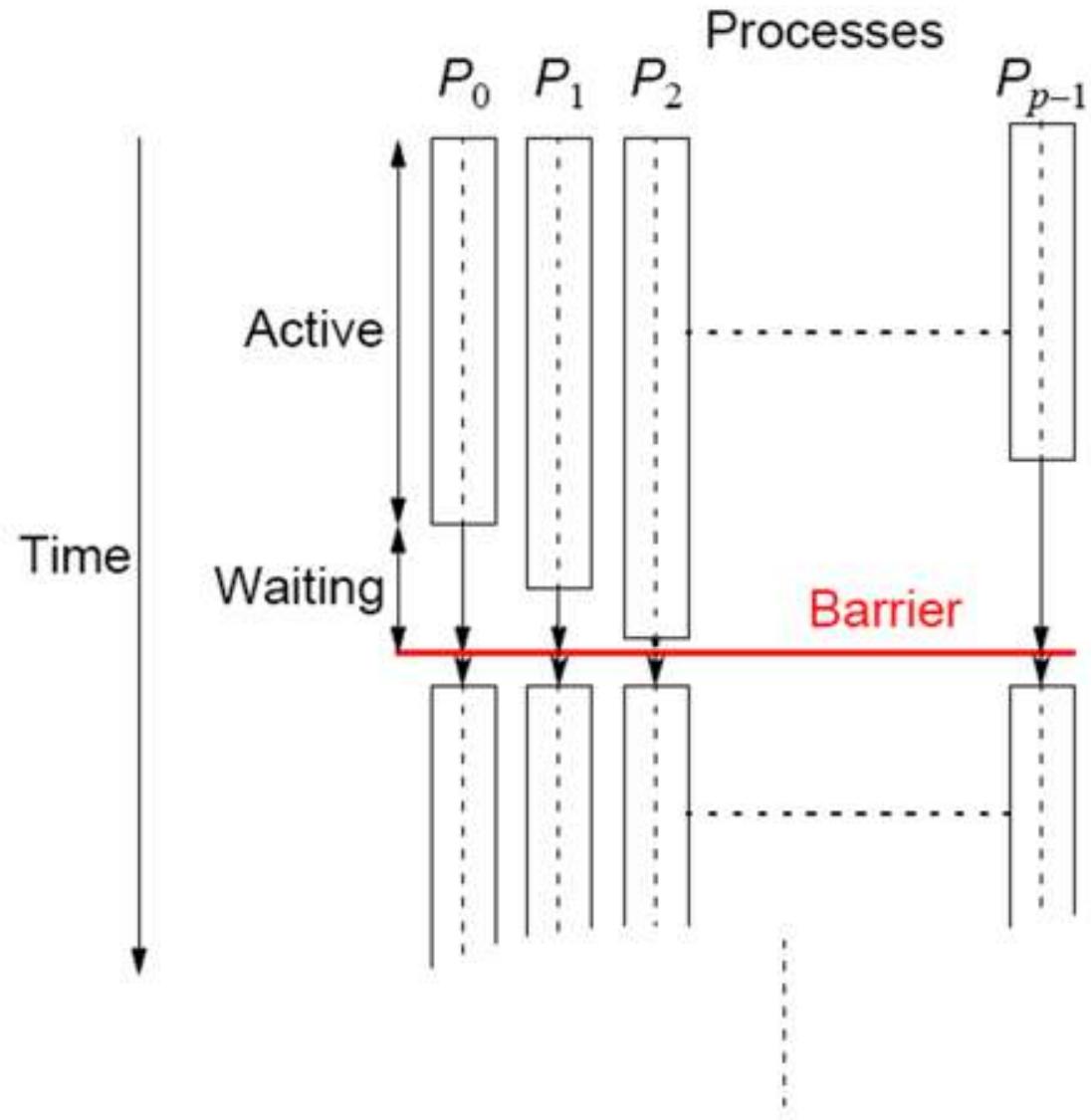
changing to synchronous routines

- Message buffers only of finite length
- 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. routine will behave as a synchronous routine.

Barrier Synchronization

Basic mechanism for synchronizing processes - inserted at point in each process where it must wait.

All processes can only continue from this point when all processes have reached it (or, in some systems, when a stated number of processes have reached this point).

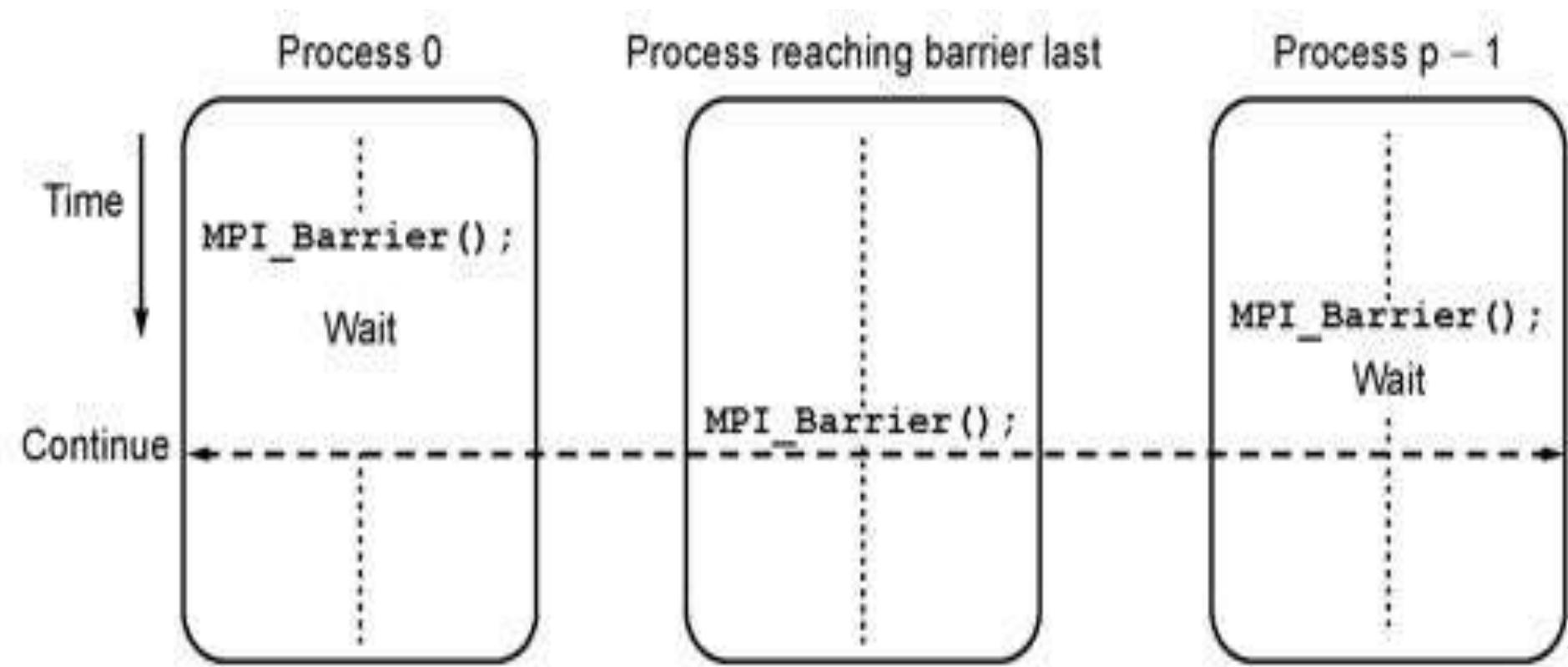


MPI Barrier

MPI_Barrier(comm)

Communicator

Barrier with a named communicator being only parameter.
Called by each process in the group, blocking until all members
of group have reached barrier call and only returning then.



MPI_Barrier use with time stamps

A common example of using a barrier is to synchronize processes before taking a time stamp.

```
MPI_Barrier(MPI_COMM_WORLD);
start_time = MPI_Wtime();
... \\\ Do work
MPI_Barrier(MPI_COMM_WORLD);
end_time = MPI_Wtime();
```

2nd barrier not always needed if synchronization already present, for example a gather in the root. Once root has correct data, who cares what the other processes are doing. We have the answer.

Safety and Deadlock

When all processes send their messages **first** and then receive all of their messages is “**unsafe**” because it relies upon buffering in the **send()**s. The amount of buffering is not specified in MPI.

If insufficient storage available, send routine may be delayed from returning until storage becomes available or until the message can be sent without buffering.

Then, a locally blocking **send()** could behave as a synchronous **send()**, only returning when the matching **recv()** is executed. Since a matching **recv()** would never be executed if all the **send()**s are synchronous, **deadlock would occur**.

Making the code safe

Alternate the order of the **send()**s and **recv()**s in adjacent processes so that only one process performs the **send()**s first.

Then even synchronous **send()**s would not cause deadlock.

Example

Linear pipeline, deadlock can be avoided by arranging so the even-numbered processes perform their sends first and the odd-numbered processes perform their receives first.

Good way you can test for safety is to replace message-passing routines in a program with synchronous versions.

MPI Safe Message Passing Routines

MPI offers several methods for safe communication:

- Combined send and receive routines:

`MPI_Sendrecv()`

which is guaranteed not to deadlock

- Buffered send(s):

`MPI_Bsend()`

here the user provides explicit storage space

- Nonblocking routines:

`MPI_Isend()` and `MPI_Irecv()`

which return immediately.

Separate routine used to establish whether message has been received:

`MPI_Wait()`, `MPI_Waitall()`, `MPI_Waitany()`, `MPI_Test()`,
`MPI_Testall()`, or `MPI_Testany()`.

Combined deadlock-free blocking sendrecv() routines

Example

Process P_{i-1}

Process P_i

Process P_{i+1}

```
sendrecv(Ri) ; ←→ sendrecv(Ri-1) ;
                           sendrecv(Ri+1) ; ←→ sendrecv(Ri) ;
```

MPI provides **`MPI_Sendrecv()`** and **`MPI_Sendrecv_replace()`**.
(with 12 parameters!)

MPI_Sendrecv()

Combines blocking send with blocking receive operation without deadlock. Source and destination can be the same or *different*.

```
int MPI_Sendrecv(  
    void *sendbuf,  
    int sendcount,  
    MPI_Datatype sendtype,  
    int dest,  
    int sendtag,  
    void *recvbuf,  
    int recvcount,  
    MPI_Datatype recvtype,  
    int source,  
    int recvtag,  
    MPI_Comm comm,  
    MPI_Status *status)
```

Parameters for send

Parameters for receive

Same communicator



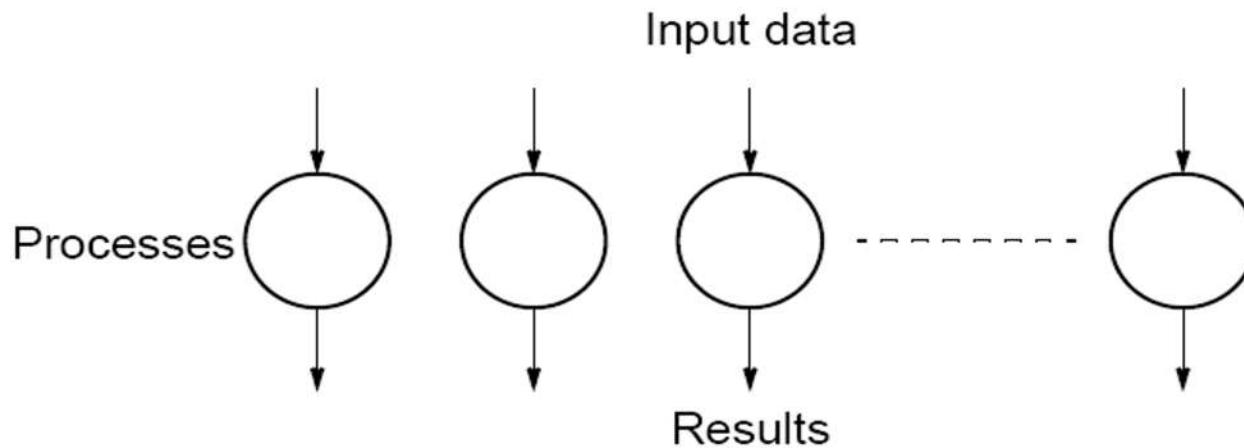
Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Embarrassingly Parallel Computations

Embarrassingly Parallel Computations

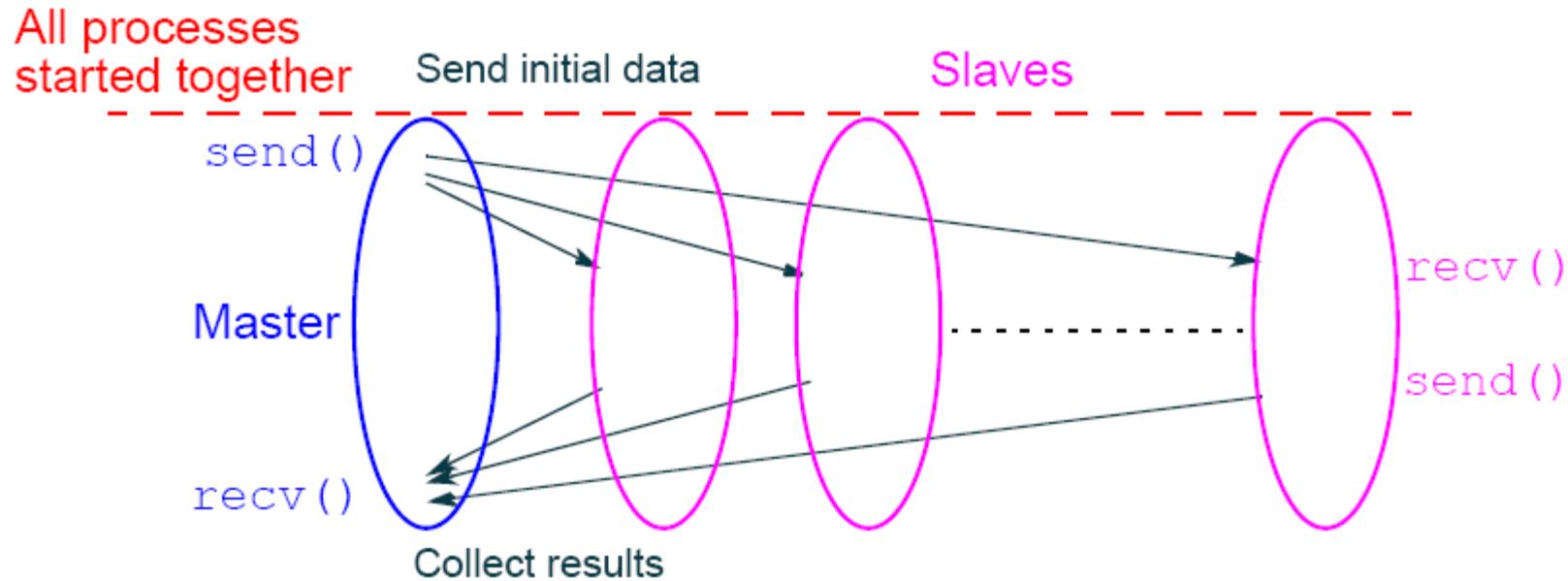
A computation that can **obviously** be divided into a number of completely independent parts, each of which can be executed by a separate process(or).



No communication or very little communication between processes

Each process can do its tasks without any interaction with other processes

Practical embarrassingly parallel computation with static process creation and master-slave approach



Usual MPI approach

Embarrassingly Parallel Computation Examples

- Low level image processing
- Mandelbrot set
- Monte Carlo Calculations

Low level image processing

Many low level image processing operations only involve local data with very limited if any communication between areas of interest.

Some geometrical operations

Shifting

Object shifted by Δx in the x -dimension and Δy in the y -dimension:

$$x' = x + \Delta x$$

$$y' = y + \Delta y$$

where x and y are the original and x' and y' are the new coordinates.

Scaling

Object scaled by a factor S_x in x -direction and S_y in y -direction:

$$x' = xS_x$$

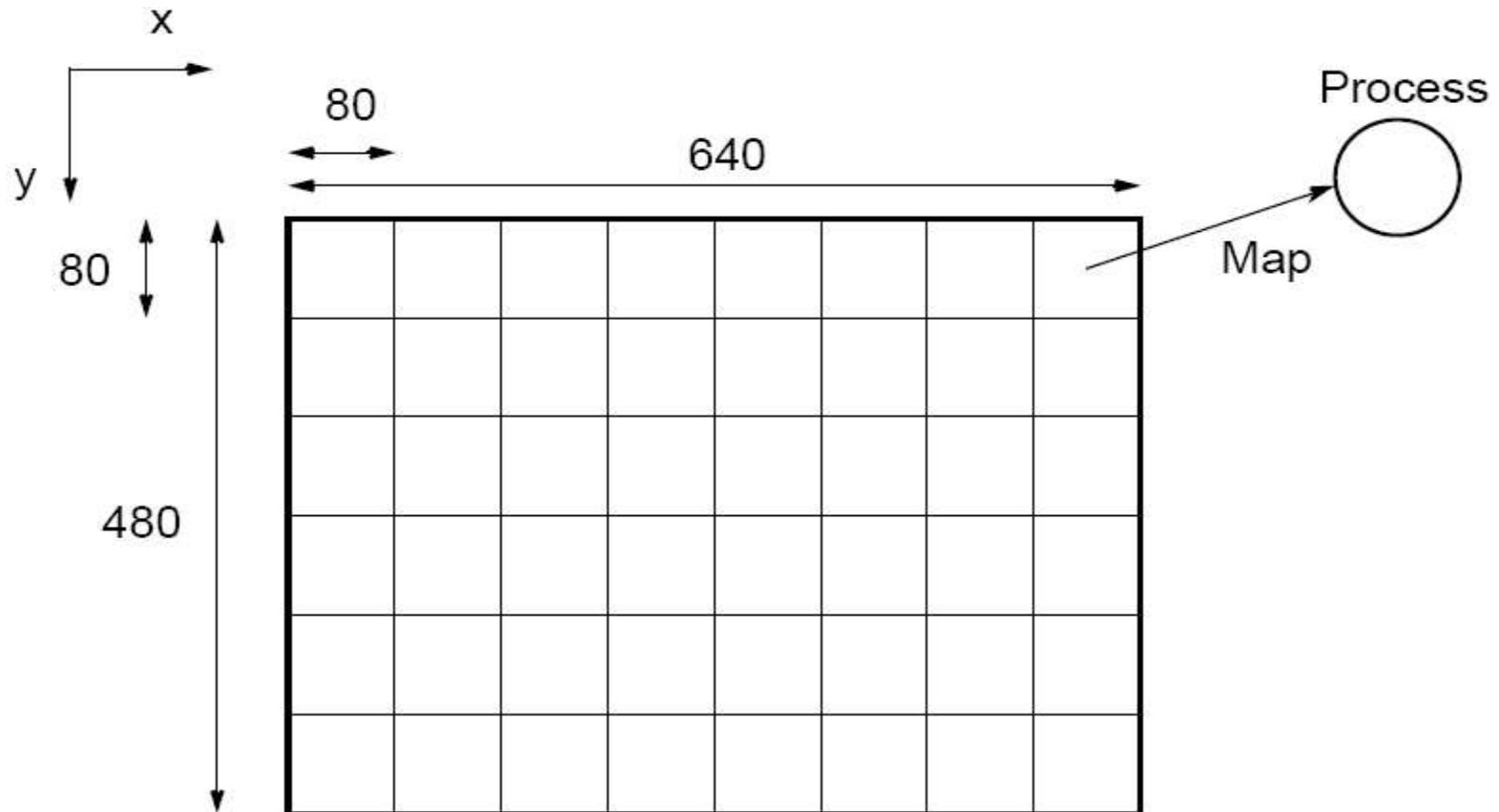
$$y' = yS_y$$

Rotation

Object rotated through an angle θ about the origin of the coordinate system:

$$\begin{aligned}x' &= x \cos\theta + y \sin\theta \\y' &= -x \sin\theta + y \cos\theta\end{aligned}$$

Partitioning into regions for individual processes



Square region for each process (can also use strips)

Monte Carlo Methods

Another embarrassingly parallel computation.

Monte Carlo methods use of random selections.

Example - To calculate π

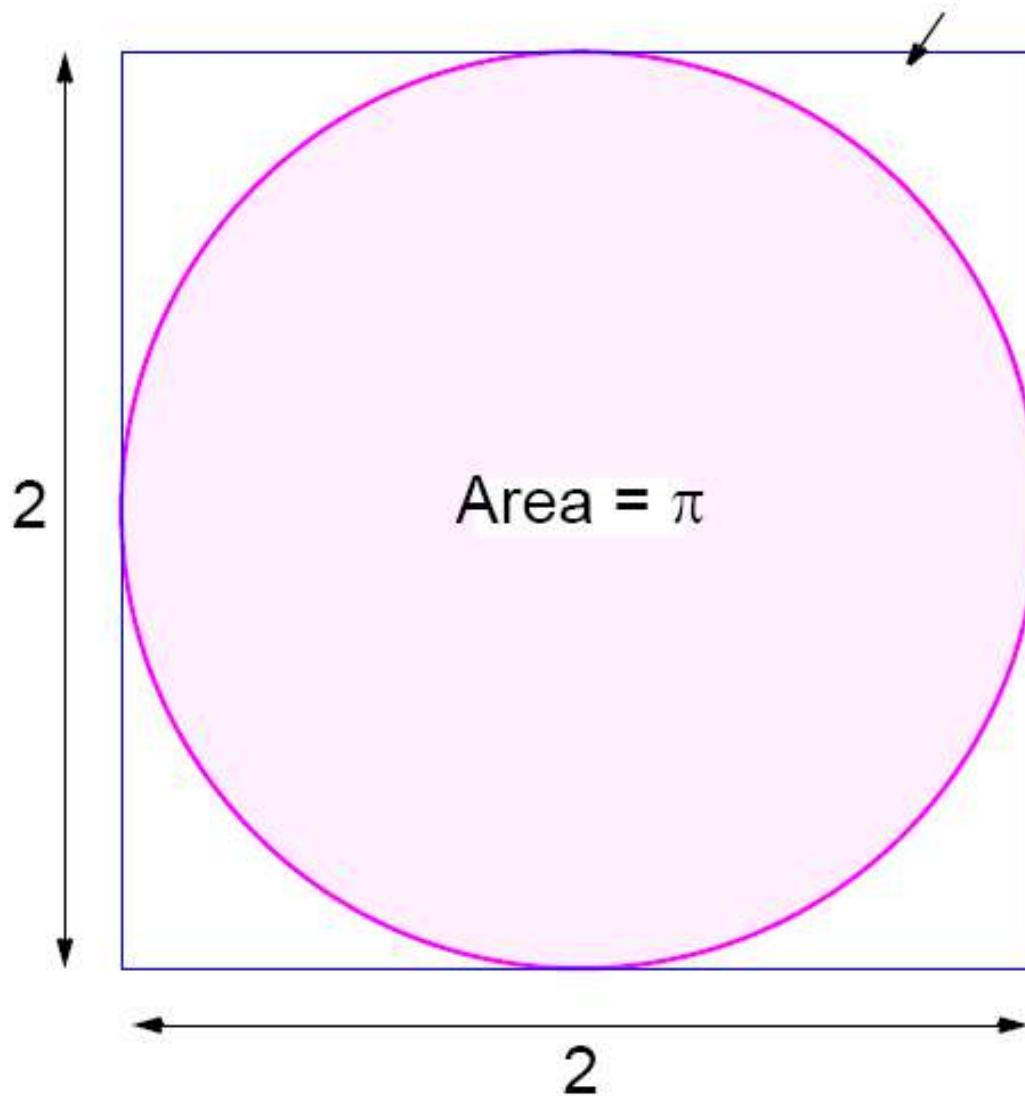
Circle formed within a 2×2 square. Ratio of area of circle to square given by:

$$\frac{\text{Area of circle}}{\text{Area of square}} = \frac{\pi(1)^2}{2 \times 2} = \frac{\pi}{4}$$

Points within square chosen randomly. Score kept of how many points happen to lie within circle.

Fraction of points within the circle will be $\pi/4$, given a sufficient number of randomly selected samples.

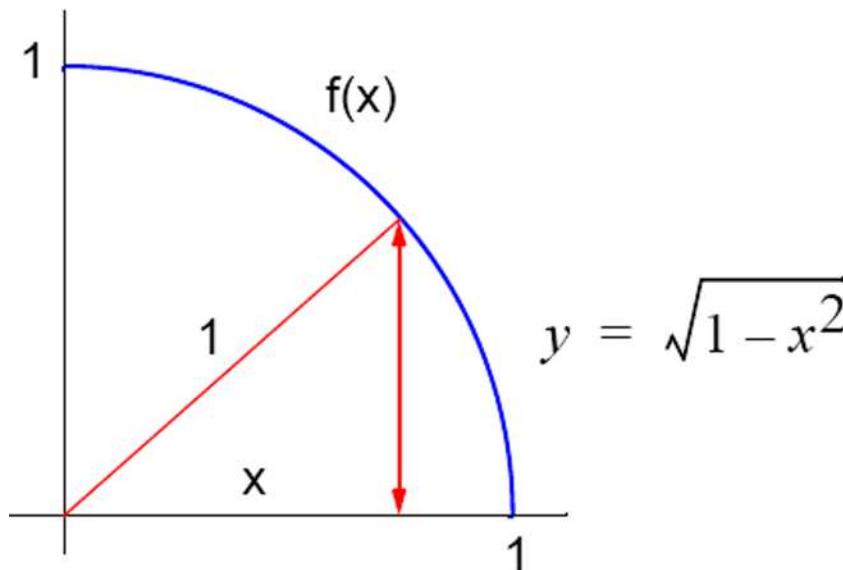
Total area = 4



Computing an Integral

One quadrant can be described by integral

$$\int_0^1 \sqrt{1 - x^2} dx = \frac{\pi}{4}$$



Random pairs of numbers, (x_r, y_r) generated, each between 0 and 1.

Counted as in circle if $y_r \leq \sqrt{1 - x_r^2}$; that is, $y_r^2 + x_r^2 \leq 1$.

Alternative (better) Method

Use random values of x to compute $f(x)$ and sum values of $f(x)$:

$$\text{Area} = \int_{x_1}^{x_2} f(x) dx = \lim_{N \rightarrow \infty} \frac{1}{N} \sum_{i=1}^N f(x_r)(x_2 - x_1)$$

where x_r are randomly generated values of x between x_1 and x_2 .

Monte Carlo method very useful if the function cannot be integrated numerically (maybe having a large number of variables)

Example

Computing the integral

$$I = \int_{x_1}^{x_2} (x^2 - 3x) dx$$

Sequential Code

```
sum = 0;  
for (i = 0; i < N; i++) {          /* N random samples */  
    xr = rand_v(x1, x2);           /* generate next random value */  
    sum = sum + xr * xr - 3 * xr;  /* compute f(xr) */  
}  
area = (sum / N) * (x2 - x1);
```

Routine `randv(x1, x2)` returns a pseudorandom number between `x1` and `x2`.

For parallelizing Monte Carlo code, must address best way to generate random numbers in parallel - see textbook

Mandelbrot Set

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

$$z_{k+1} = z_k^2 + c$$

where z_{k+1} is the $(k + 1)$ th iteration of the complex number $z = a + bi$ and c is a complex number giving position of point in the complex plane. The initial value for z is zero.

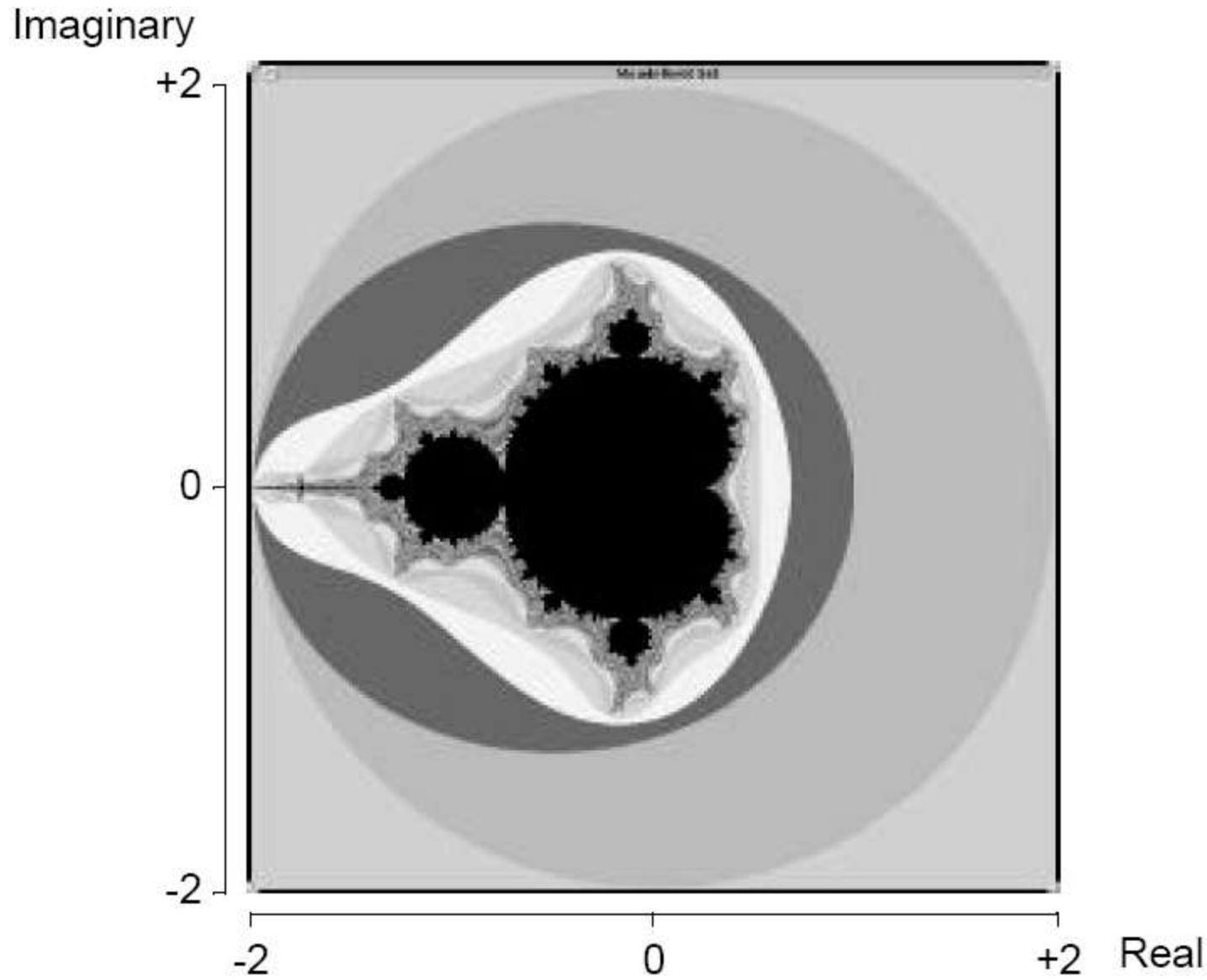
Iterations continued until magnitude of z is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of z is the length of the vector given by

$$z_{\text{length}} = \sqrt{a^2 + b^2}$$

Sequential routine computing value of one point returning number of iterations

```
structure complex {  
    float real;  
    float imag;  
};  
int cal_pixel(complex c)  
{  
    int count, max;  
    complex z;  
    float temp, lengthsq;  
    max = 256;  
    z.real = 0; z.imag = 0;  
    count = 0; /* number of iterations */  
    do {  
        temp = z.real * z.real - z.imag * z.imag + c.real;  
        z.imag = 2 * z.real * z.imag + c.imag;  
        z.real = temp;  
        lengthsq = z.real * z.real + z.imag * z.imag;  
        count++;  
    } while ((lengthsq < 4.0) && (count < max));  
    return count;  
}
```

Mandelbrot set



Parallelizing Mandelbrot Set Computation

Static Task Assignment

Simply divide the region in to fixed number of parts, each computed by a separate processor.

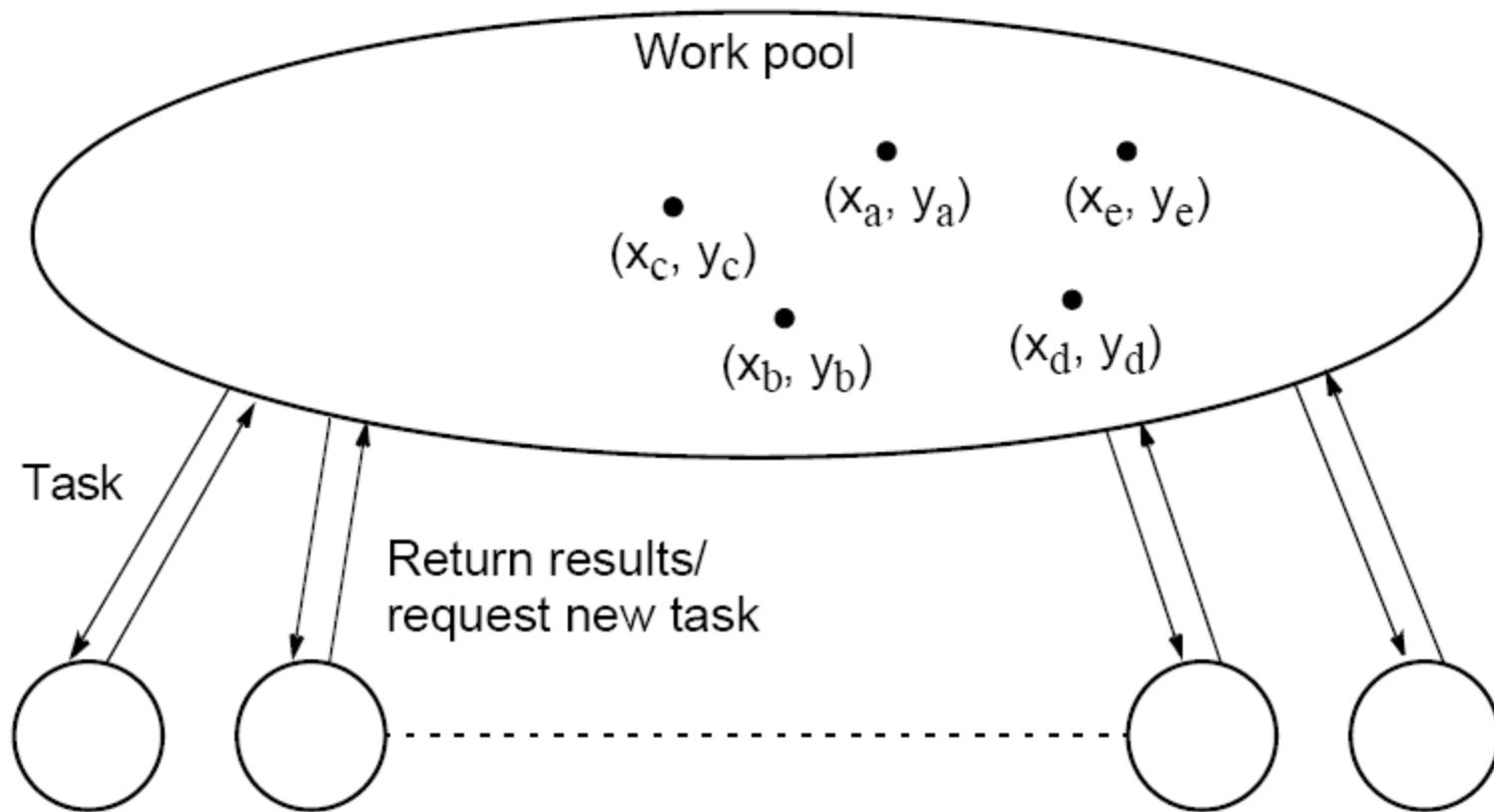
Not very successful because different regions require different numbers of iterations and time.

Dynamic Task Assignment

Have processor request regions after computing previous regions

Dynamic Task Assignment

Work Pool/Processor Farms





Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Partitioning and Divide-and-Conquer Strategies

Partitioning

Partitioning simply divides the problem into parts.

Divide and Conquer

Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.

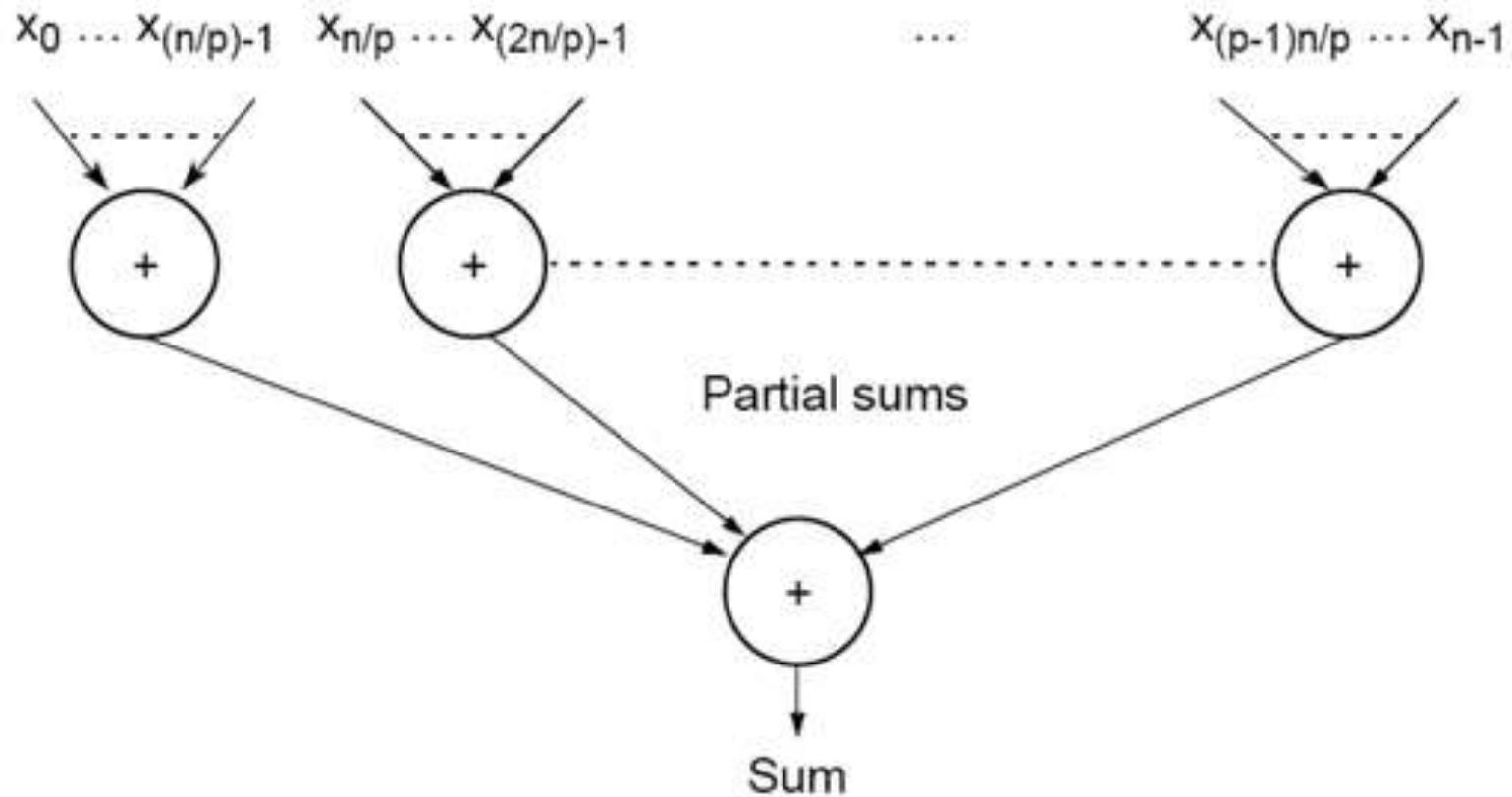
Recursive divide and conquer amenable to parallelization because separate processes can be used for divided parts. Also usually data is naturally localized.

Partitioning/Divide and Conquer Examples

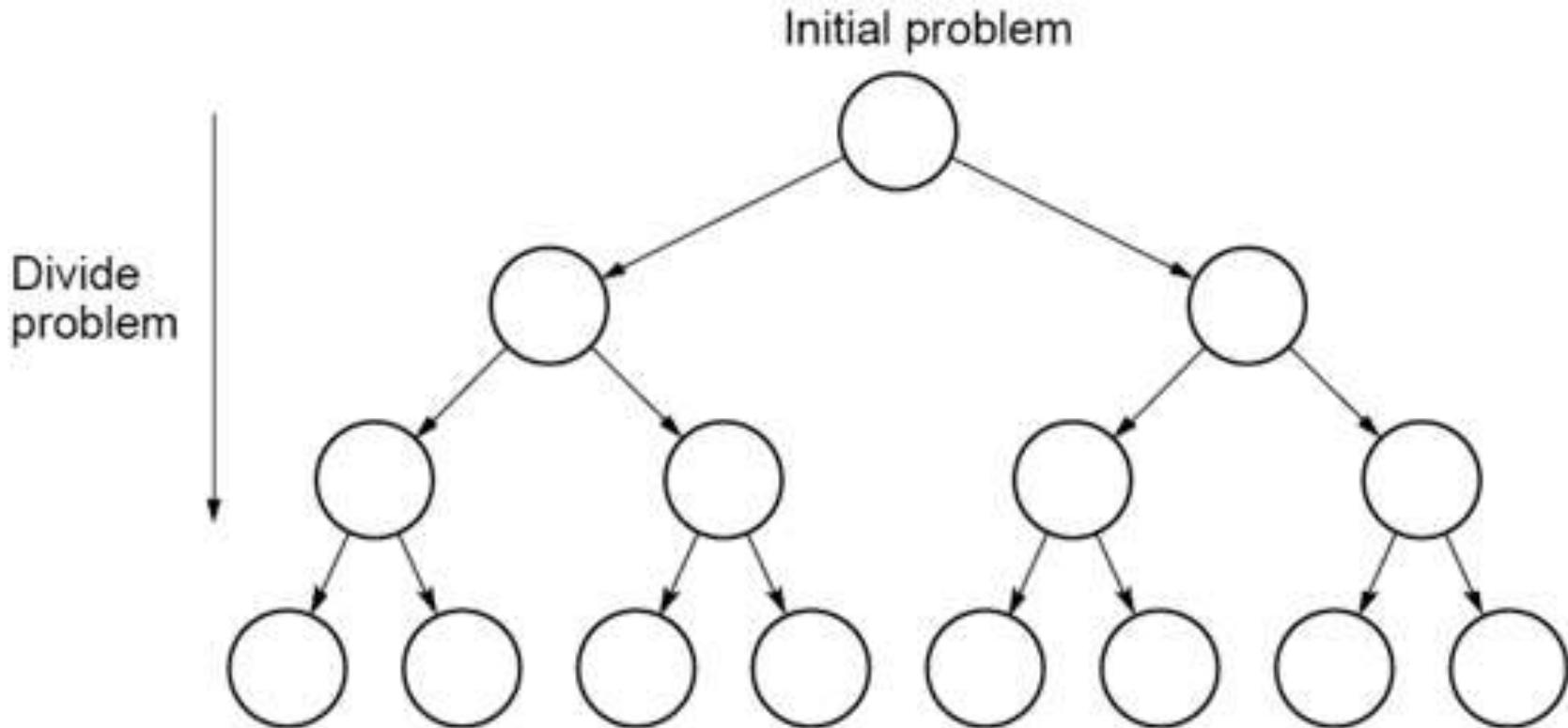
Many possibilities.

- Operations on sequences of numbers such as simply adding them together
- Several sorting algorithms can often be partitioned or constructed in a recursive fashion
- Numerical integration
- N -body problem

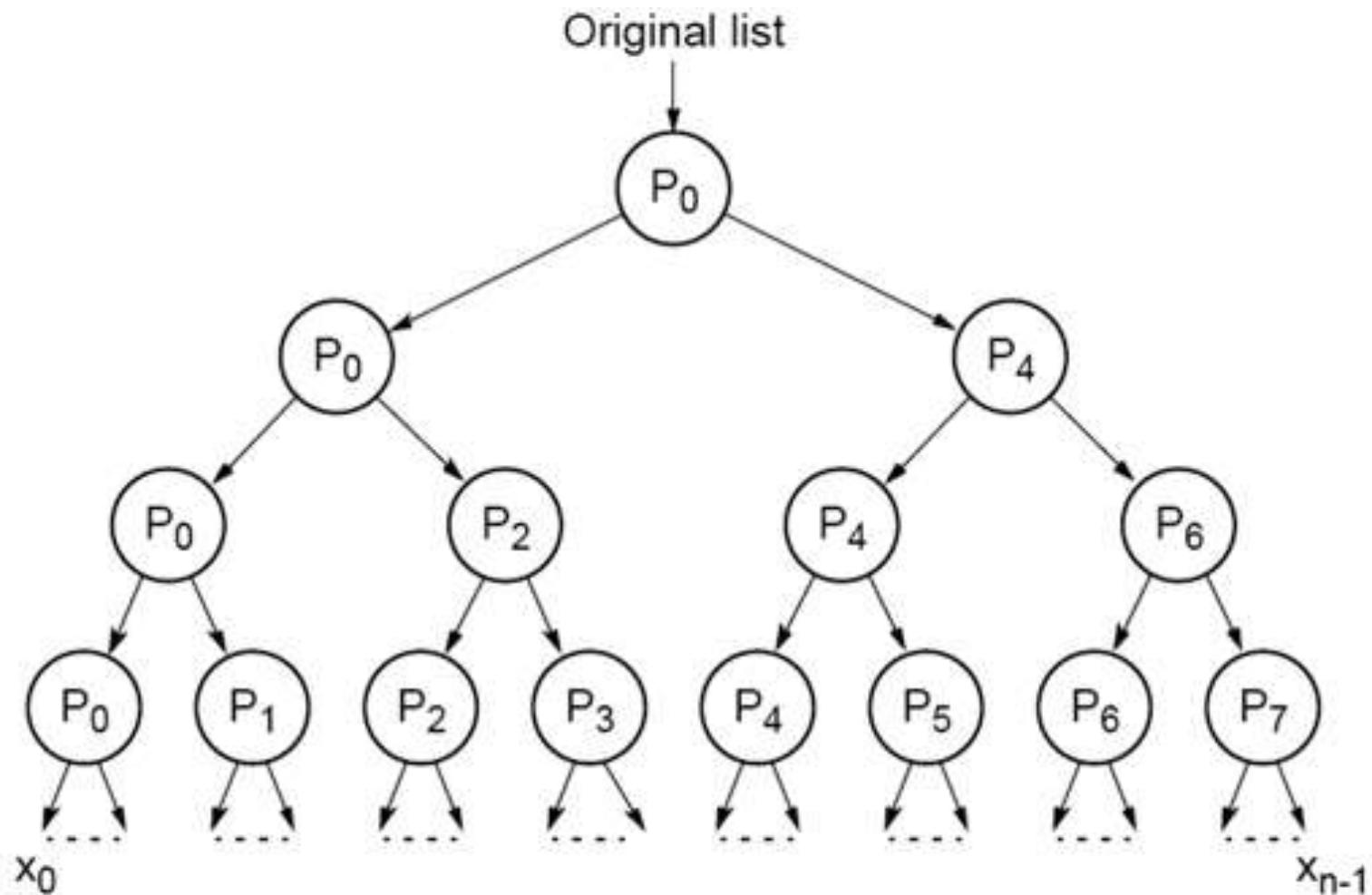
Partitioning a sequence of numbers into parts and adding the parts



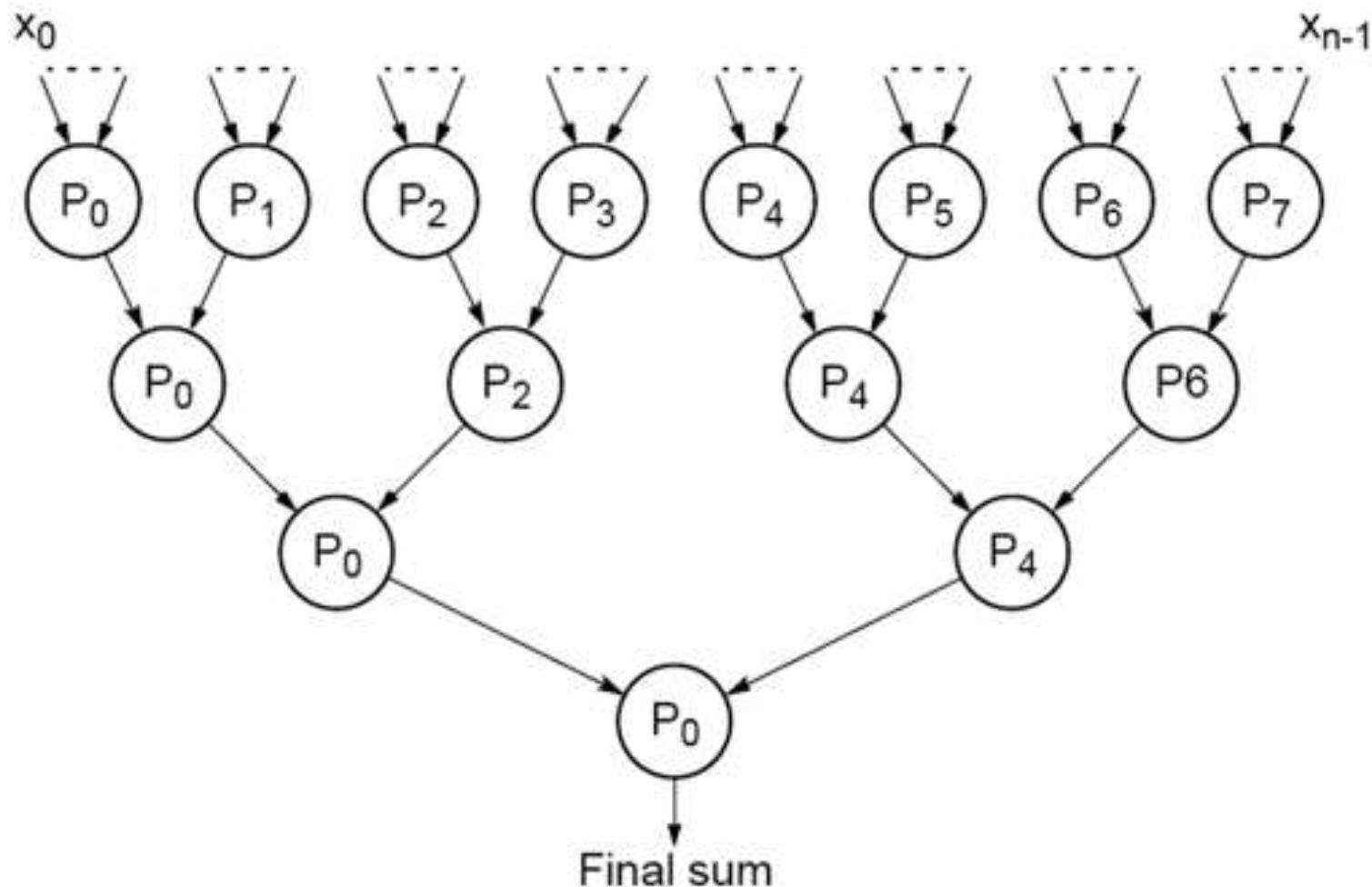
Tree construction



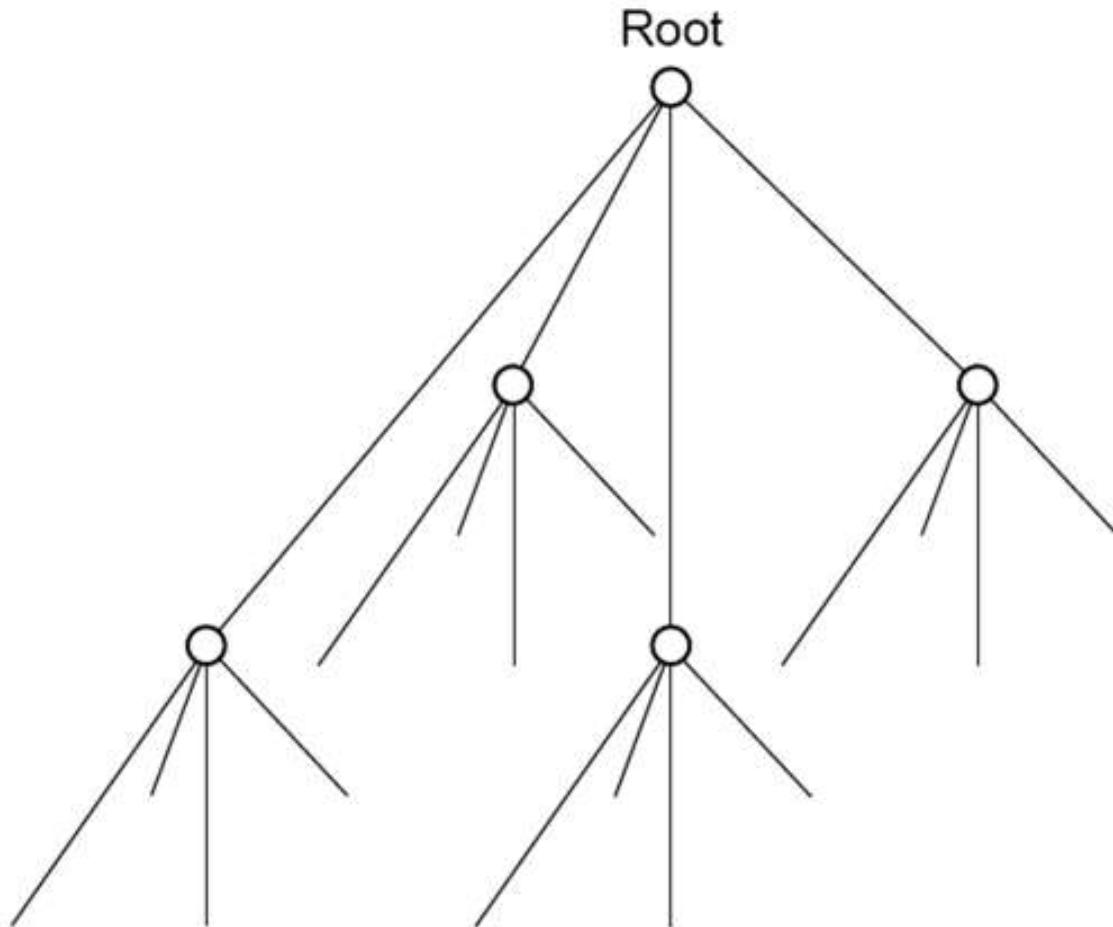
Dividing a list into parts



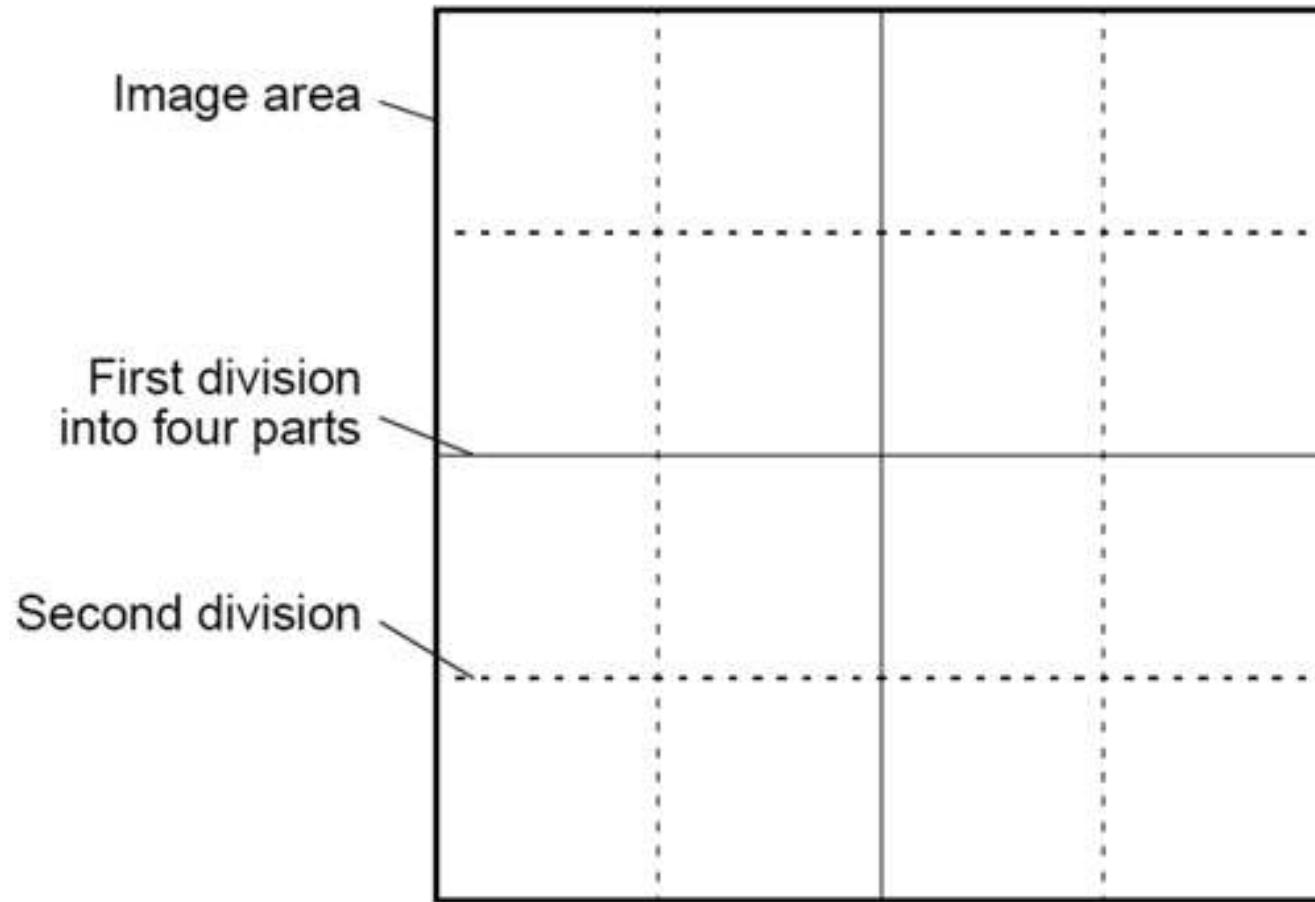
Partial summation



Quadtree

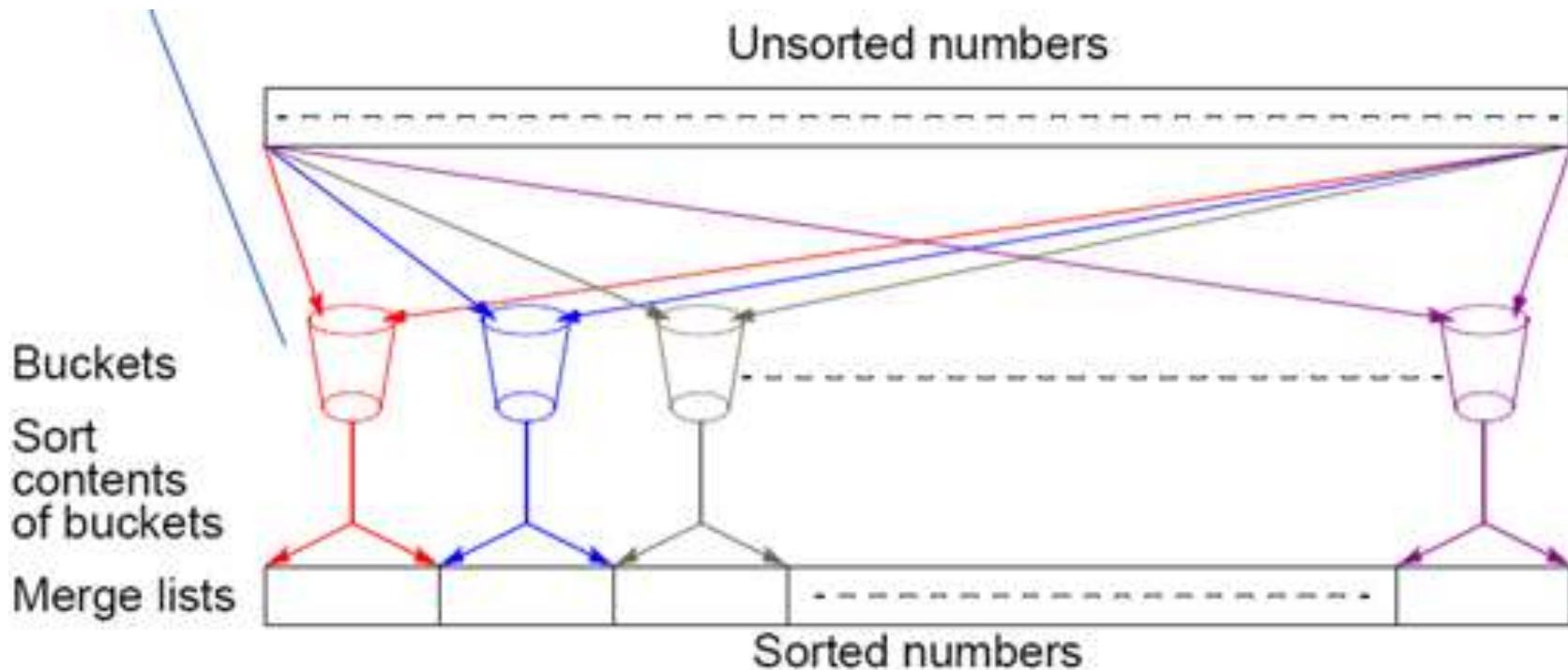


Dividing an image



Bucket sort

One “bucket” assigned to hold numbers that fall within each region.
Numbers in each bucket sorted using a sequential sorting algorithm.



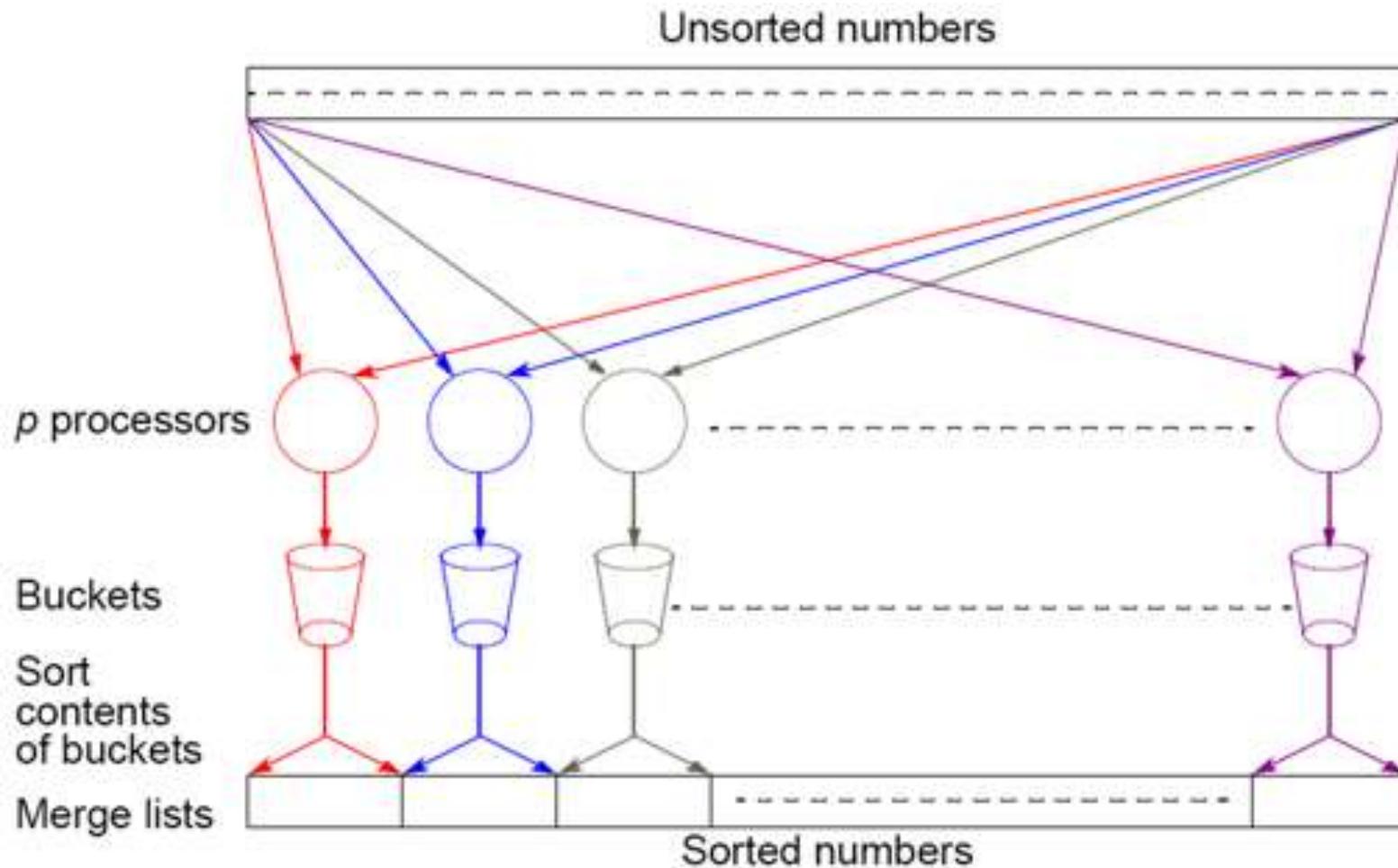
Sequential sorting time complexity: $O(n \log(n/m))$.

Works well if the original numbers uniformly distributed across a known interval, say 0 to $a - 1$.

Parallel version of bucket sort

Simple approach

Assign one processor for each bucket.



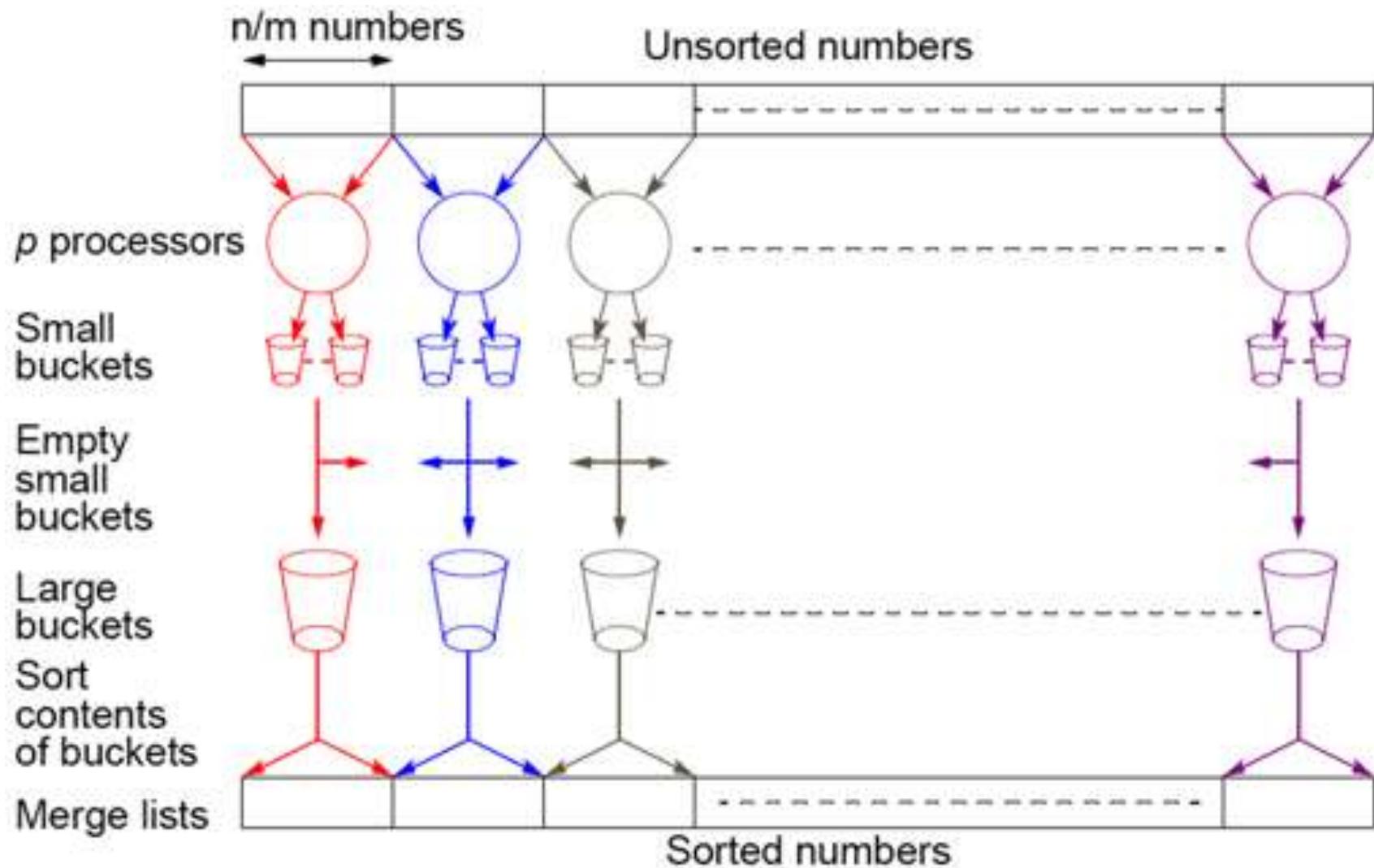
Further Parallelization

Partition sequence into m regions, one region for each processor.

Each processor maintains p “small” buckets and separates numbers in its region into its own small buckets.

Small buckets then emptied into p final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket i to processor j).

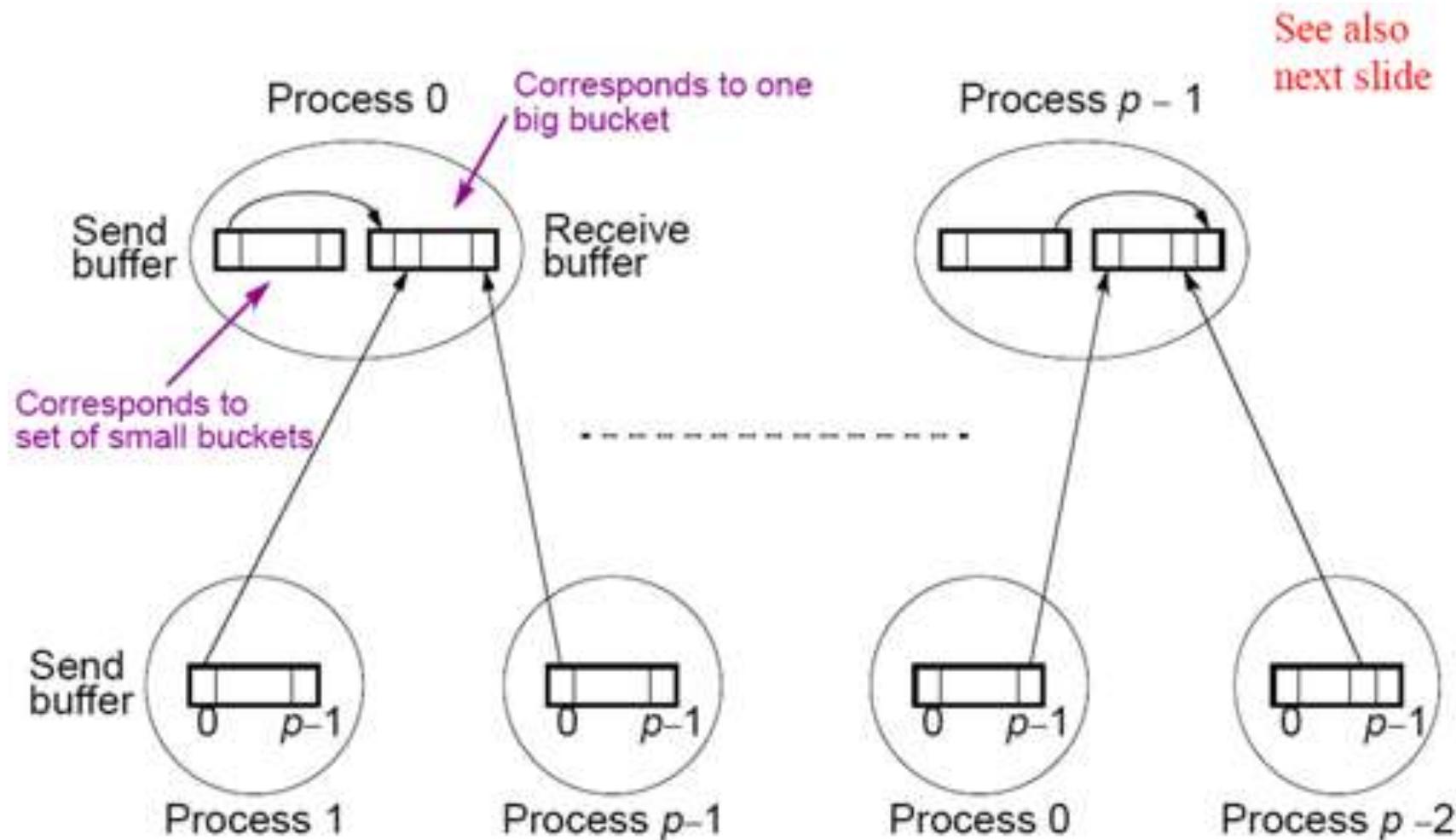
Another parallel version of bucket sort



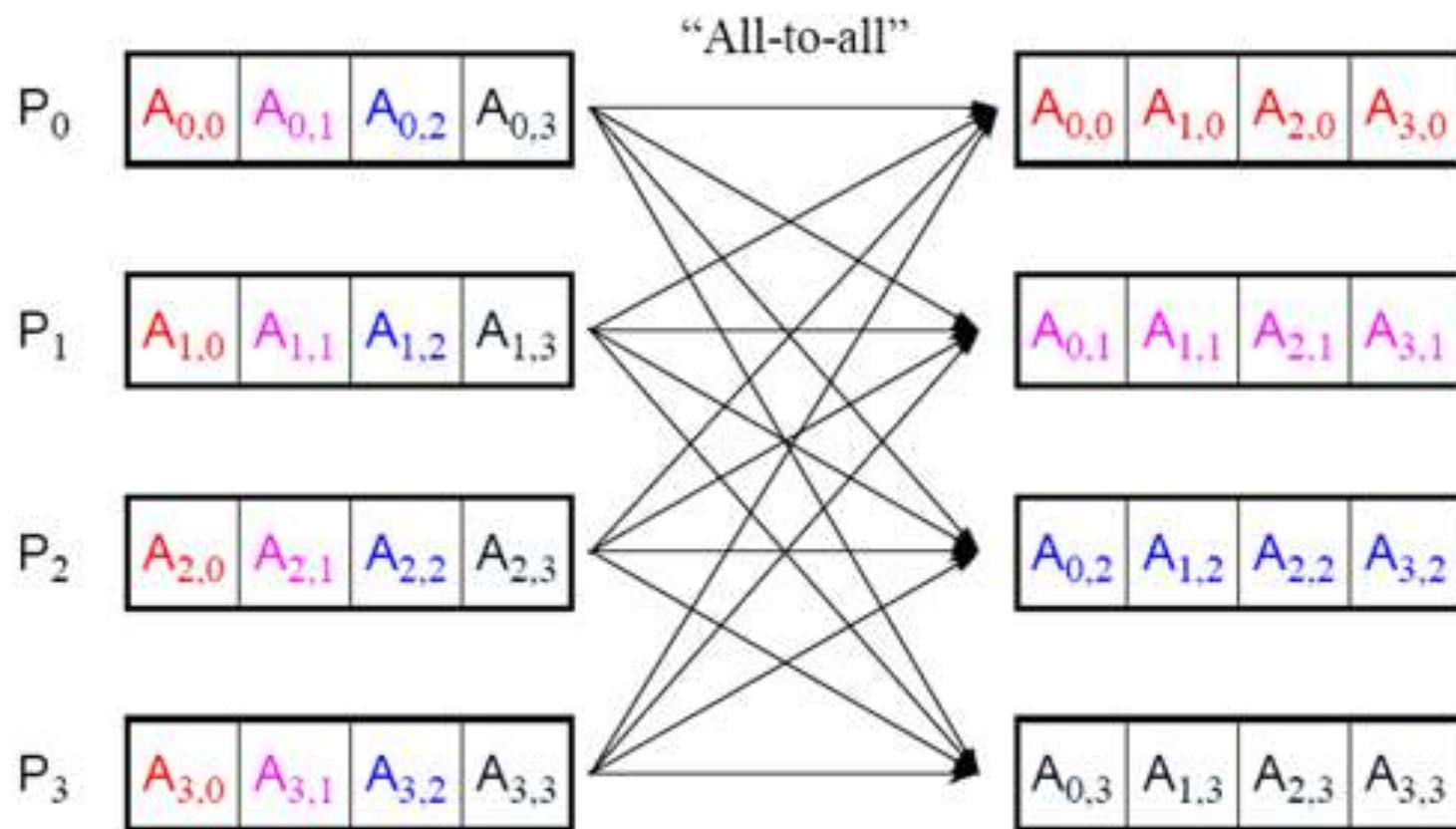
Introduces new message-passing operation - all-to-all broadcast.

“all-to-all” broadcast routine

Sends data from each process to every other process



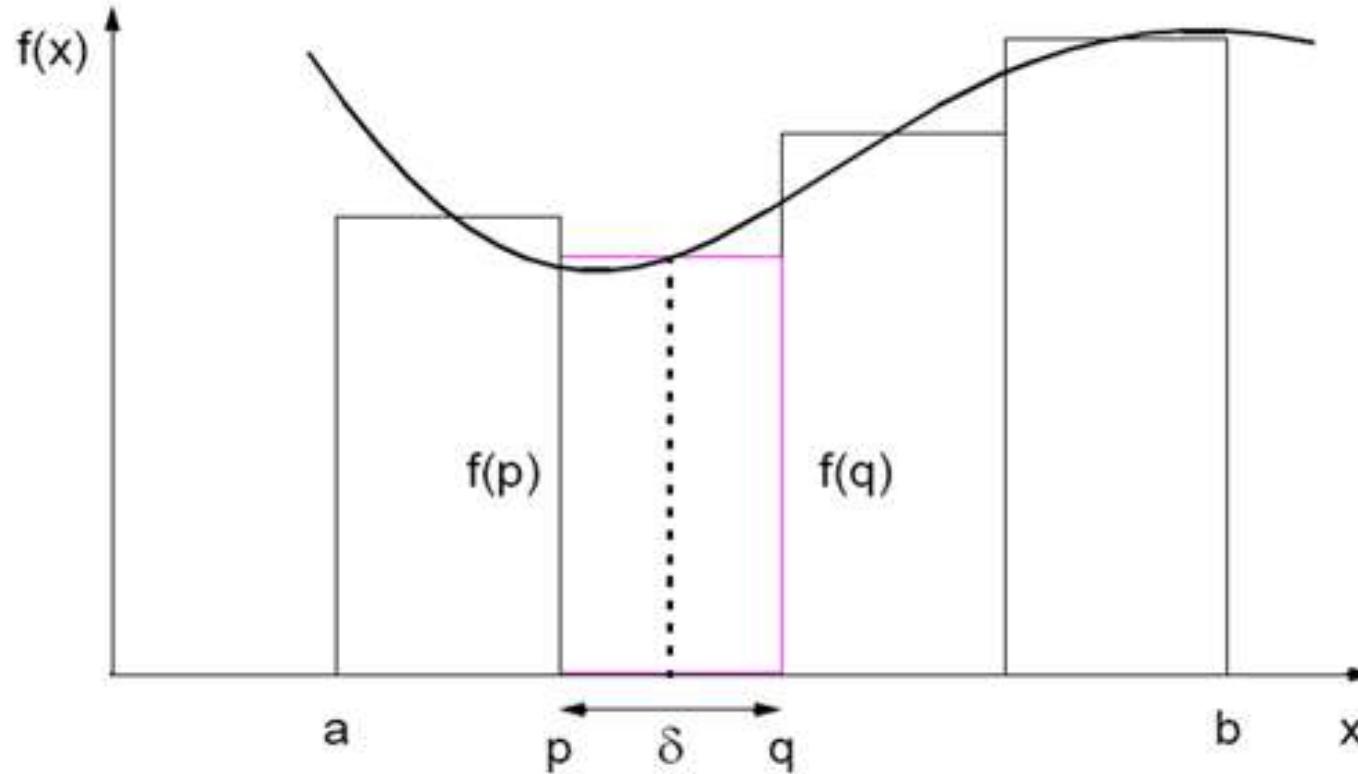
“all-to-all” routine actually transfers rows of an array to columns:
Transposes a matrix.



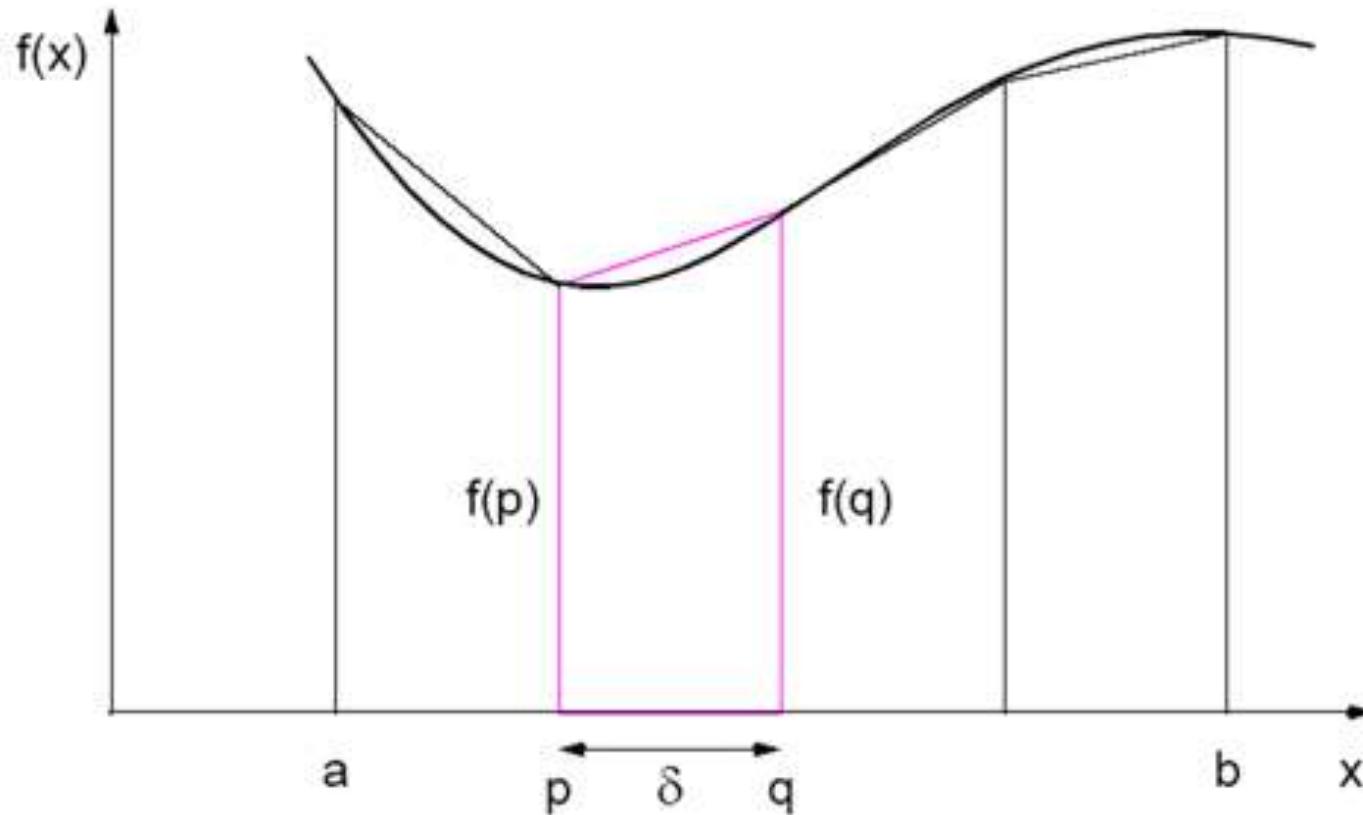
Numerical integration using rectangles

Each region calculated using an approximation given by rectangles:

Aligning the rectangles:



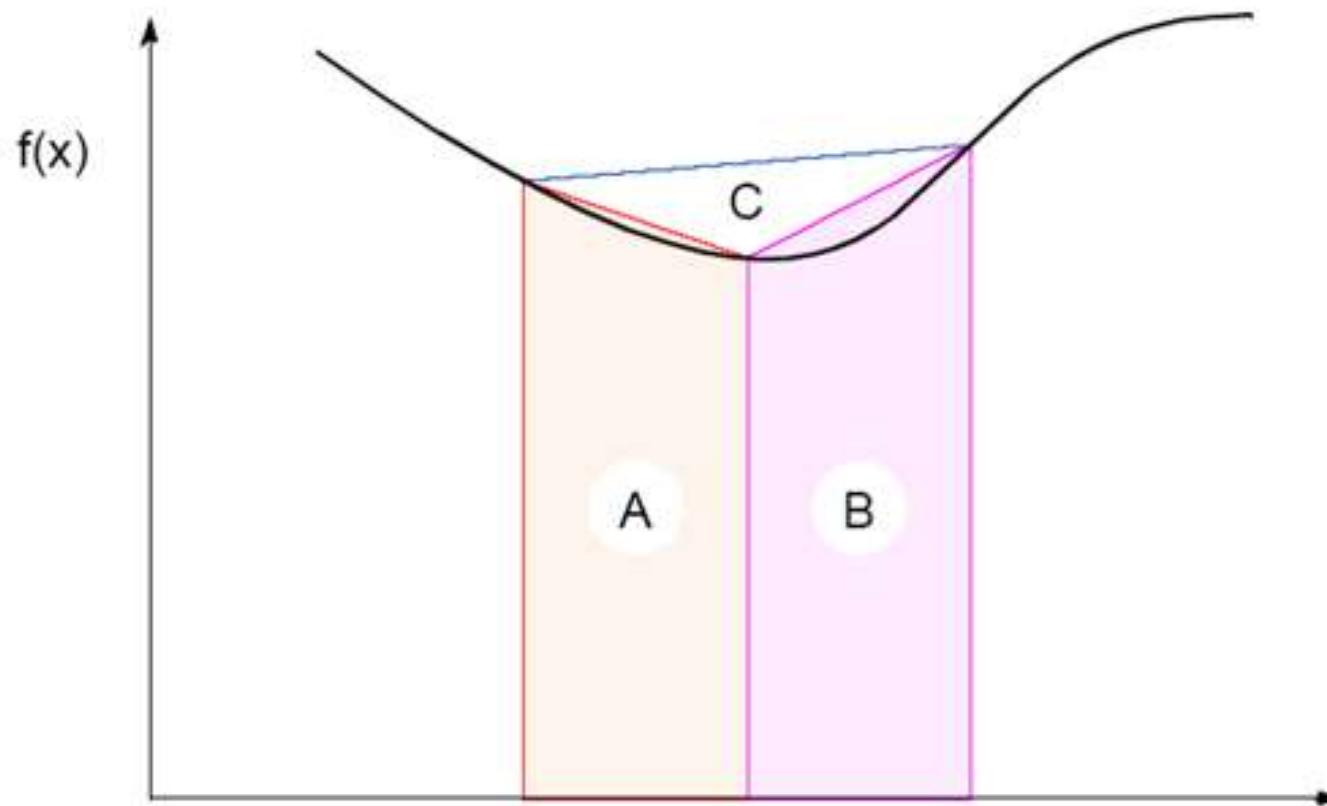
Numerical integration using trapezoidal method



May not be better!

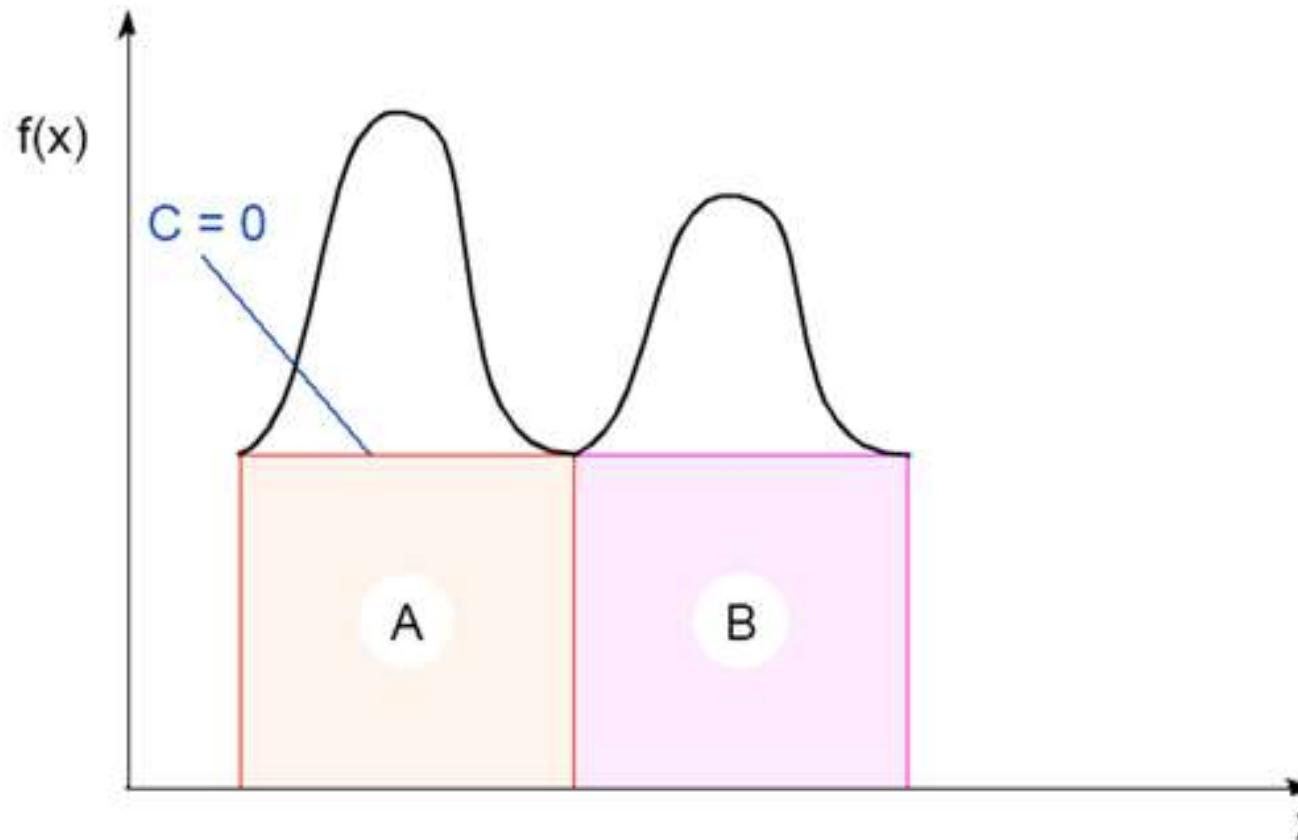
Adaptive Quadrature

Solution adapts to shape of curve. Use three areas, A , B , and C . Computation terminated when largest of A and B sufficiently close to sum of remain two areas .



Adaptive quadrature with false termination.

Some care might be needed in choosing when to terminate.



Might cause us to terminate early, as two large regions are the same (i.e., $C = 0$).

Simple program to compute π

Using C++ MPI routines

pi_calc.cpp calculates value of pi and compares with actual
value (to 25digits) of pi to give error. Integrates function $f(x)=4/(1+x^2)$.
July 6, 2001 K. Spry CSCI3145

```
#include <math.h> //include files
#include <iostream.h>
#include "mpi.h"

void printit();                                //function prototypes
int main(int argc, char *argv[])
{
double actual_pi = 3.141592653589793238462643;    //for comparison later
int n, rank, num_proc, i;
double temp_pi, calc_pi, int_size, part_sum, x;
char response = 'y', resp1 = 'y';
MPI::Init(argc, argv);                          //initiate MPI
```

```

num_proc = MPI::COMM_WORLD.Get_size();
rank = MPI::COMM_WORLD.Get_rank();
if (rank == 0) printit();           /* I am root node, print out welcome */

while (response == 'y') {
    if (resp1 == 'y') {
        if (rank == 0) {           /*I am root node*/
            cout << _____           " << endl;
            cout << "\nEnter the number of intervals: (0 will exit)" << endl;
            cin >> n;}
    } else n = 0;
}

```

```

MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0); //broadcast n
if (n==0) break; //check for quit condition

```

```

else {
    int_size = 1.0 / (double) n;                      //calcs interval size
    part_sum = 0.0;

    for (i = rank + 1; i <= n; i += num_proc)          //calcs partial sums
    {
        x = int_size * ((double)i - 0.5);
        part_sum += (4.0 / (1.0 + x*x));
    }
    temp_pi = int_size * part_sum;
}                                                       //collects all partial sums computes pi

MPI::COMM_WORLD.Reduce(&temp_pi,&calc_pi, 1,
MPI::DOUBLE, MPI::SUM, 0);

```

```

if (rank == 0) {                                /*I am server*/
    cout << "pi is approximately " << calc_pi
    << ". Error is " << fabs(calc_pi - actual_pi)
    << endl
    << "____"
    << endl;
}
}

if (rank == 0) { /*I am root node*/
    cout << "\nCompute with new intervals? (y/n)" << endl; cin >> resp1;
}
//end while
MPI::Finalize();                                //terminate MPI
return 0;
}                                                 //end main

```

//functions

```
void printit()
{
    cout << "\n*****\n"
    << "Welcome to the pi calculator!" << endl
    << "Programmer: K. Spry" << endl
    << "You set the number of divisions \nfor estimating the
integral:
\n\tf(x)=4/(1+x^2)"
    << endl
    << "*****\n" << endl;
```

} *//end printit*

Gravitational N-Body Problem

Finding positions and movements of bodies in space subject to gravitational forces from other bodies, using Newtonian laws of physics.

Gravitational N-Body Problem Equations

Gravitational force between two bodies of masses m_a and m_b is:

$$F = \frac{Gm_a m_b}{r^2}$$

G is the gravitational constant and r the distance between the bodies. Subject to forces, body accelerates according to Newton's 2nd law:

$$F = ma$$

m is mass of the body, F is force it experiences, and a the resultant acceleration.

Details

Let the time interval be Δt . For a body of mass m , the force is:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

New velocity is:

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where v^{t+1} is the velocity at time $t + 1$ and v^t is the velocity at time t .

Over time interval Δt , position changes by

$$x^{t+1} - x^t = v\Delta t$$

where x^t is its position at time t .

Once bodies move to new positions, forces change.

Computation has to be repeated.

Sequential Code

Overall gravitational N -body computation can be described by:

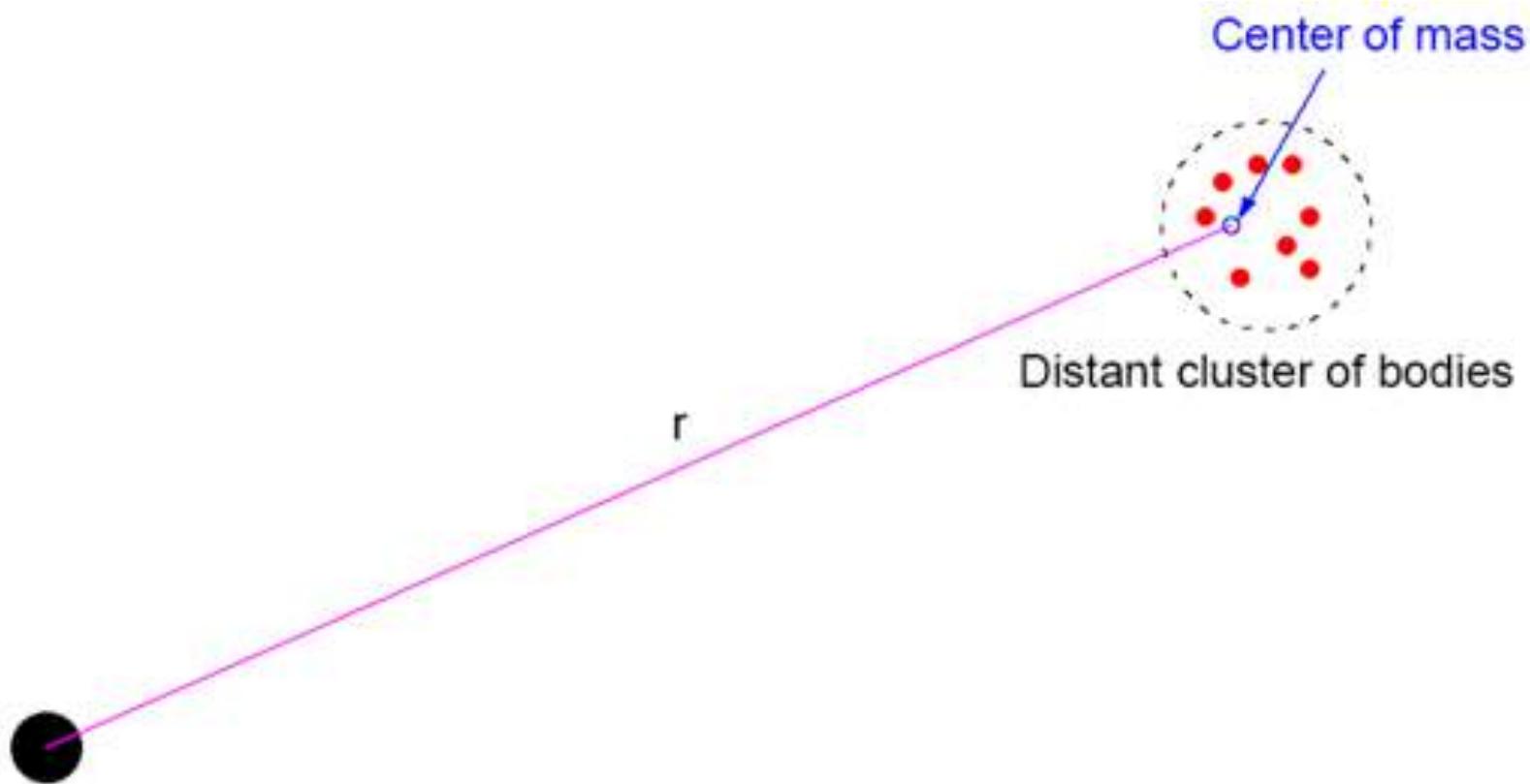
```
for (t = 0; t < tmax; t++)          /* for each time period */  
    for (i = 0; i < N; i++) {  
        F = Force_routine(i);      /* compute force on ith body */  
        v[i]new = v[i] + F * dt / m; /* compute new velocity */  
        x[i]new = x[i] + v[i]new * dt; /* and new position */  
    }  
    for (i = 0; i < nmax; i++) {  
        x[i] = x[i]new;           /* for each body */  
        v[i] = v[i]new;           /* update velocity & position */  
    }
```

Parallel Code

The sequential algorithm is an $O(N^2)$ algorithm (for one iteration) as each of the N bodies is influenced by each of the other $N - 1$ bodies.

Not feasible to use this direct algorithm for most interesting N -body problems where N is very large.

Time complexity can be reduced approximating a cluster of distant bodies as a single distant body with mass sited at the center of mass of the cluster:



Barnes-Hut Algorithm

Start with whole space in which one cube contains the bodies (or particles).

- First, this cube is divided into eight subcubes.
- If a subcube contains no particles, subcube deleted from further consideration.
- If a subcube contains one body, subcube retained.
- If a subcube contains more than one body, it is recursively divided until every subcube contains one body.

Creates an *octtree* - a tree with up to eight edges from each node.

The leaves represent cells each containing one body.

After the tree has been constructed, the total mass and center of mass of the subcube is stored at each node.

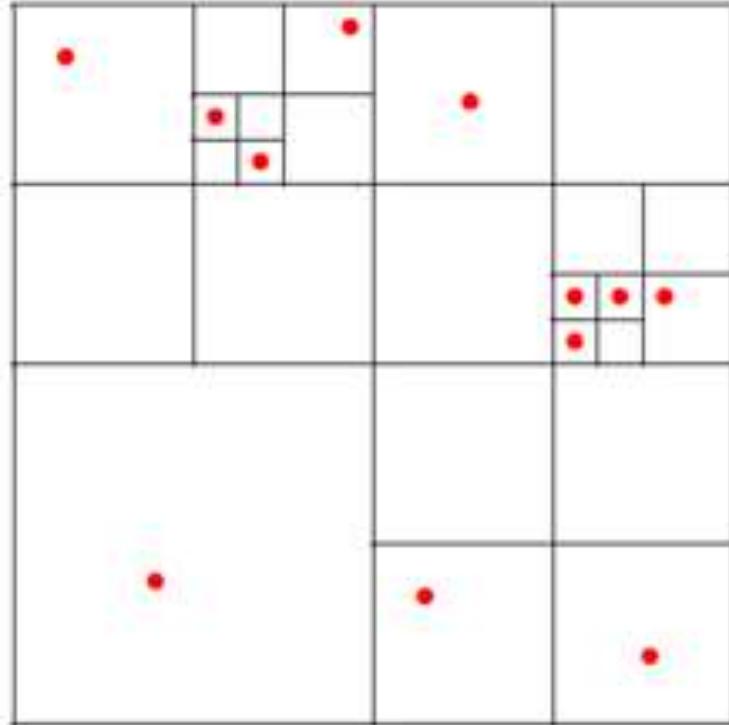
Force on each body obtained by traversing tree starting at root, stopping at a node when the clustering approximation can be used, e.g. when:

$$r \geq \frac{d}{\bar{\theta}}$$

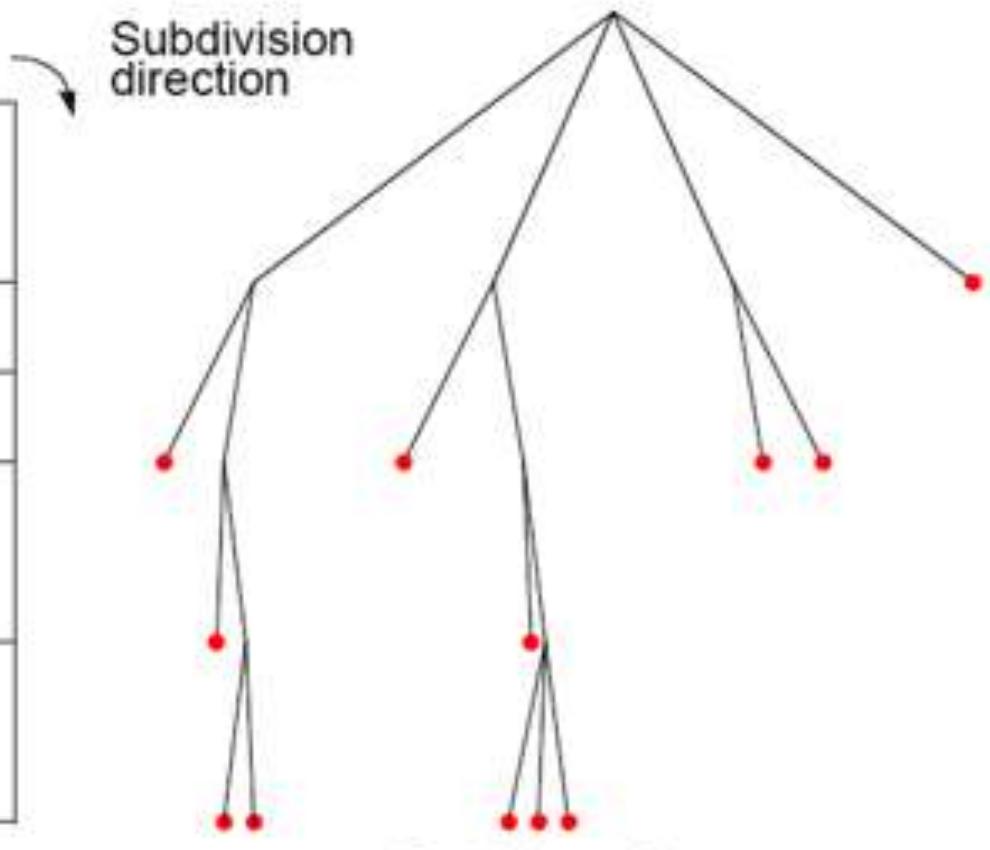
where θ is a constant typically 1.0 or less.

Constructing tree requires a time of $O(n \log n)$, and so does computing all the forces, so that overall time complexity of method is $O(n \log n)$.

Recursive division of 2-dimensional space



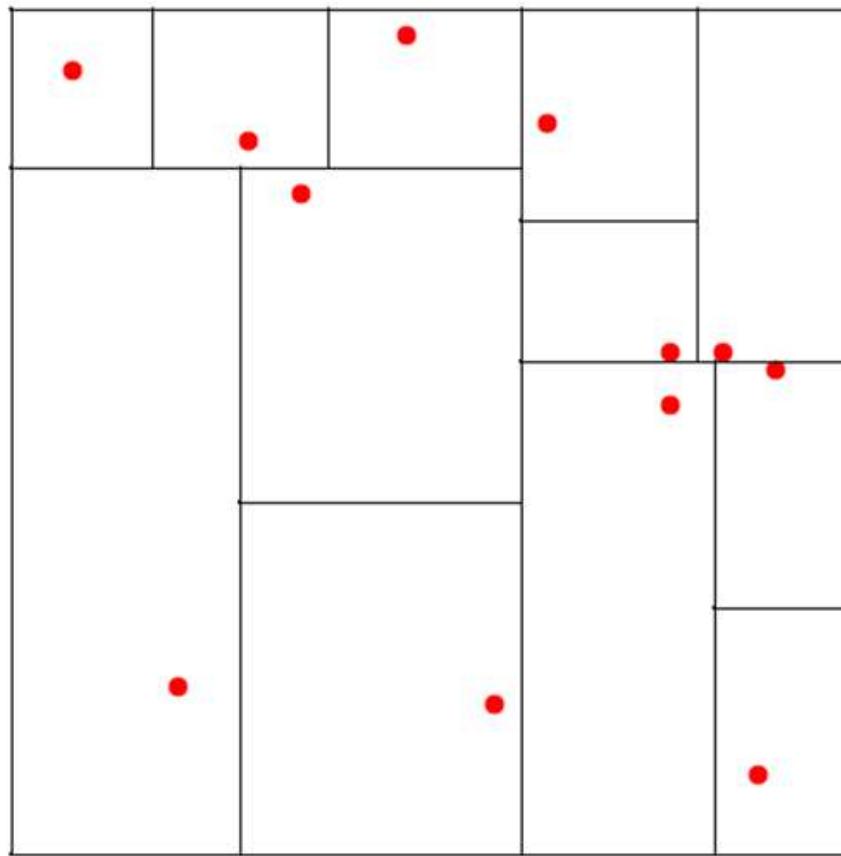
Particles



Partial quadtree

Orthogonal Recursive Bisection

(For 2-dimensional area) First, a vertical line found that divides area into two areas each with equal number of bodies. For each area, a horizontal line found that divides it into two areas each with equal number of bodies. Repeated as required.



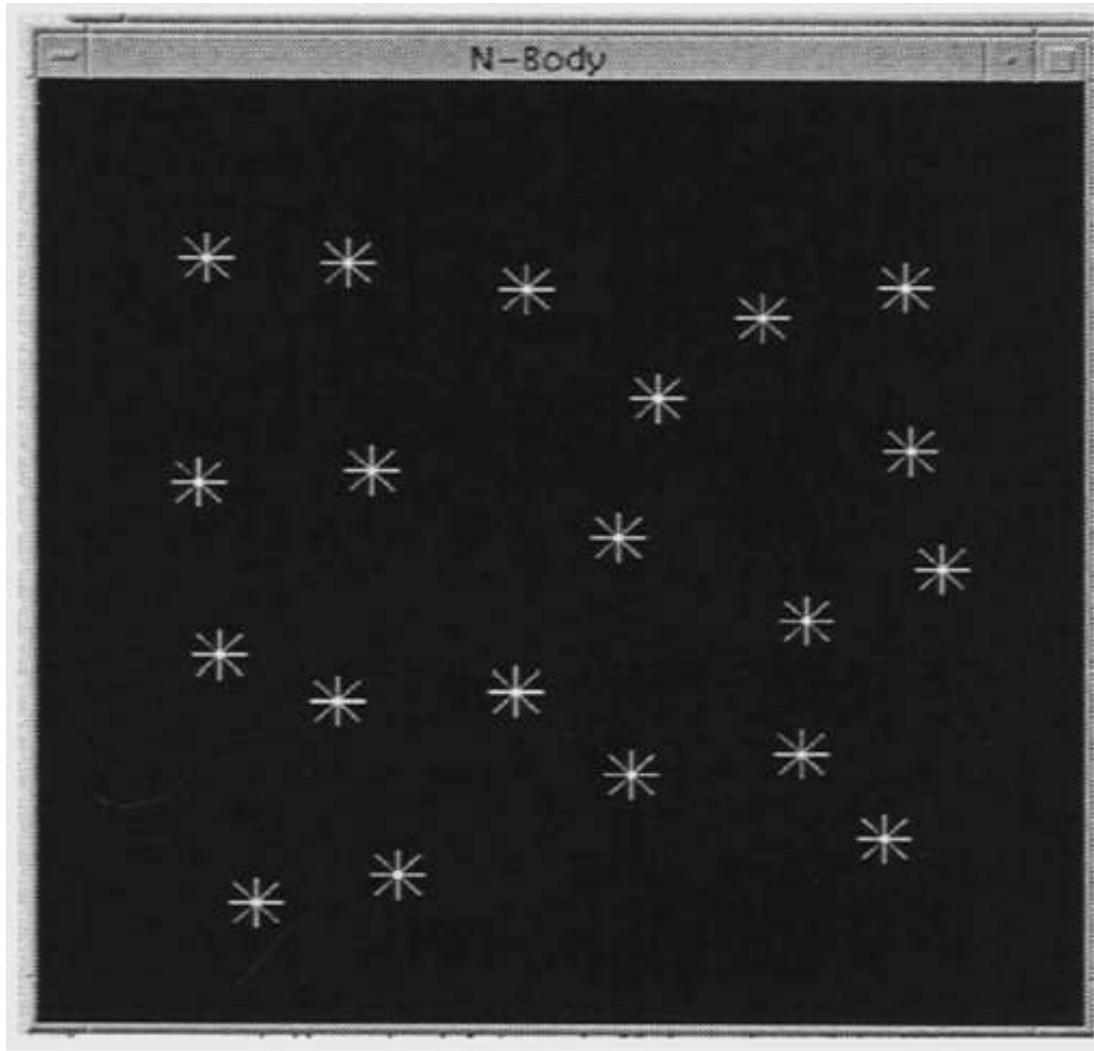
Astrophysical N-body simulation

By Scott Linssen (UNCC student, 1997) using O(N²) algorithm.



Astrophysical N -body simulation

By David Messager (UNCC student 1998) using Barnes-Hut algorithm.





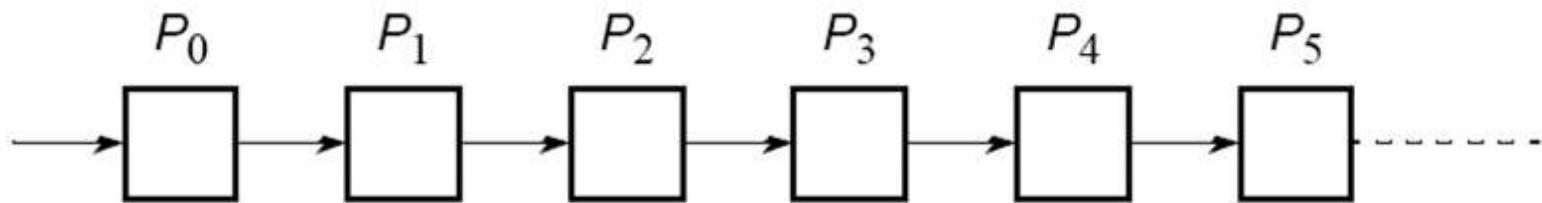
Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems Spring 2021

Pipelined Computations

Pipelined Computations

Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.



Example

Add all the elements of array **a** to an accumulating sum:

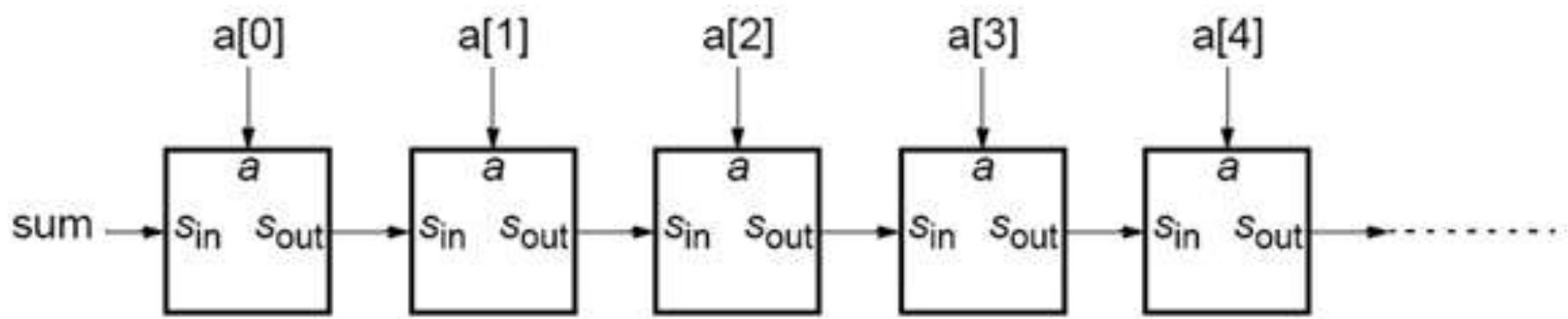
```
for (i = 0; i < n; i++)
    sum = sum + a[i];
```

The loop could be “unfolded” to yield

```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
```

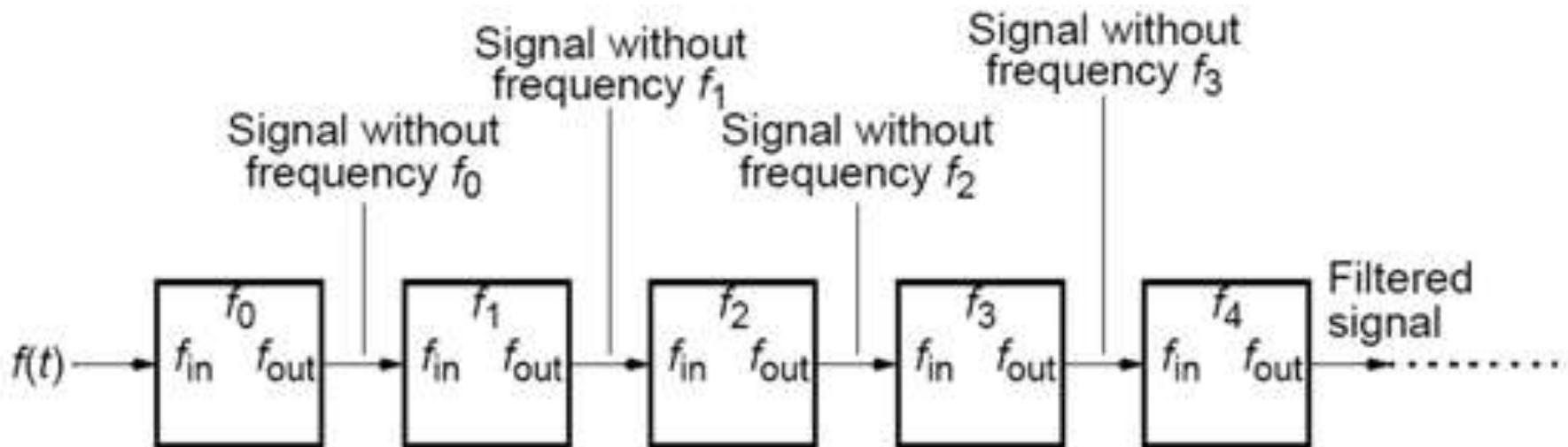
-
-
-

Pipeline for an unfolded loop



Another Example

Frequency filter - Objective to remove specific frequencies (f_0, f_1, f_2, f_3 , etc.) from a digitized signal, $f(t)$.
Signal enters pipeline from left:

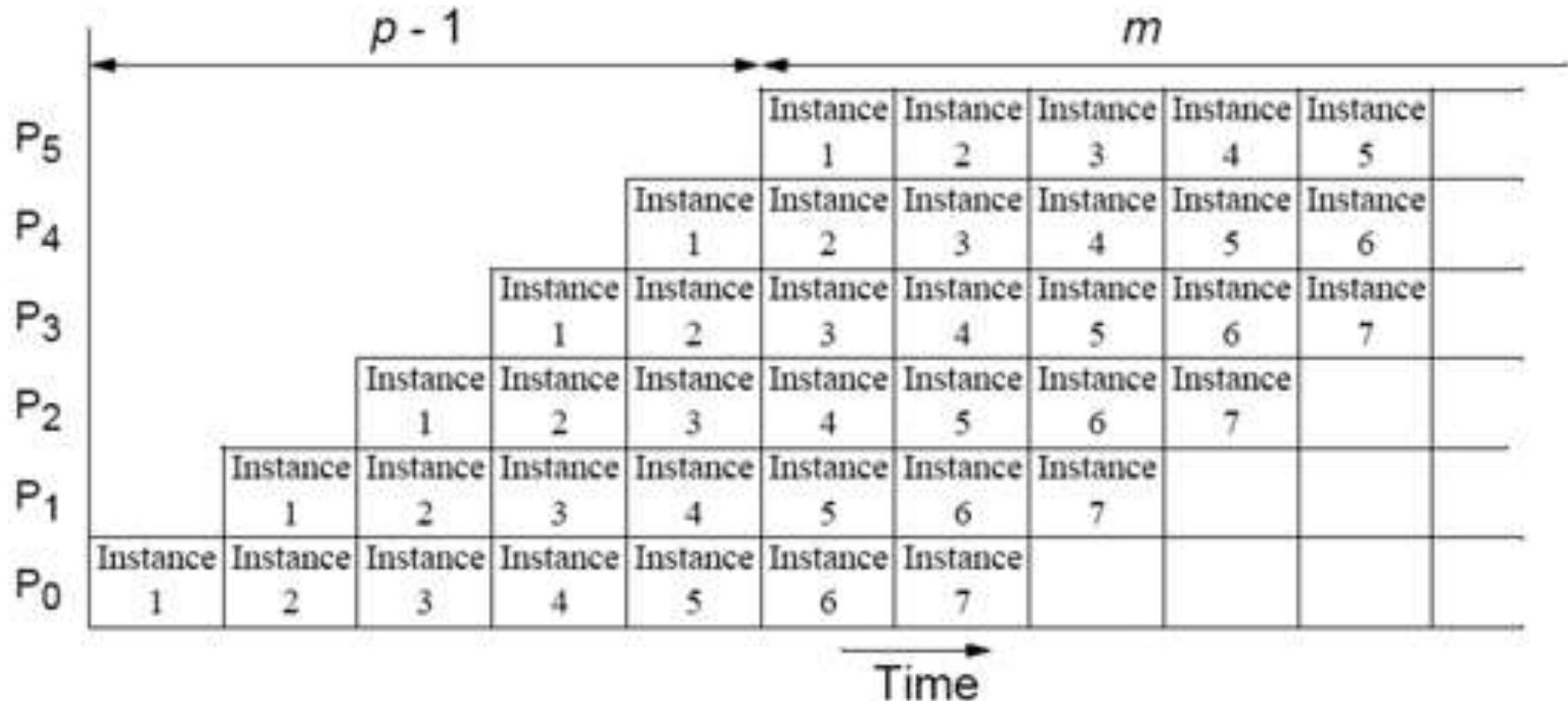


Where pipelining can be used to good effect

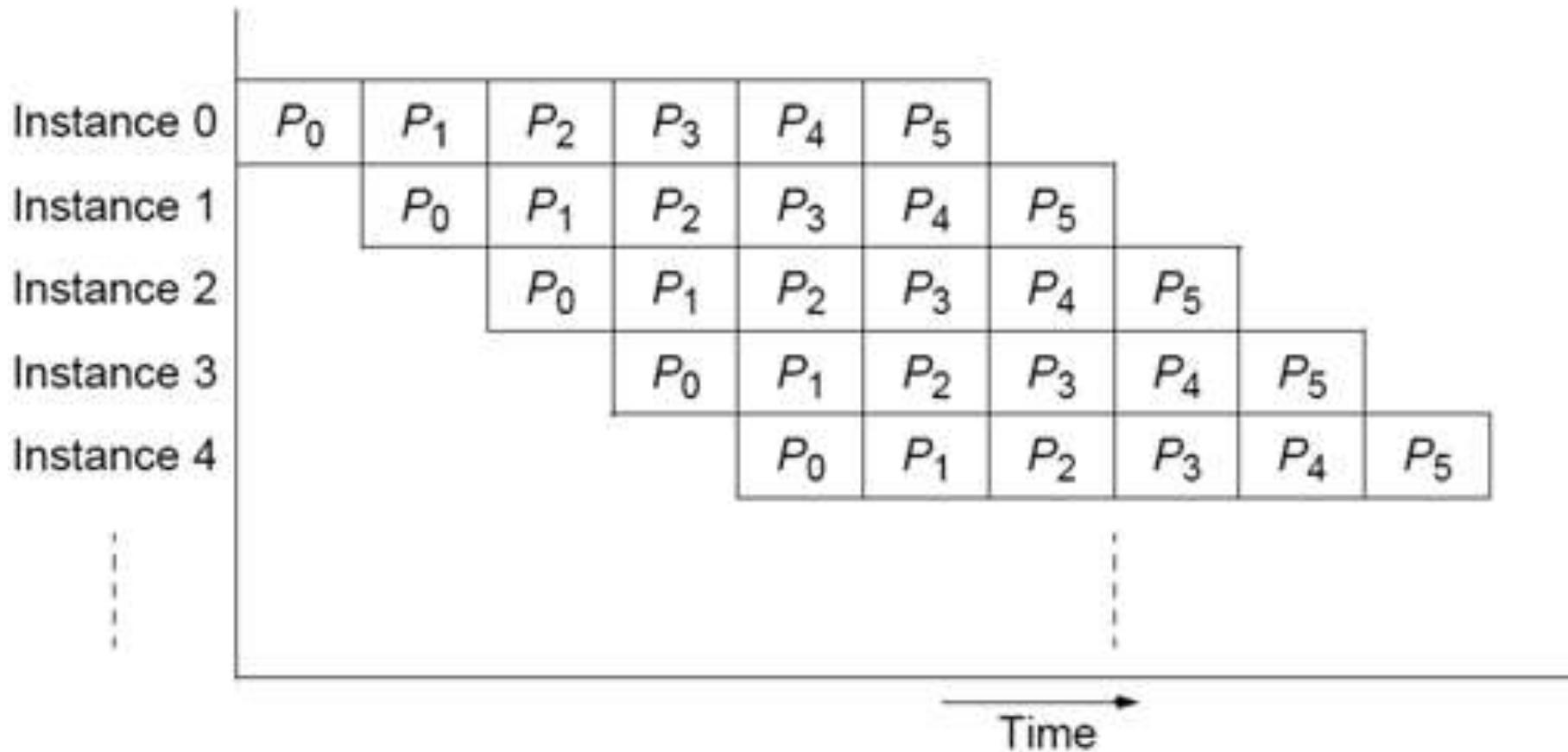
Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations:

1. If more than one instance of the complete problem is to be Executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start next process can be passed forward before process has completed all its internal operations

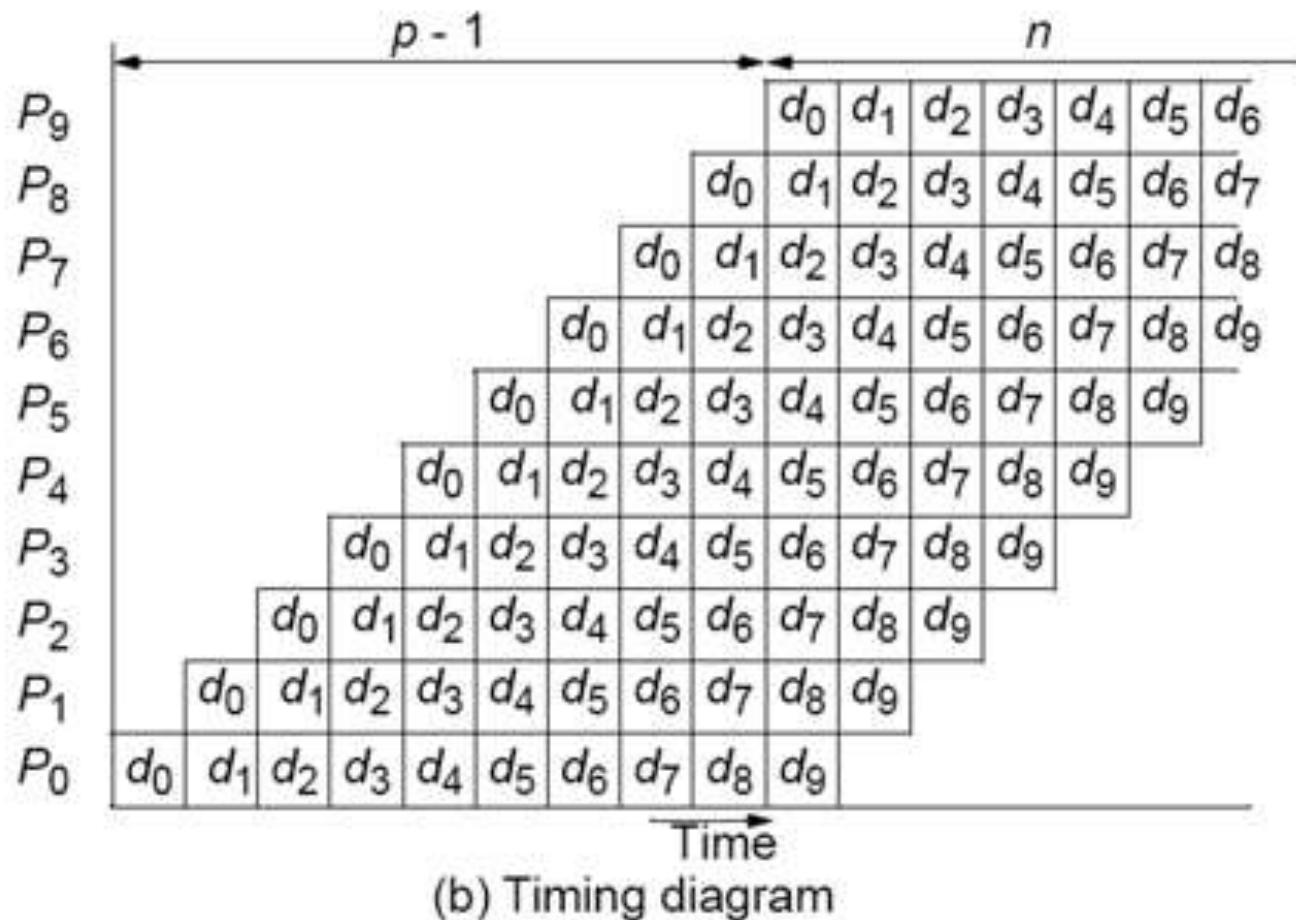
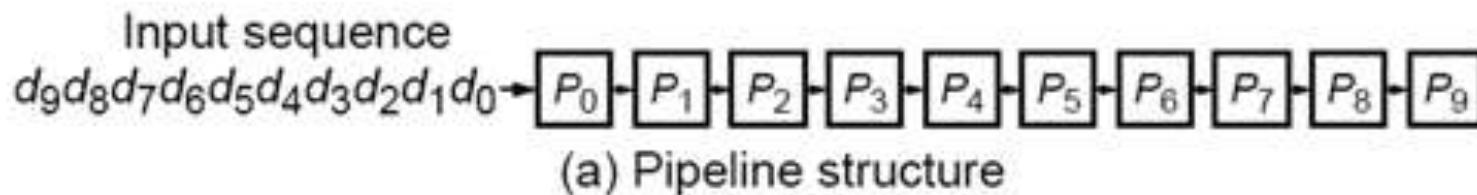
“Type 1” Pipeline Space-Time Diagram



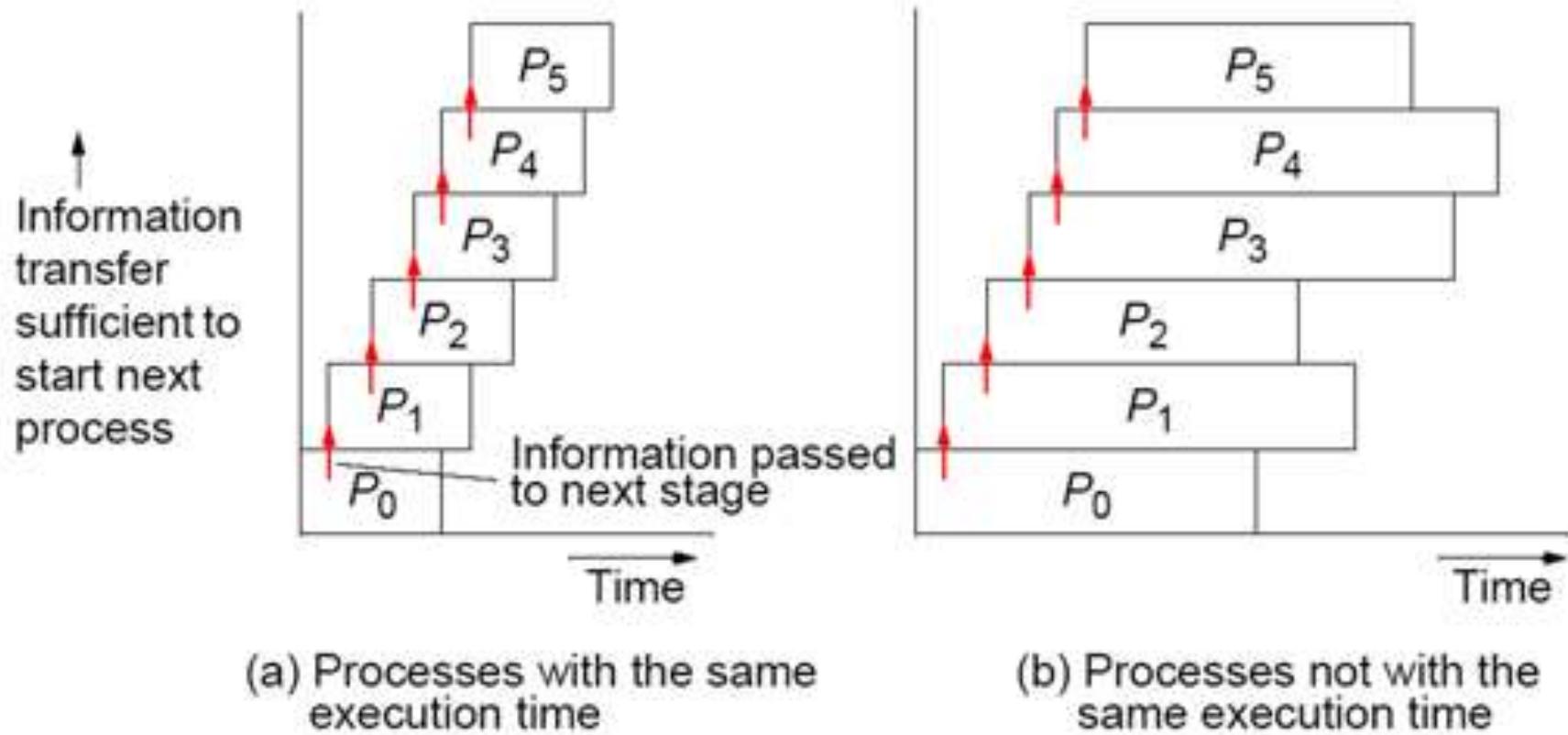
Alternative space-time diagram



“Type 2” Pipeline Space-Time Diagram

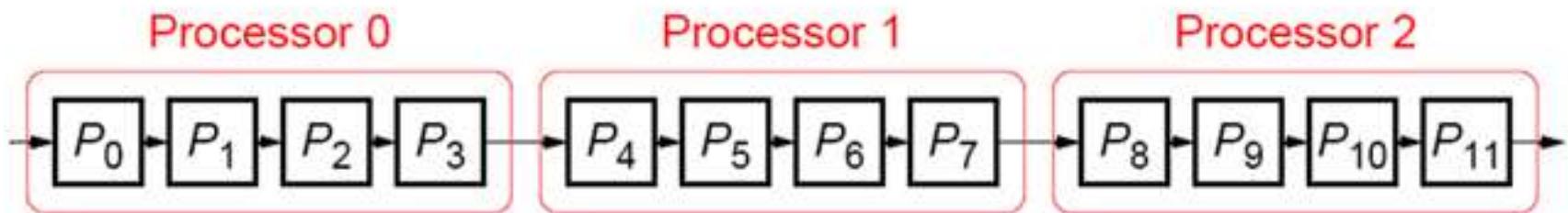


“Type 3” Pipeline Space-Time Diagram



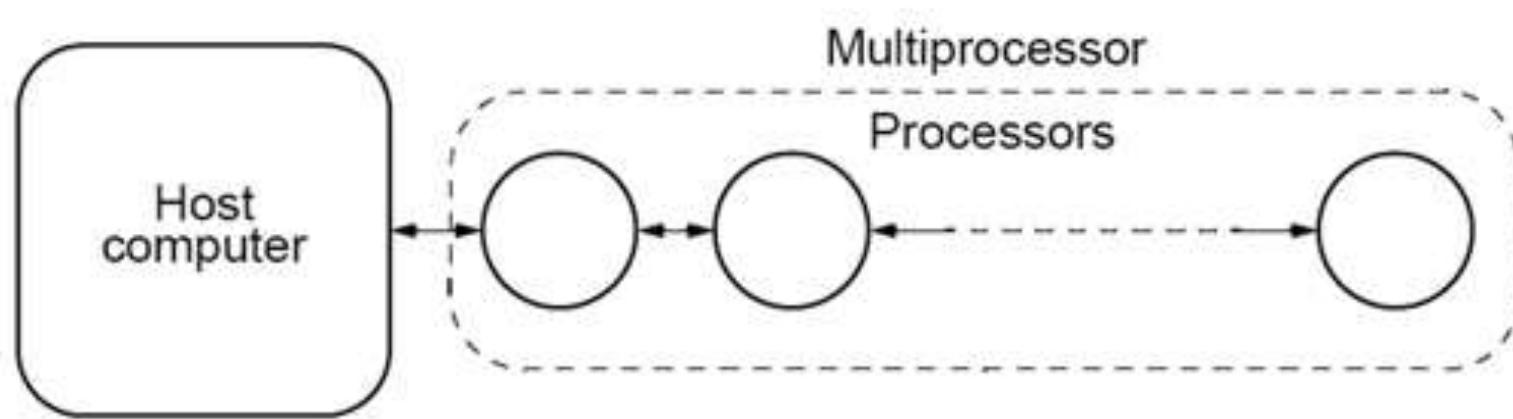
Pipeline processing where information passes to next stage before previous state completed.

If the number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor:



Computing Platform for Pipelined Applications

Multiprocessor system with a line configuration



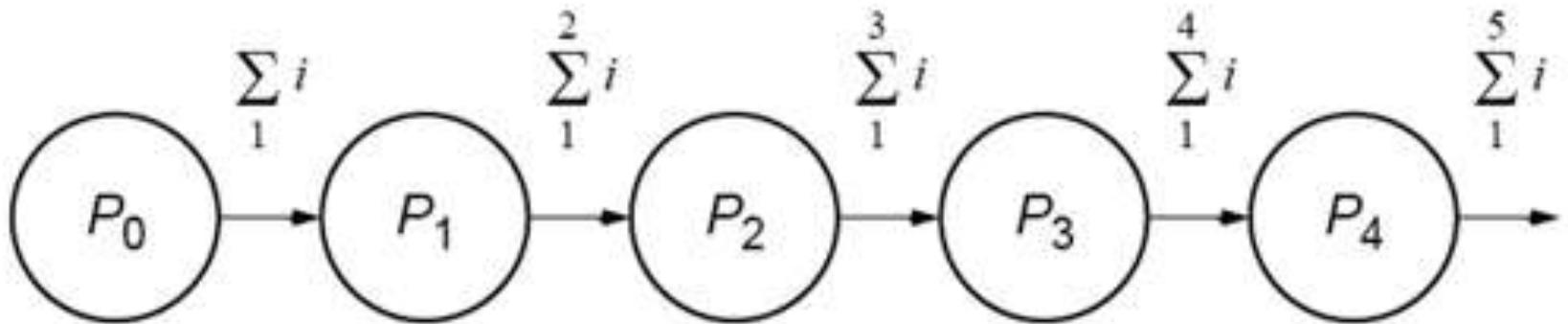
Strictly speaking pipeline may not be the best structure for a cluster - however a cluster with switched direct connections, as most have, can support simultaneous message passing.

Example Pipelined Solutions

(Examples of each type of computation)

Pipeline Program Examples

Adding Numbers



Type 1 pipeline computation

Basic code for process P_i :

```
recv(&accumulation, Pi-1);  
accumulation = accumulation + number;  
send(&accumulation, Pi+1);
```

except for the first process, P_0 , which is

```
send(&number, P1);
```

and the last process, P_{n-1} , which is

```
recv(&number, Pn-2);  
accumulation = accumulation + number;
```

SPMD program

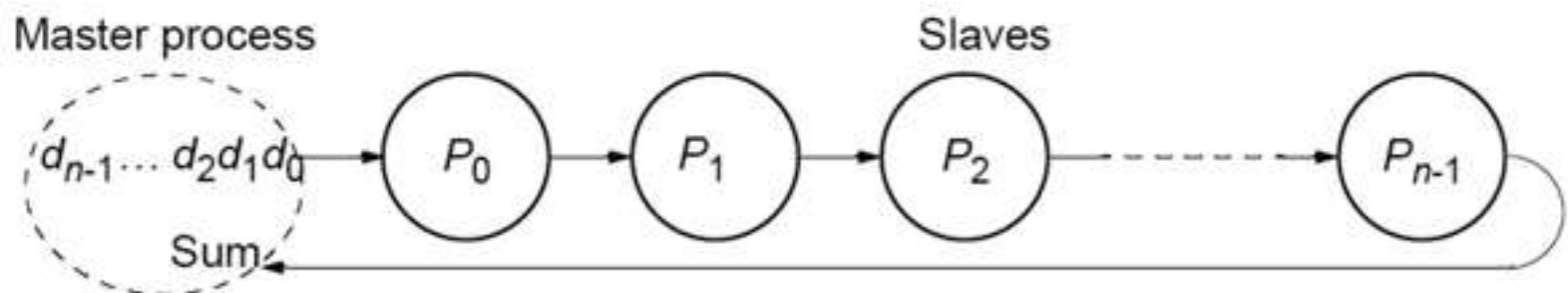
```
if (process > 0) {  
    recv(&accumulation, P i-1);  
    accumulation = accumulation + number;  
}  
if (process < n-1)  
    send(&accumulation, P i+1);
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

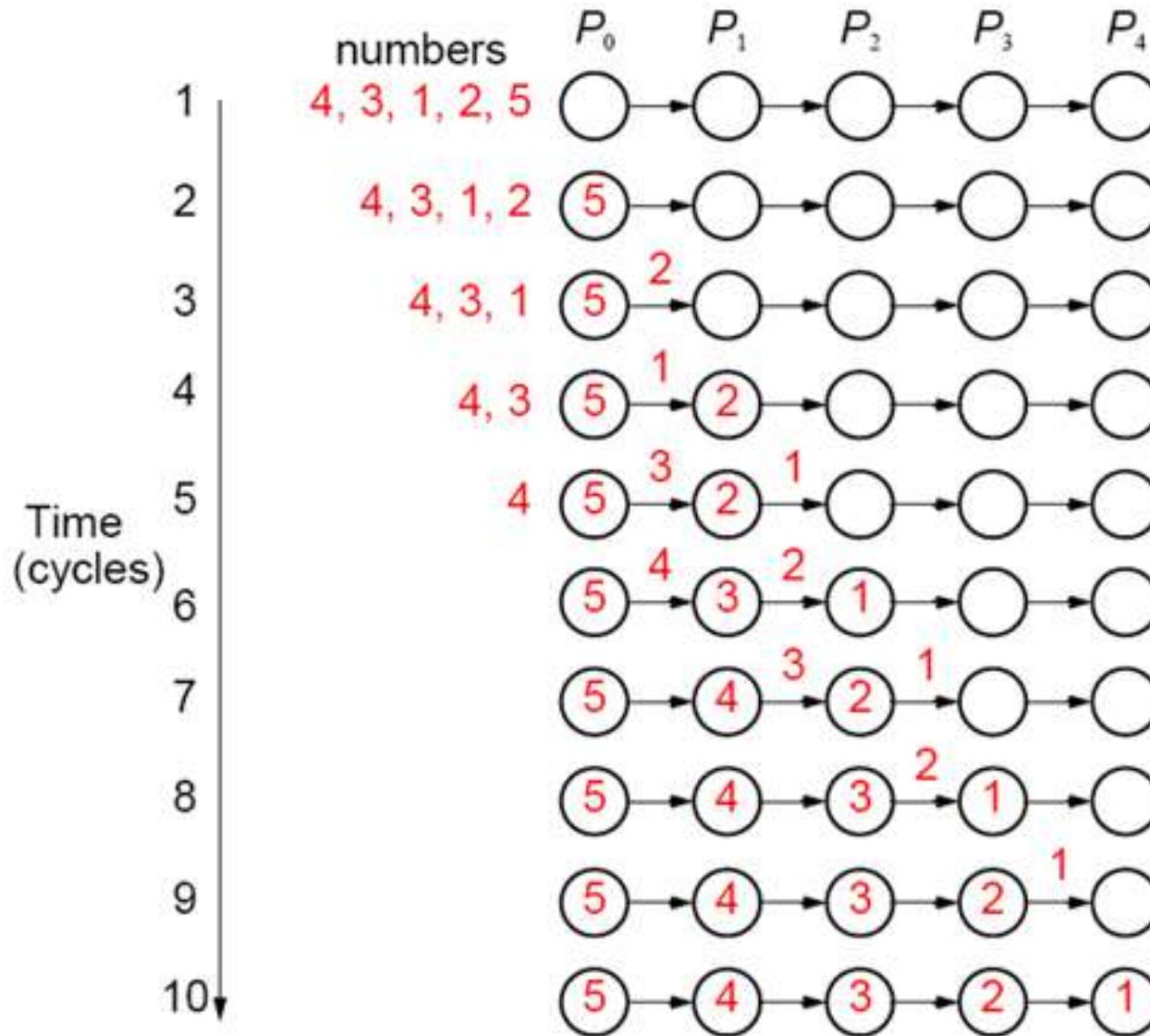
Pipelined addition numbers

Master process and ring configuration

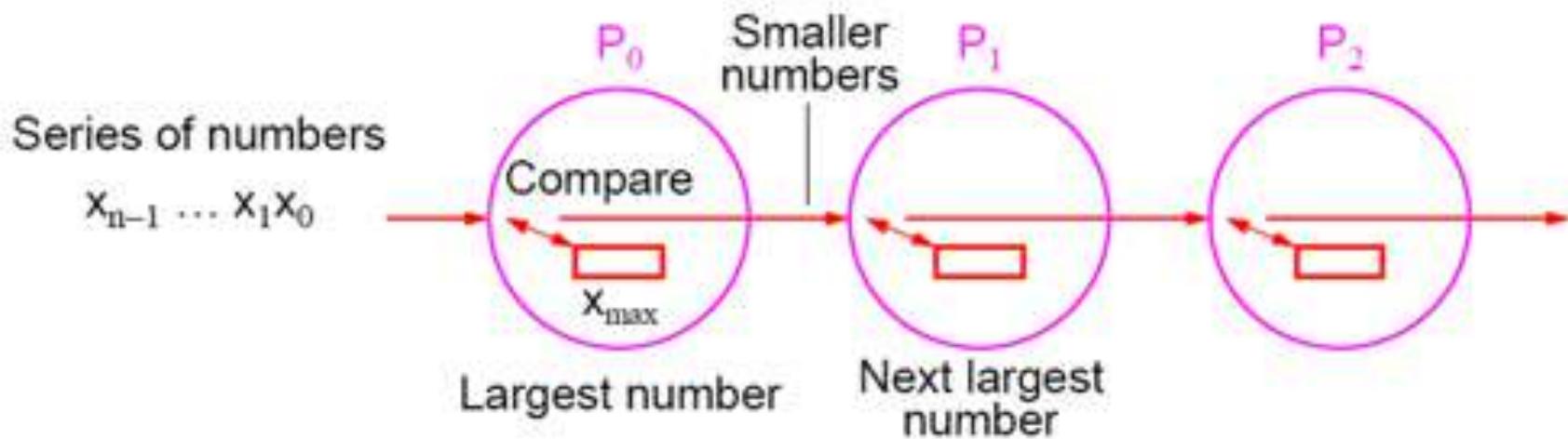


Sorting Numbers

A parallel version of *insertion sort*.



Pipeline for sorting using insertion sort



Type 2 pipeline computation

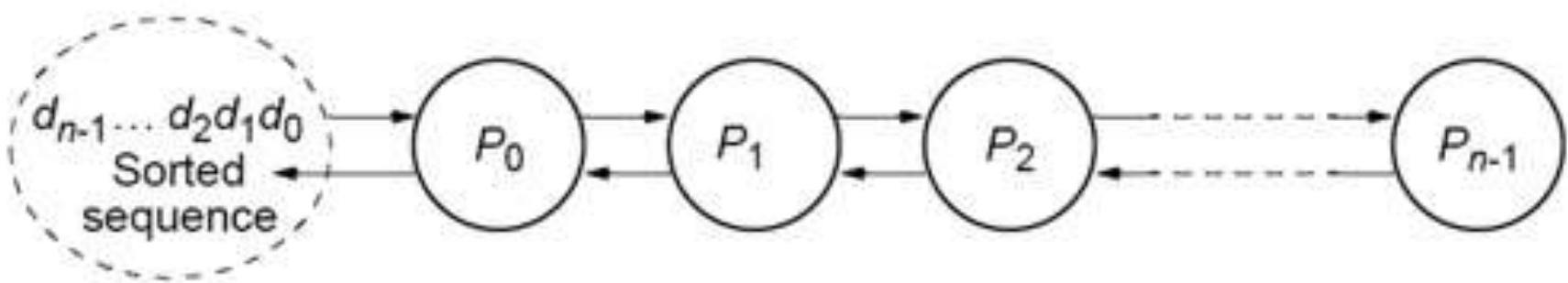
The basic algorithm for process P_i is

```
recv(&number, Pi-1);
if (number > x) {
    send(&x, Pi+1);
    x = number;
} else send(&number, Pi+1);
```

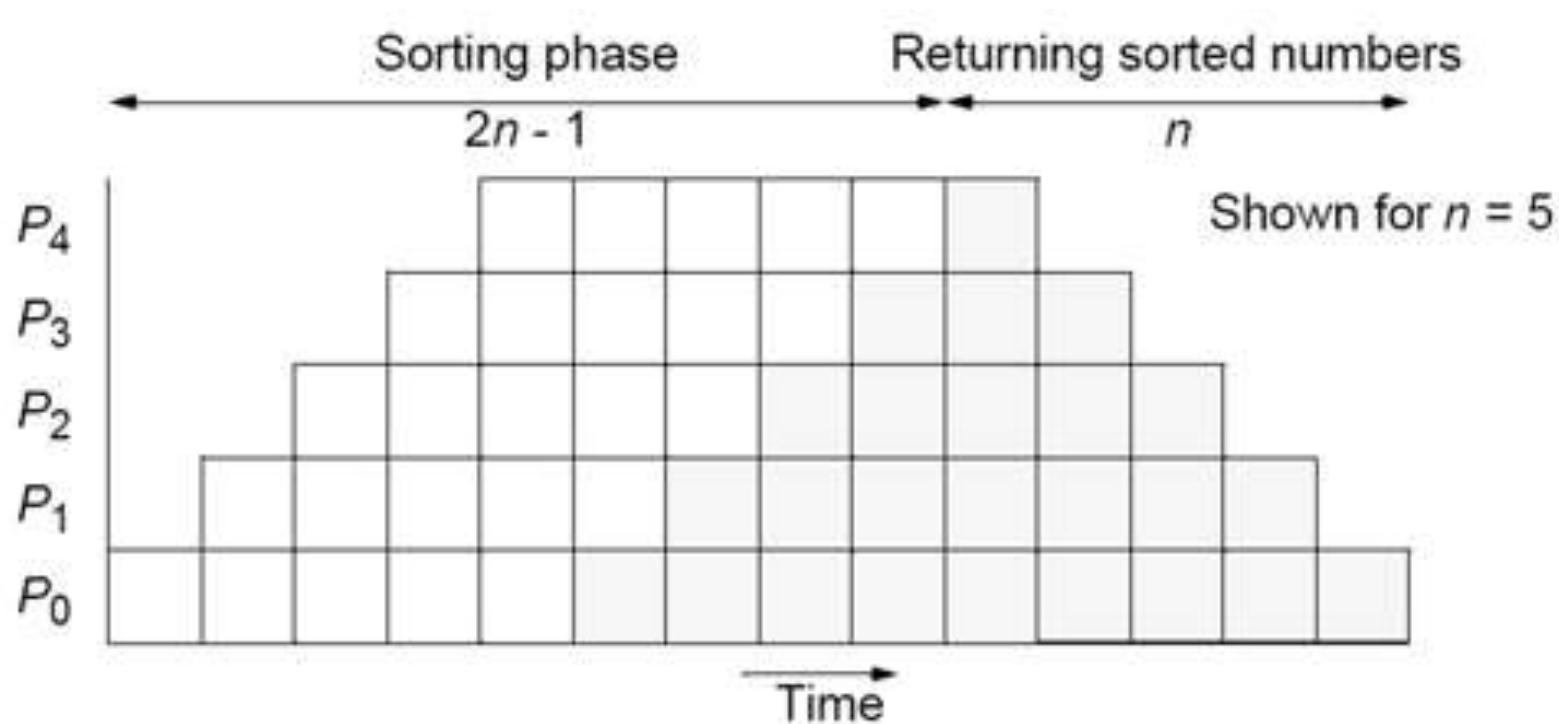
With n numbers, number i th process is to accept $= n - i$.
Number of passes onward $= n - i - 1$
Hence, a simple loop could be used.

Insertion sort with results returned to master process using bidirectional line configuration

Master process



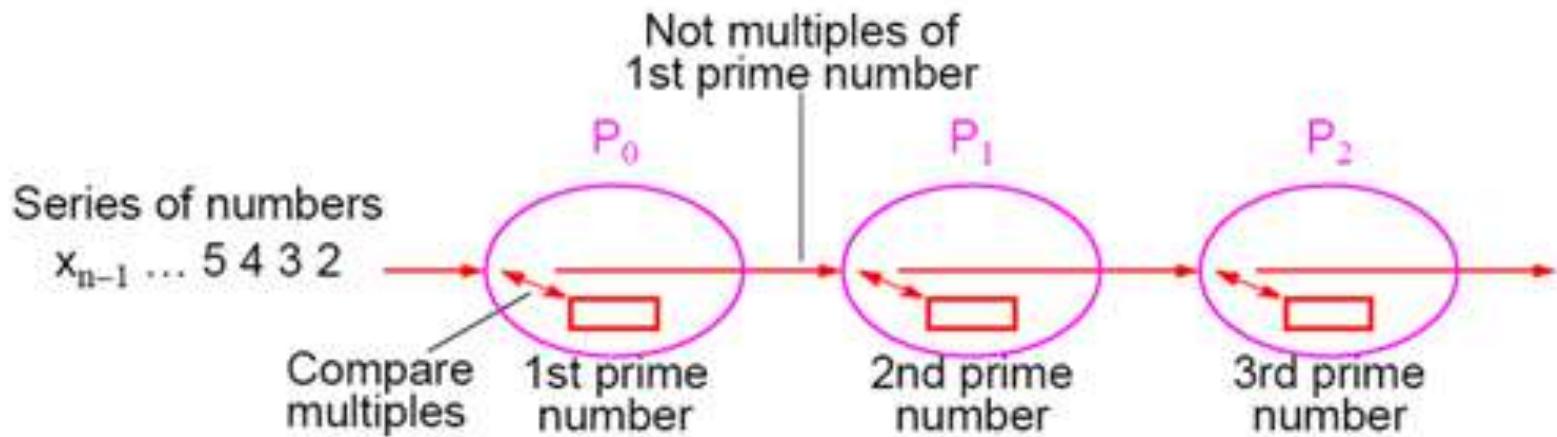
Insertion sort with results returned



Prime Number Generation

Sieve of Eratosthenes

- Series of all integers generated from 2.
- First number, 2, is prime and kept.
- All multiples of this number deleted as they cannot be prime.
- Process repeated with each remaining number.
- The algorithm removes non-primes, leaving only primes.



Type 2 pipeline computation

The code for a process, P_i , could be based upon

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, P i+1);
```

Each process will not receive the same number of numbers and is not known beforehand. Use a “terminator” message, which is sent at the end of the sequence:

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
    recv(&number, Pi-1);
    If (number == terminator) break;
    (number % x) != 0) send(&number, P i+1);
}
```

Solving a System of Linear Equations

Upper-triangular form

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \dots + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

.

.

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

where a 's and b 's are constants and x 's are unknowns to be found.

Back Substitution

First, unknown x_0 is found from last equation; i.e.,

$$x_0 = \frac{b_0}{a_{0,0}}$$

Value obtained for x_0 substituted into next equation to obtain x_1 ; i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

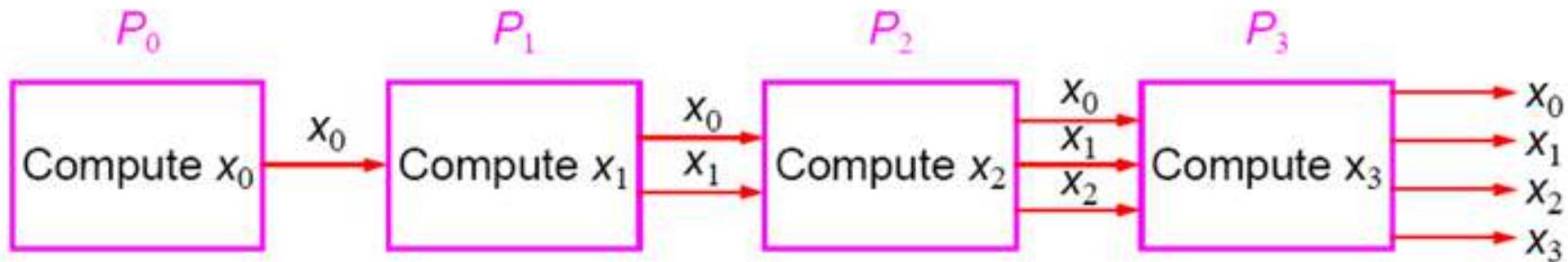
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2 :

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

Pipeline Solution

First pipeline stage computes x_0 and passes x_0 onto the second stage, which computes x_1 from x_0 and passes both x_0 and x_1 onto the next stage, which computes x_2 from x_0 and x_1 , and so on.



Type 3 pipeline computation

The i th process ($0 < i < n$) receives the values $x_0, x_1, x_2, \dots, x_{i-1}$ and computes x_i from the equation:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}}$$

Sequential Code

Given constants $a_{i,j}$ and b_k stored in arrays $\mathbf{a[][]}$ and $\mathbf{b[]}$, respectively, and values for unknowns to be stored in array, $\mathbf{x[]}$, sequential code could be

```
x[0] = b[0]/a[0][0];          /* computed separately */  
for (i = 1; i < n; i++) {      /*for remaining unknowns*/  
    sum = 0;  
    For (j = 0; j < i; j++)  
        sum = sum + a[i][j]*x[j];  
    x[i] = (b[i] - sum)/a[i][i];  
}
```

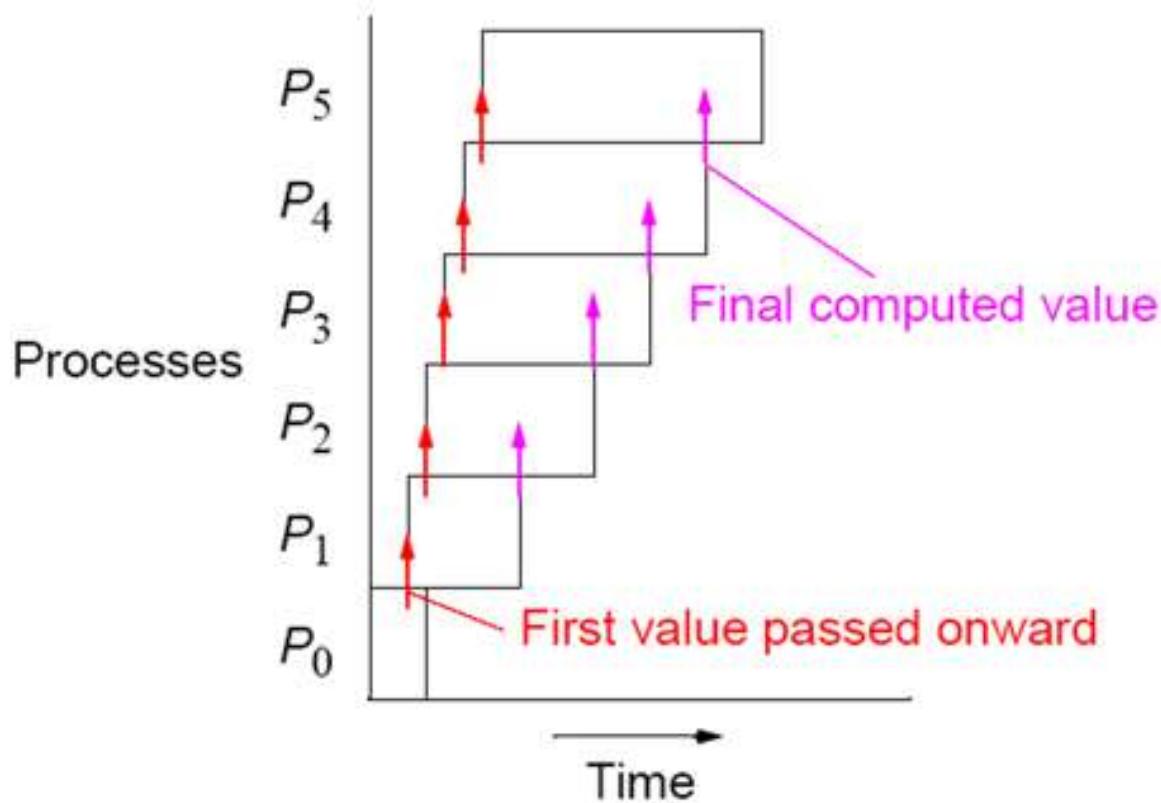
Parallel Code

Pseudocode of process P_i ($1 < i < n$) of could be

```
Sum=0
for (j = 0; j < i; j++) {
    recv(&x[j], Pi-1);
    send(&x[j], Pi+1);
    sum = sum + a[i][j]*x[j];
}
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);
```

Now have additional computations to do after receiving and resending values.

Pipeline processing using back substitution





Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Programming with Shared Memory

Multiprocessors and Multicore Processors

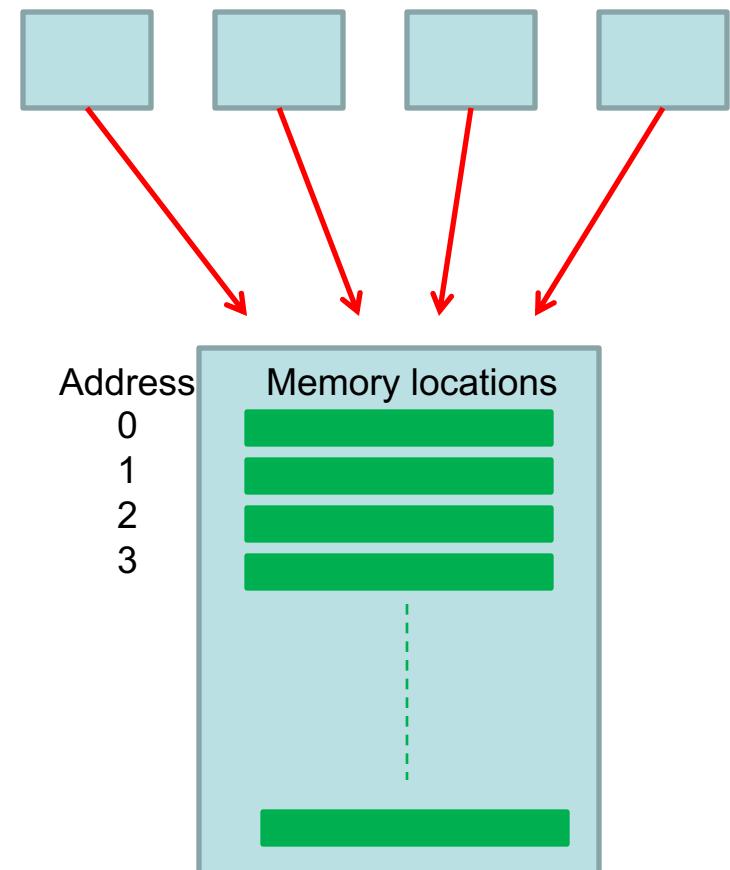
Shared memory multiprocessor/multicore processor system

Single address space exists – each memory location given unique address within single range of addresses.

Any memory location can be accessible by any of the processors.

Programming can take advantage of shared memory for holding data. However access to shared data by different processors needs to be carefully controlled, usually explicitly by programmer.

Processors or processor cores



Alternatives for Programming Shared Memory Multiprocessors:

- Using heavy weight processes.
- Using threads. Example Pthreads
- Using a completely new programming language for parallel programming - not popular. Example Ada.
- Using library routines with an existing sequential programming language.
- Modifying the syntax of an existing sequential programming language to create a parallel programming language. Example UPC
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism. Example OpenMP

Concept of a process

Basically a self-contained program having its own allocation of memory, stack, registers, instruction pointer, and other resources.

Operating systems often based upon notion of a process.

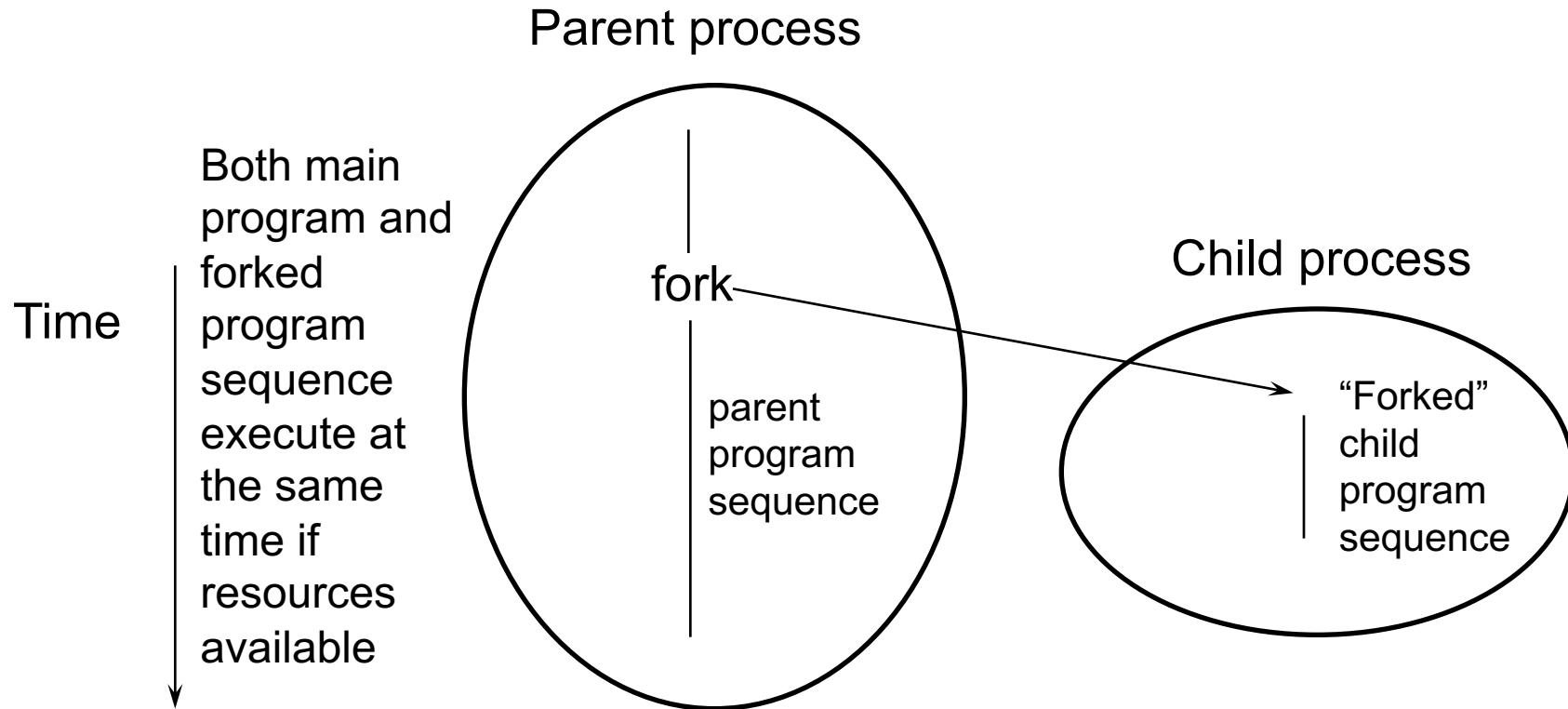
Processor time shared between processes, switching from one process to another. Might occur at regular intervals or when an active process becomes delayed.

Offers opportunity to de-schedule processes blocked from proceeding for some reason, e.g. waiting for an I/O operation to complete.

Process is the basic execution unit in message-passing MPI.

Fork pattern

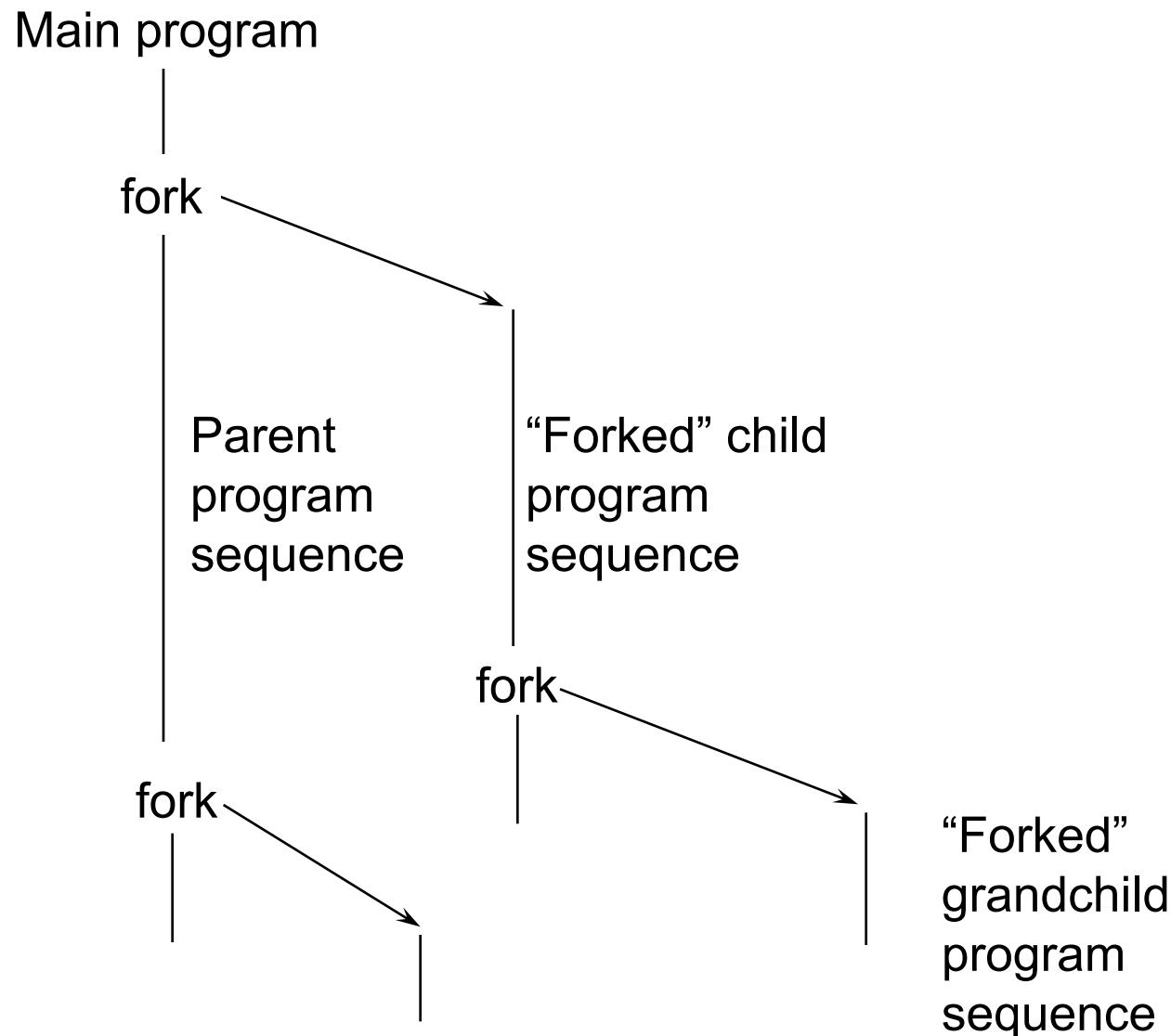
As used to dynamically create a process from a process



Although general concept of a fork does not require it, child process created by the Linux fork is a replica of parent program with same instructions and variable declarations even prior to fork. However, child process only starts at fork and both parent and child process execute onwards together.

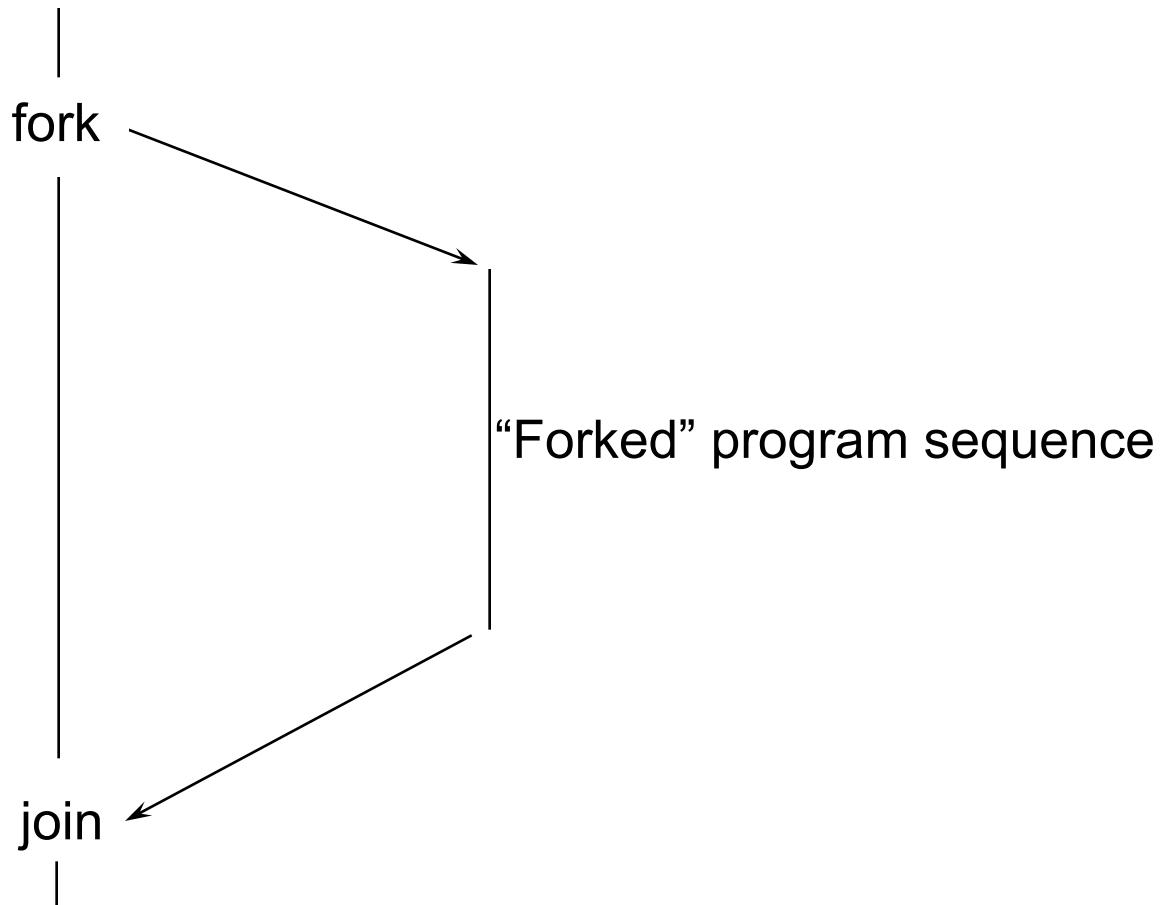
Multiple and nested fork patterns

Both main program and forked program sequence execute at the same time if resources available



Fork-join pattern

Main program



Explicit “join” placed in calling parent program. Parent will not proceed past this point until child has terminated. Join acts a barrier synchronization point for both sequences. Child can terminate before join is reached, but if not, parent will wait for it terminate.

UNIX System Calls to Create Fork-Join Pattern

No join routine – use **exit()** to exit from process and **wait()** to wait for child to complete:

```
    .
    .
pid = fork(); // returns 0 to child and positive # to parent (-1 if error)
if (pid == 0) {
    // code to be executed by child
} else {
    //code to be executed by parent
}
if (pid == 0) exit(0); else wait (0);      // join
    .
    .
```

Using processes in shared memory programming

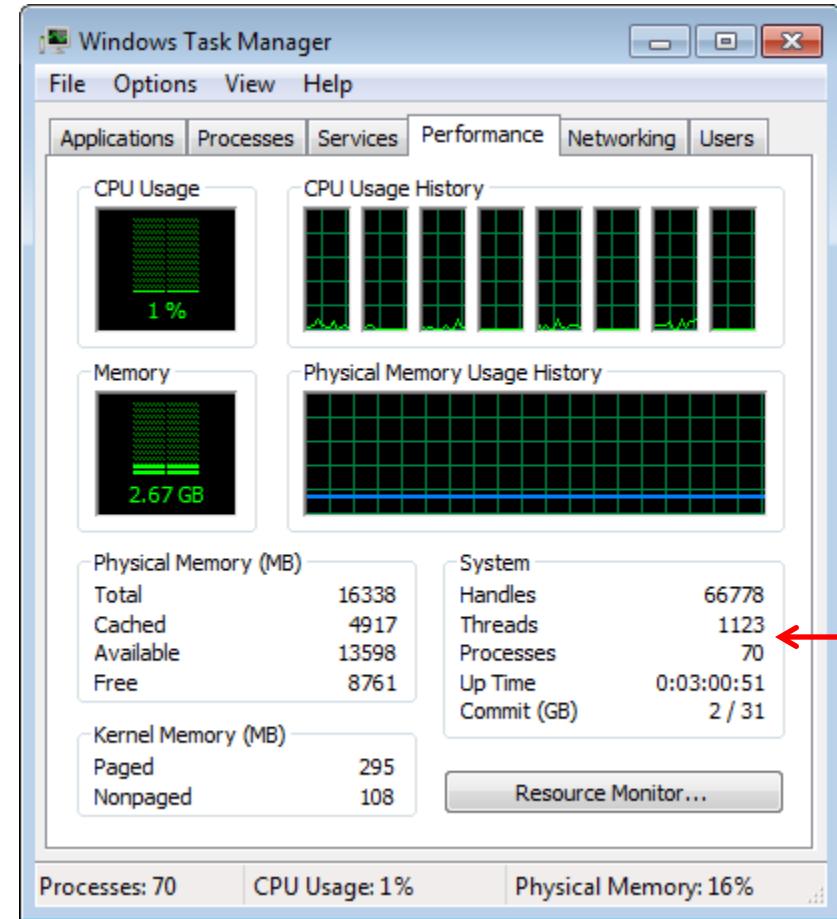
Concept could be used for shared memory parallel programming but not much used because of overhead of process creation and not being able to share data directly between processes

Threads

A separate program sequence that can be executed separately by a processor core, usually within a process.

Threads share memory space and global variables but have their own instruction pointer and stack.

An OS will manage the threads within each process.

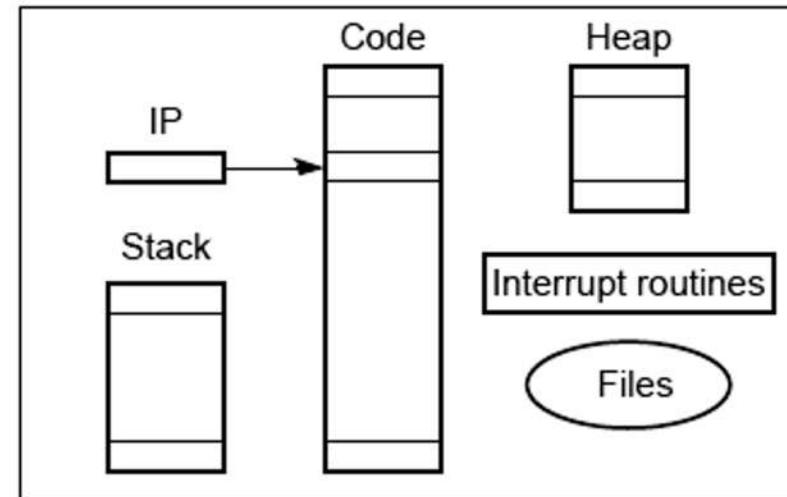


Example : destop i7-3770 quad core processor. Supports 8 threads simultaneously (hyperthreading)

Differences between a process and threads

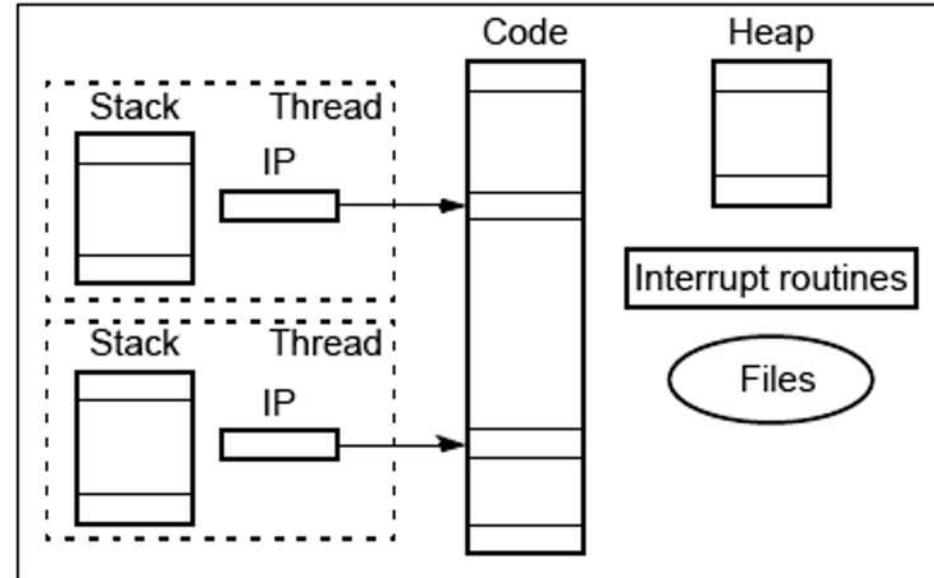
"heavyweight" process - completely separate program with its own variables, stack, and memory allocation.

(a) Process



Threads - shares the same memory space and global variables between routines.

(b) Threads



Threads in shared memory programming

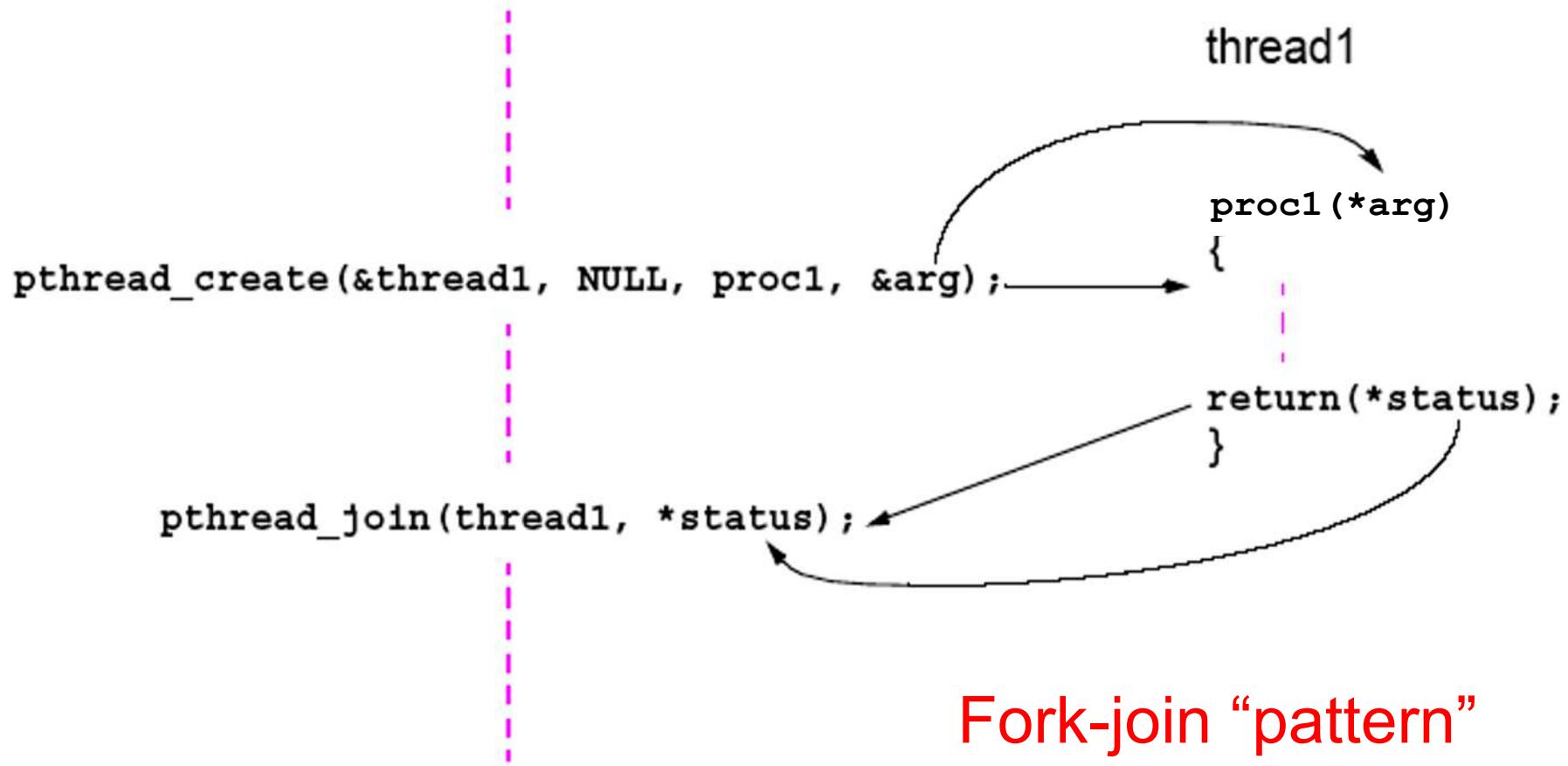
A common approach, either directly creating threads
(a low level approach) or indirectly.

Really low level -- Pthreads

IEEE Portable Operating System Interface, POSIX standard.

Executing a Pthread Thread

Main program



```
int pthread_create(pthread_t *restrict tid, const pthread_attr_t *restrict attr, void*(*start_routine)(void *), void *restrict arg);
```

Example:

```
#include <pthread.h>
pthread_t tid;
extern void *start_routine(void *arg);
void *arg;
int ret;
/* Create a thread with default behavior*/
ret = pthread_create(&tid, NULL, start_routine, arg);
```

- *start_routine* is the function with which the new thread begins execution. When *start_routine* returns, the thread exits with the exit status set to the value returned by *start_routine*.
- When **pthread_create()** is successful, the ID of the created thread is stored in the location referred to as *tid*.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
//thread function definition
void *slave(void *argument) {
    int tid = *((int *) argument);
    printf("Hello World! It's me, thread %d!\n", tid);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    int i;

    for (i = 0; i < NUM_THREADS; i++) { // create threads
        thread_args[i] = i;
        printf("In main: creating thread %d\n", i);
        if ( pthread_create(&threads[i], NULL, slave, (void *) &thread_args[i] ) != 0 )
            printf("Pthread_create fails");
    }

    for (i = 0; i < NUM_THREADS; i++) { // join threads
        if ( pthread_join(threads[i], NULL) != 0 )
            printf("Pthread_join fails");
        printf("In main: thread %d is complete\n", i);
    }
    printf("In main: All threads completed successfully\n");
    return 0;
}
```

Save your Program as **hello.c**

Compile: **gcc -o hello hello.c -lpthread**

Sample Output

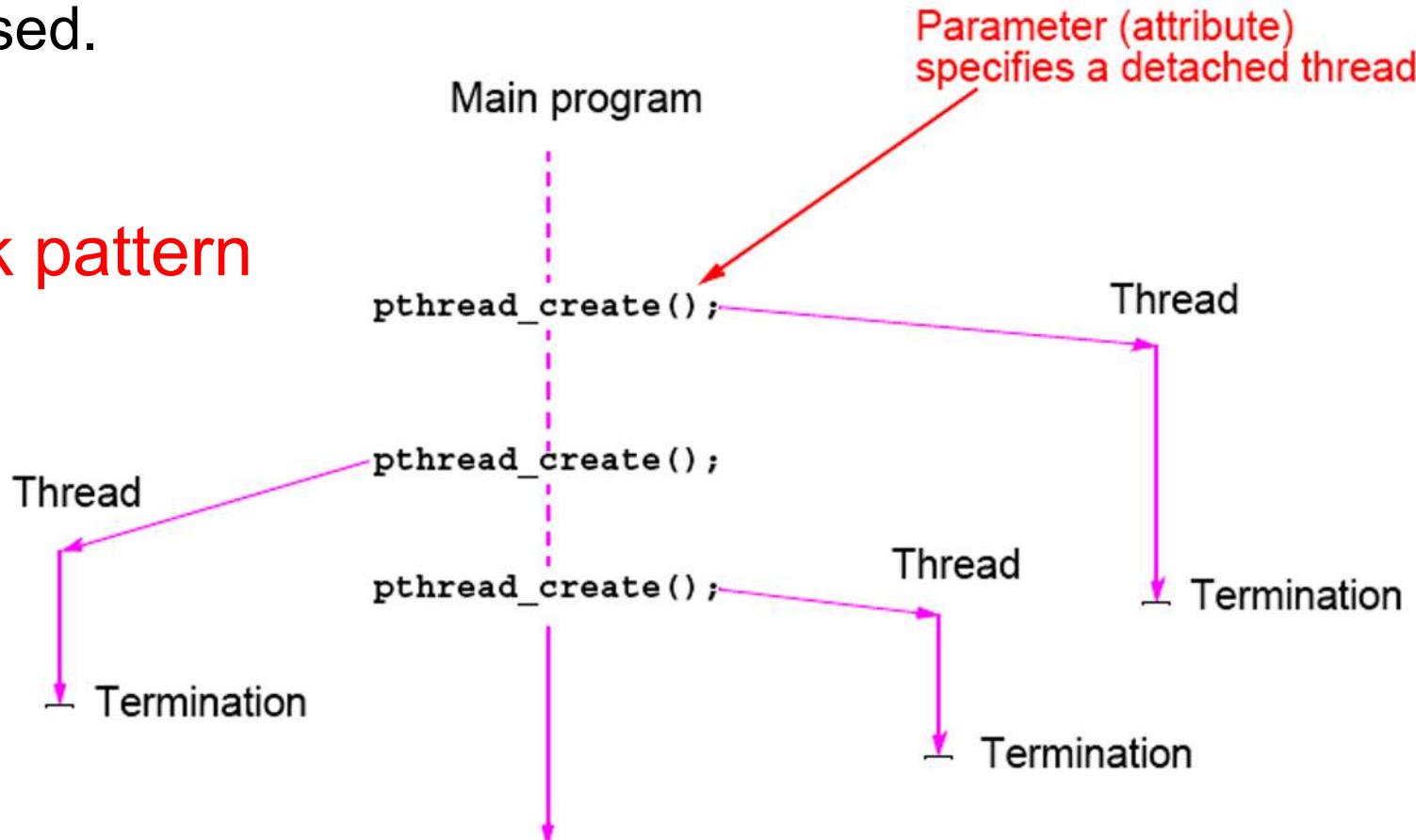
```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread 4!
Hello World! It's me, thread 0!
Hello World! It's me, thread 1!
Hello World! It's me, thread 2!
In main: thread 0 is complete
In main: thread 1 is complete
In main: thread 2 is complete
Hello World! It's me, thread 3!
In main: thread 3 is complete
In main: thread 4 is complete
In main: All threads completed successfully
```

Very simple to compile,
Just add pthread library,
but Pthreads very low
level programming

Pthreads detached threads

Threads not joined are called *detached threads*. When detached threads terminate, they are destroyed and their resources released.

Fork pattern



By default, the thread is created in joinable state unless the *attr* parameter is set to create thread in a detached state.

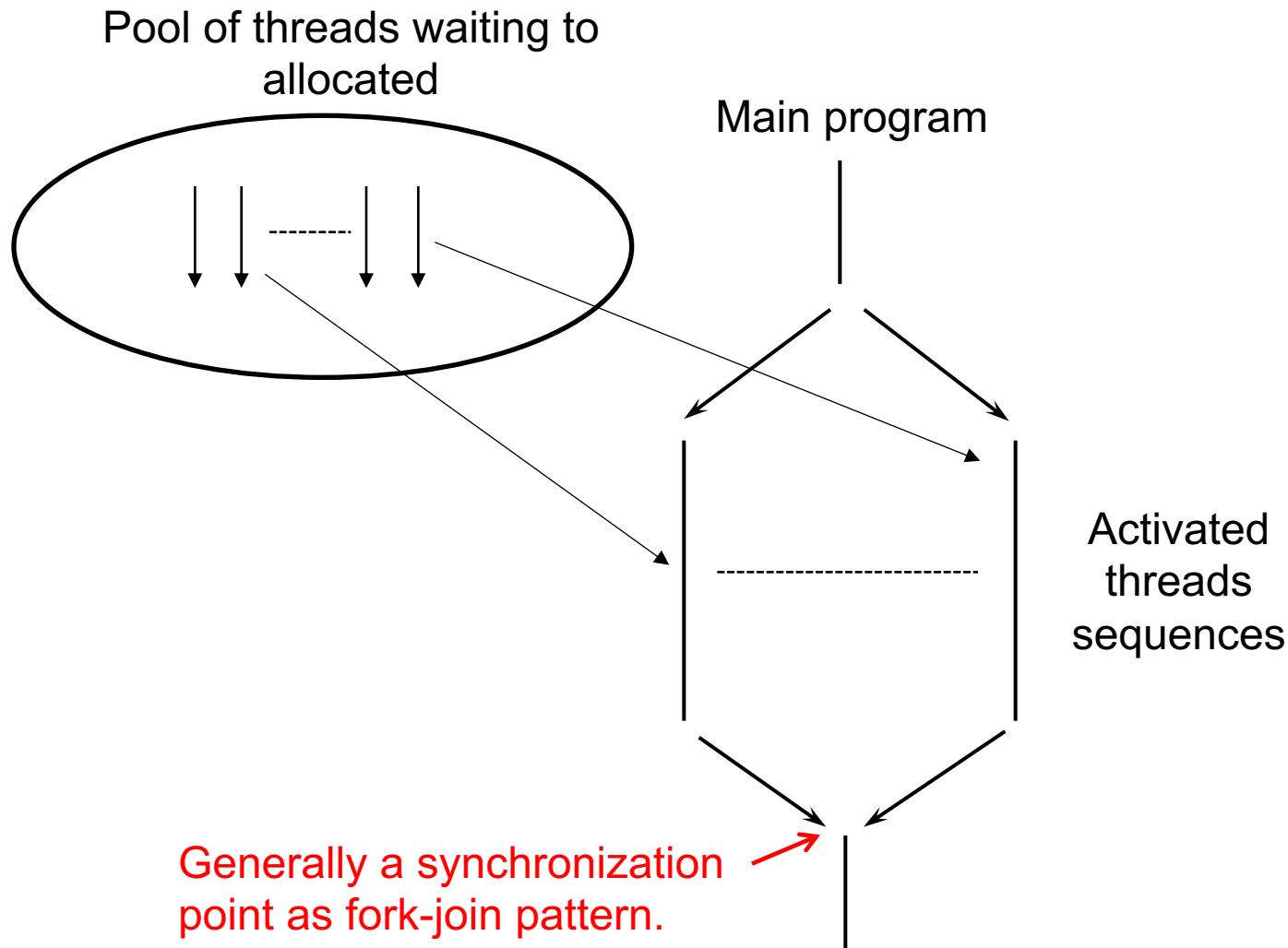
Thread pool pattern

Common to need group of threads to be used together from one execution point.

Group of threads readied to be allocated work and are brought into service.

Whether threads actually exist or are created just for then is an implementation detail.

Thread pool implies threads already created. Probably best as eliminates thread creation overhead.



Thread pool pattern or the thread team pattern is the underlying structure of OpenMP, see next.

Questions



Department of Computer Science & Mathematics

CSC 447 Parallel Computing for Multi-Core and Cluster Systems
Spring 2021

Programming with Shared Memory - 2

Issues with sharing data

Accessing Shared Data

Accessing shared data needs careful control.

Consider two threads each of which is to add one to a shared data item, **x**.

Location **x** is read, **x + 1** computed, and the result written back to the location **x**:

Instruction	Thread 1	Thread 2
x = x + 1;	read x compute x + 1 write to x	read x compute x + 1 write to x

time ↓

Different possible interleavings:

Assume initially $x = 0$

Instruction Thread 1

$x = x + 1;$ **read x**

compute $x + 1$

write to x

Thread 2

read x

compute $x + 1$

write to x

Get $x = 2$ finally.

Instruction Thread 1

$x = x + 1;$ **read x**

compute $x + 1$

write to x

Thread 2

read x

compute $x + 1$

write to x

Get $x = 1$ finally.

Shared data

- A different scheduling of the execution by various processors can lead to different results (*race condition*).
- More importantly, simultaneous access can lead to incorrect or corrupted data.
- At least one of those accesses needs to be an update or write to need special handling.

Critical Section

A mechanism for ensuring that only one process accesses a particular resource at a time (write access).

critical section – a section of code for accessing resource
Arrange that only one such critical section is executed at a time.

This mechanism is known as *mutual exclusion*.

Concept also appears in operating systems.

Locks

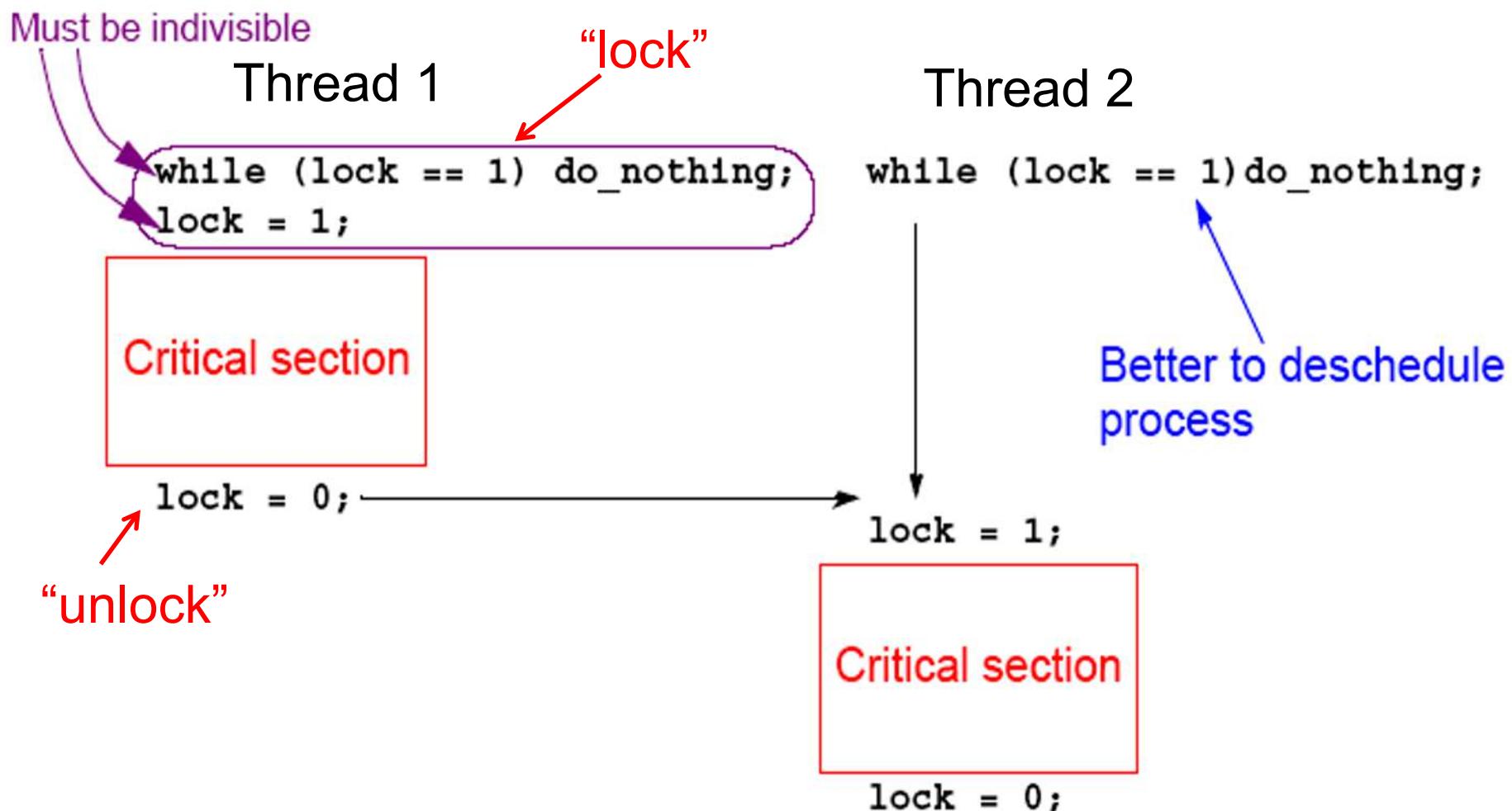
Simplest mechanism for ensuring mutual exclusion of critical sections.

A lock - a 1-bit variable that is a 1 to indicate that a process has entered the critical section and a 0 to indicate that no process is in the critical section.

Operates much like that of a door lock:

A process coming to the “door” of a critical section and finding it open may enter the critical section, locking the door behind it to prevent other processes from entering. Once the process has finished the critical section, it unlocks the door and leaves.

Control of critical sections through busy waiting



Example two threads

Thread 1

Thread 2

:

:

lock(L1)

lock(L1)

x = x + 1;

x = x + 1;

unlock(L1)

unlock(L1)

:

:

where the lock variable **L1** is associated with accessing the shared variable **x**.

Pthreads example

Locks implemented in Pthreads with *mutually exclusive lock variables*, or “mutex” variables:

```
pthread_mutex_t mutex1; // declare mutex  
...  
pthread_mutex_lock(&mutex1);  
    critical section  
pthread_mutex_unlock(&mutex1);  
...  
...
```

The diagram illustrates the use of a mutex variable. It starts with the declaration of a mutex variable `mutex1` using the `pthread_mutex_t` type. This is followed by an ellipsis `...`. Then, the mutex is locked using the `pthread_mutex_lock(&mutex1);` function. This is immediately followed by the label `critical section` in bold blue text, which is enclosed in curly braces {}, indicating a block of code that must be executed atomically. Finally, the mutex is unlocked using the `pthread_mutex_unlock(&mutex1);` function. Another ellipsis `...` follows. Red arrows from the text "Same mutex variable" point to both the declaration and the unlock function, indicating they share the same mutex object.

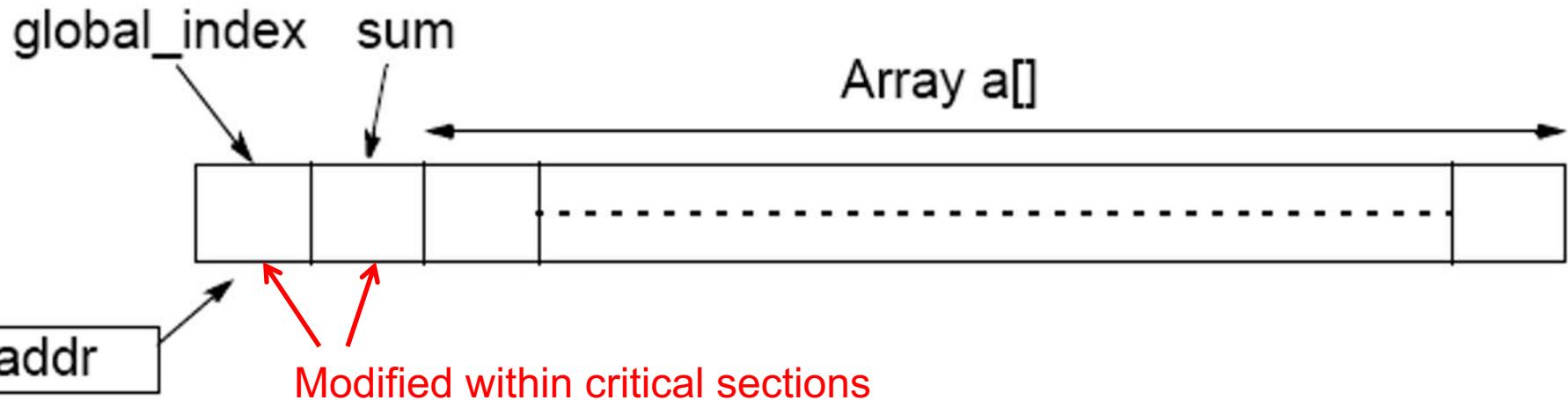
If a thread reaches mutex lock and finds it locked, it will wait for lock to open. If more than one thread waiting for lock to open, when it does open, system selects one thread to be allowed to proceed. Only thread that locks a mutex can unlock it.

Program example

To sum the elements of an array, **a[1000]**:

```
int sum, a[1000];
    sum = 0;
for (i = 0; i < 1000; i++)
    sum = sum + a[i];
```

Pthreads program example



n threads created, each taking numbers from list to add to their local partial sums. When all numbers taken, threads can add their partial sums to a shared location **sum**.

Shared location **global_index** used by each thread to select next element of `a[]`. After **global_index** read, it is incremented in preparation for next element to be read.

```
#include <stdio.h>
#include <pthread.h>
#define N 10
#define NUM_THREADS 10
int a[N]; int global_index=0; int sum=0; // shared variables
pthread_mutex_t mutex1, mutex2; // mutexes, actually could share 1

void *slave(void *argument) {
    int tid = *((int *) argument);
    printf("Hello World! It's me, thread %d!\n", tid);
    int local_index, partial_sum = 0;
    do {
        pthread_mutex_lock(&mutex1); // critical section
        local_index= global_index;
        global_index++;
        pthread_mutex_unlock(&mutex1);
        if (local_index < N) {
            printf("thread %d takes index %d\n",tid,local_index);
            partial_sum += a[local_index];
        }
    } while (local_index < N);
    pthread_mutex_lock(&mutex2);      // critical section
    sum+= partial_sum;
    pthread_mutex_unlock(&mutex2);
    return NULL;
}
```

```
int main(int argc, char *argv[]) {
    int i;
    pthread_t threads[NUM_THREADS];
    int thread_args[NUM_THREADS];
    pthread_mutex_init(&mutex1,NULL); pthread_mutex_init(&mutex2,NULL);

    for (i = 0; i < N; i++) // initialize array with some values
        a[i] = i+1;

    for (i = 0; i < NUM_THREADS; i++) // create threads
        thread_args[i] = i;
        if (pthread_create(&threads[i],NULL,slave,(void *) &thread_args[i]) != 0)
            perror("Pthread_create fails");
    }

    for (i = 0; i < NUM_THREADS; i++) // join threads
        if(pthread_join(threads[i],NULL) != 0)
            perror("Pthread_join fails");
    printf("The sum of 1 to %d is %d\n",N, sum);
    return (0);
}
```

Sample Output

Program on VM in directory
Pthreads as **critical.c**

Compile:
cc -o critical critical.c -lpthread

Hello World! It's me, thread 8!
thread 8 takes index 0
thread 8 takes index 1
thread 8 takes index 2
thread 8 takes index 3
thread 8 takes index 4
thread 8 takes index 5
thread 8 takes index 6
Hello World! It's me, thread 5!
thread 5 takes index 8
thread 5 takes index 9
Hello World! It's me, thread 7!
Hello World! It's me, thread 9!
Hello World! It's me, thread 6!
thread 8 takes index 7
Hello World! It's me, thread 4!
Hello World! It's me, thread 3!
Hello World! It's me, thread 2!
Hello World! It's me, thread 1!
Hello World! It's me, thread 0!
The sum of 1 to 10 is 55

Another run

Hello World! It's me, thread 5!

Hello World! It's me, thread 1!

thread 1 takes index 0

thread 1 takes index 1

thread 1 takes index 2

thread 1 takes index 3

thread 1 takes index 4

thread 1 takes index 5

thread 1 takes index 6

thread 1 takes index 7

thread 1 takes index 8

thread 1 takes index 9

Hello World! It's me, thread 6!

Hello World! It's me, thread 8!

Hello World! It's me, thread 7!

Hello World! It's me, thread 4!

Hello World! It's me, thread 3!

Hello World! It's me, thread 2!

Hello World! It's me, thread 0!

Hello World! It's me, thread 9!

The sum of 1 to 10 is 55

Critical Sections Serializing Code

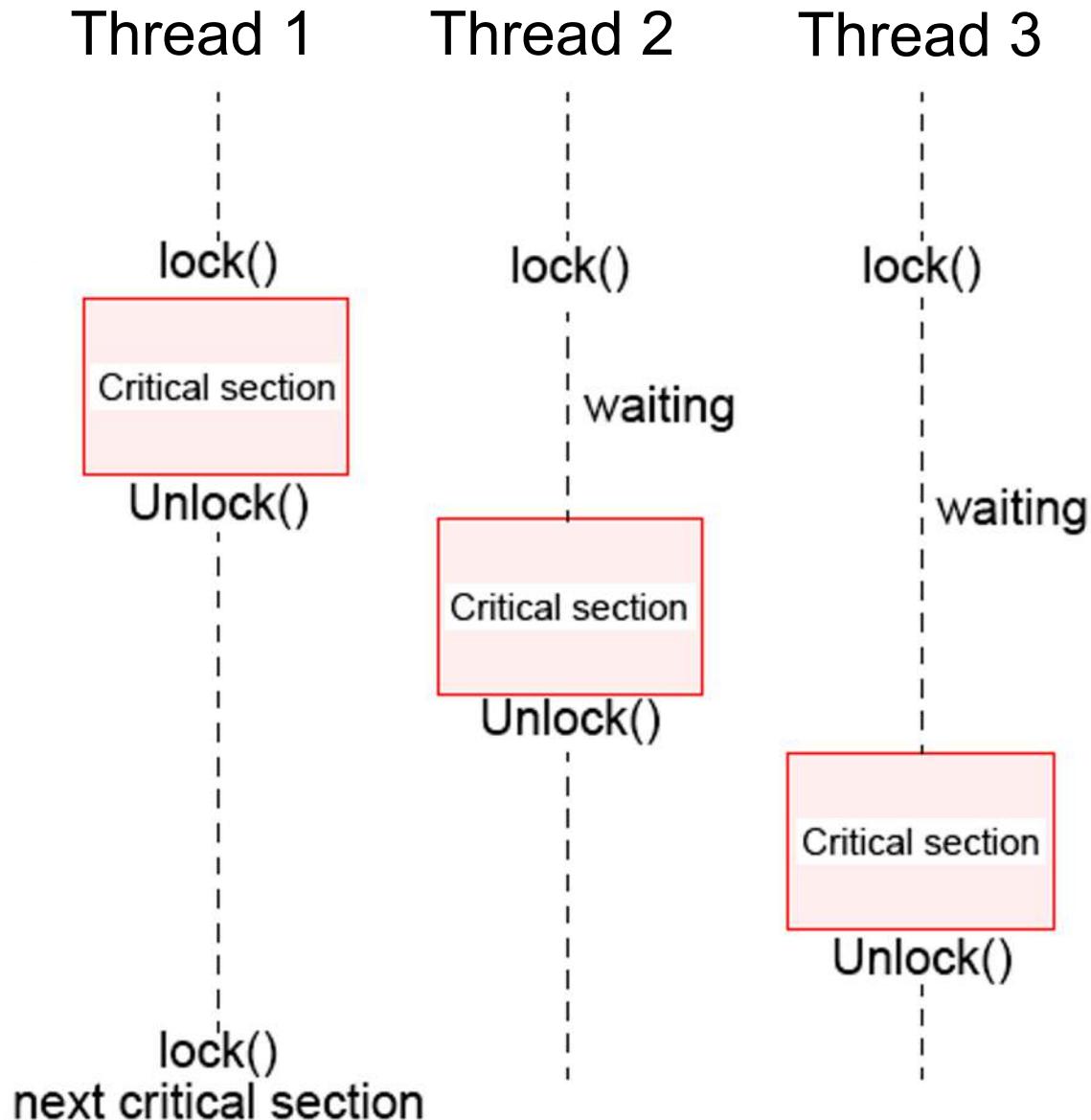
High performance programs should have as few as possible critical sections as their use can serialize the code.

Suppose, all processes happen to come to their critical section together.

They will execute their critical sections one after the other.

In that situation, the execution time becomes almost that of a single processor.

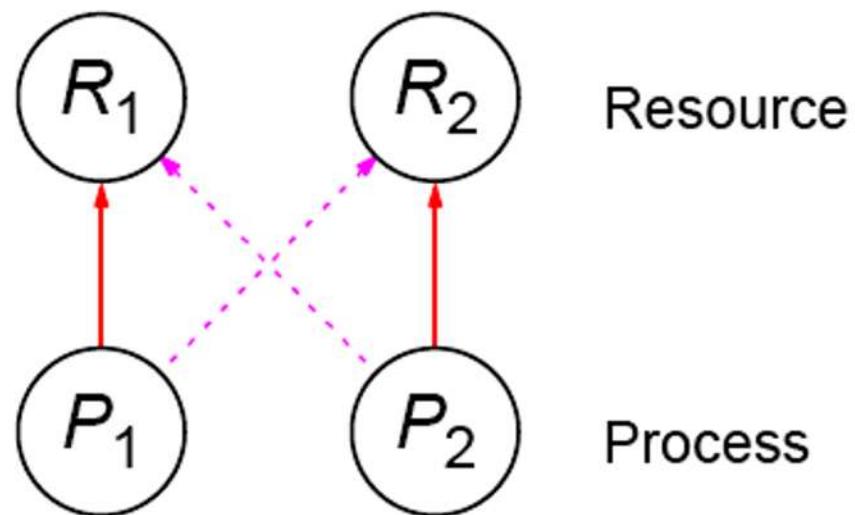
Illustration



Deadlock

Can occur with two processes/threads when one requires a resource held by the other, and this process/thread requires a resource held by the first process/thread.

Two-process deadlock



“Deadly embrace”

Deadlock Example

Two threads each wanting to access shared variables **x** and **y**.

Thread 1

:

lock(L1);

lock(L2);

temp = x + y;

unlock(L2);

unlock(L1);

:

Thread 2

:

lock(L2);

lock(L1);

temp = x + y;

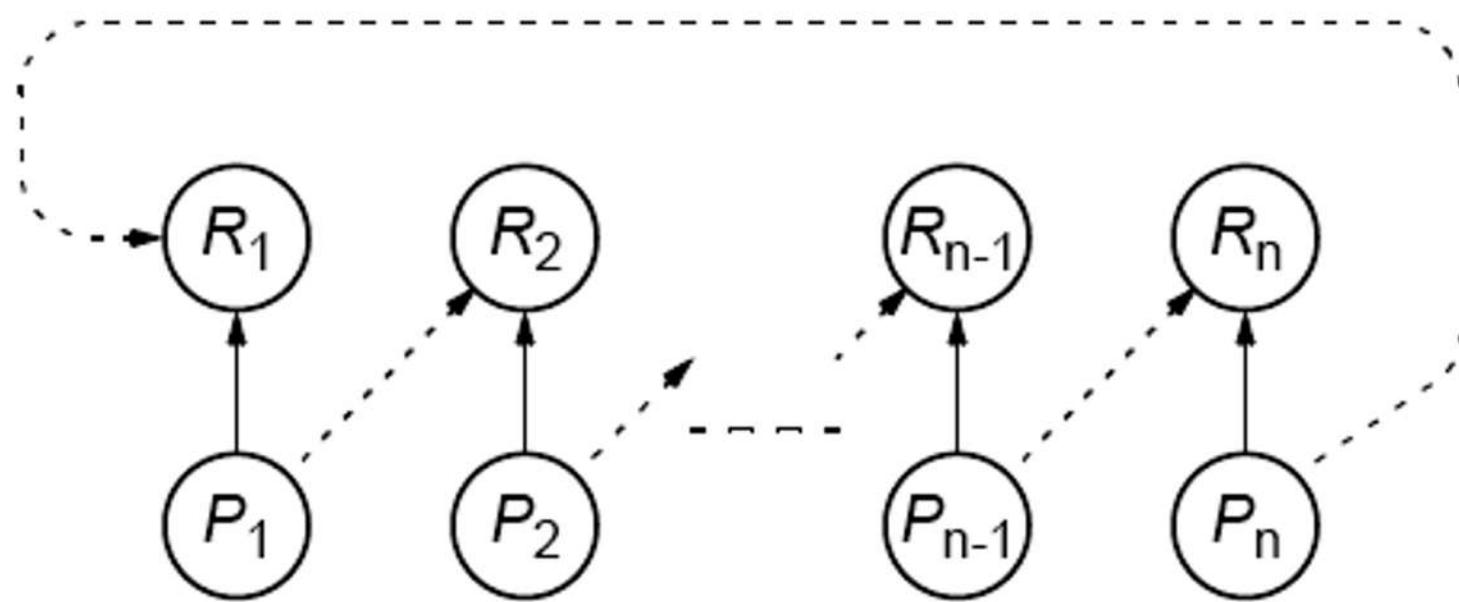
unlock(L1);

unlock(L2);

:

where the lock variable **L1** is associated with accessing the shared variable **x** and the lock variable **L2** is associated with accessing the shared variable **y**.

Deadlock can also occur in a circular fashion with several processes having a resource wanted by another.



Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals (“polled”) within a critical section.

Very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variables*.

Condition variables in Pthreads.

Pthread Condition Variables

Pthreads arrangement for signal and wait:

<pre>action() { : : pthread_mutex_lock(&mutex1); while (c <> 0) pthread_cond_wait(cond1, mutex1); ← if (c == 0) pthread_cond_signal(cond1); pthread_mutex_unlock(&mutex1); take_action(); : : }</pre>	Thread 1	<pre>counter() { : : pthread_mutex_lock(&mutex1); c--; pthread_mutex_unlock(&mutex1); : : }</pre>	Thread 2
---	-----------------	---	-----------------

Signals *not* remembered - threads must already be waiting for a signal to receive it.

Semaphores

A slightly more powerful mechanism than a binary lock. A positive integer (including zero) operated upon by two operations:

P operation on semaphore s - Waits until **s** is greater than zero and then decrements **s** by one and allows the process to continue.

V operation on semaphore s - Increments **s** by one and releases one of the waiting processes (if any). Mechanism for activating waiting processes implicit in **P** and **V** operations. Though exact algorithm not specified, algorithm expected to be fair.

P and **V** operations are performed indivisibly. Processes delayed by **P(s)** are kept in abeyance until released by a **V(s)** on the same semaphore.

Devised by Dijkstra in 1968.

Letter **P** from Dutch word *passeren*, meaning “to pass”

Letter **V** from Dutch word *vrijgeven*, meaning “to release”

Mutual exclusion of critical sections can be achieved with one semaphore having the value 0 or 1 (a binary semaphore), which acts as a lock variable, but **P** and **V** operations include a process scheduling mechanism:

Thread 1

Noncritical section

...

P(s)

Critical section

V(s)

...

Noncritical section

Thread 2

Noncritical section

...

P(s)

Critical section

V(s)

...

Noncritical section

Thread 3

Noncritical section

...

P(s)

Critical section

V(s)

...

Noncritical section

General semaphore (or counting semaphore)

Can take on positive values other than zero and one.

Provide, for example, a means of recording number of “resource units” available or used. Can solve producer/consumer problems - more on that in operating system courses.

Semaphore routines exist for UNIX processes.

Pthread Semaphore

Does not exist in Pthreads as such, though they can be written.

Do exist in real-time extension to Pthreads.

Monitor

Suite of procedures that provides only way to access shared resource. ***Only one process can use a monitor procedure at any instant.***

Could be implemented using a semaphore or lock to protect entry, i.e.:

```
monitor_proc1() {  
    lock(x);
```

monitor body

```
    unlock(x);  
    return;  
}
```

Questions

GPU Programming

Lecture 1.1 – Introduction

Introduction to Heterogeneous Parallel Computing

Objectives

- To learn the major differences between latency devices (CPU cores) and throughput devices (GPU cores)
- To understand why winning applications increasingly use both types of devices

Latency vs. Throughput

- Latency: The time required to perform some action.
 - measured in units of time.
- Throughput: The number of actions executed in a unit of time.
 - Measured in units of what is produced per unit of time.
- E.g. An assembly line manufactures GPUs. It takes 6 hours to manufacture a GPU but the assembly line can manufacture 100 GPUs per day.

Interesting Article:

D. Patterson, “Latency lags bandwidth”, IEEE International Conference on Computer Design: VLSI in Computers and Processors, ICCD, San Jose, CA, USA, 2005.

Latency vs. Throughput

- A real life example: Need to go on a trip of 400Km.

Car: 2 people, 200 km/hour

Bus: 40 people, 50 km/hour

Car:

Latency=?? Hours

Throughput=?? People/Hour

Bus:

Latency=?? Hours

Throughput=?? People/Hour

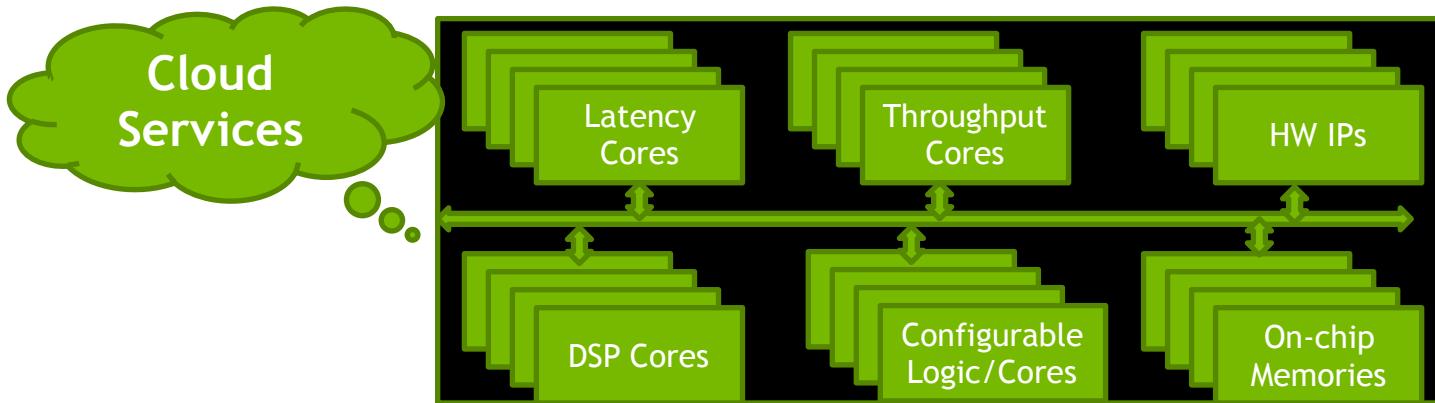
← →

Travel Distance: 400 Km

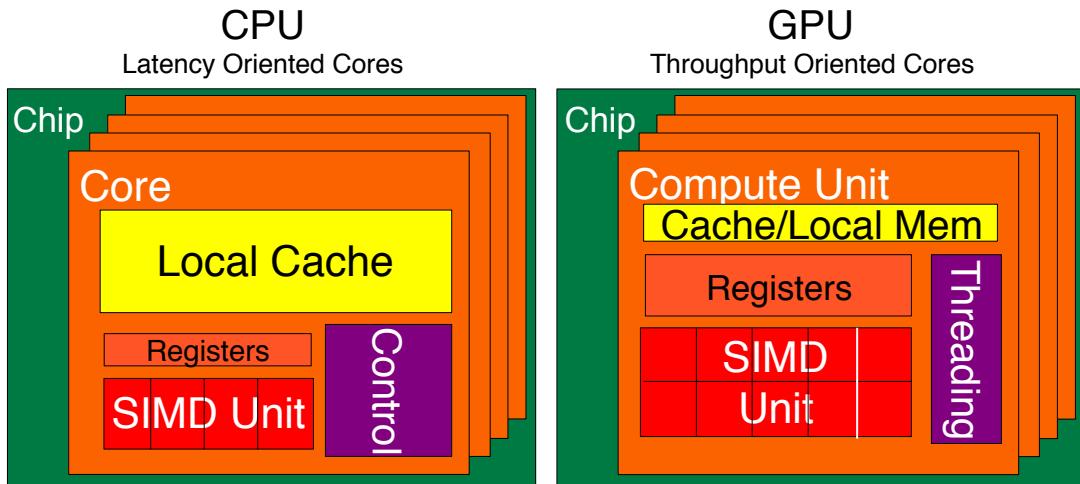


Heterogeneous Parallel Computing

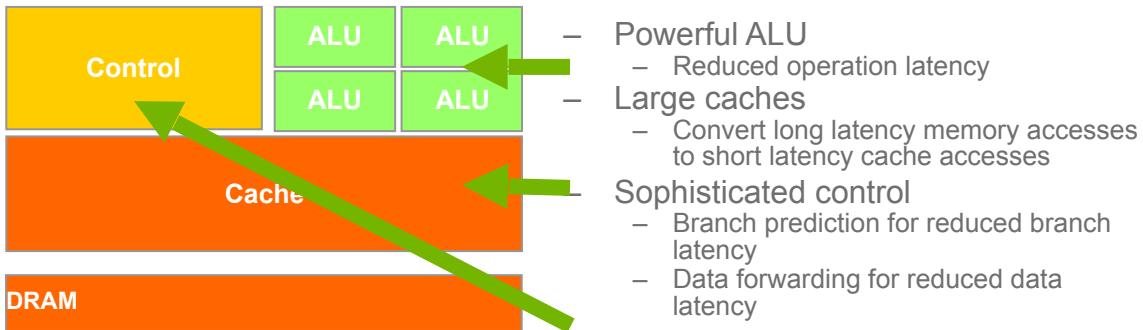
- Use the best match for the job (heterogeneity in mobile SOC)



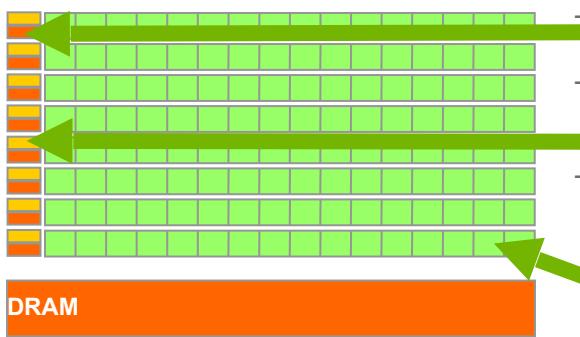
CPU and GPU are designed very differently



CPUs: Latency Oriented Design



GPUs: Throughput Oriented Design



- Small caches
 - To boost memory throughput
 - Simple control
 - No branch prediction
 - No data forwarding
 - Energy efficient ALUs
 - Many, long latency but heavily pipelined for high throughput
- Require massive number of threads to tolerate latencies
- Threading logic
 - Thread state

Winning Applications Use Both CPU and GPU

- CPUs for sequential parts where latency matters
 - CPUs can be 10X+ faster than GPUs for sequential code
- GPUs for parallel parts where throughput wins
 - GPUs can be 10X+ faster than CPUs for parallel code

Heterogeneous Parallel Computing in Many Disciplines

Financial Analysis

Scientific Simulation

Engineering Simulation

Data Intensive Analytics

Medical Imaging

Digital Audio Processing

Digital Video Processing

Computer Vision

Biomedical Informatics

Electronic Design Automation

Statistical Modeling

Numerical Methods

Ray Tracing Rendering

Interactive Physics

GPU Programming

Lecture 2.1 - Introduction to CUDA C

CUDA C vs. Thrust vs. CUDA Libraries

Objective

- To learn the main venues and developer resources for GPU computing
- Where CUDA C fits in the big picture

3 Ways to Accelerate Applications

Applications

Libraries

Compiler
Directives

Programming
Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **“Drop-in”:** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications

GPU Accelerated Libraries

Linear Algebra
FFT, BLAS,
SPARSE, Matrix



cULA tools



C U S P

Numerical & Math
RAND, Statistics



ArrayFire



Data Struct. & AI
Sort, Scan, Zero Sum



Visual Processing
Image & Video



NVIDIA
Video
Encode



Vector Addition in Thrust

```
thrust::device_vector<float> deviceInput1(inputLength);  
thrust::device_vector<float> deviceInput2(inputLength);  
thrust::device_vector<float> deviceOutput(inputLength);
```

```
thrust::copy(hostInput1, hostInput1 + inputLength,  
deviceInput1.begin());  
thrust::copy(hostInput2, hostInput2 + inputLength,  
deviceInput2.begin());
```

```
thrust::transform(deviceInput1.begin(), deviceInput1.end(),  
deviceInput2.begin(), deviceOutput.begin(),  
thrust::plus<float>());
```

Compiler Directives: Easy, Portable Acceleration

- **Ease of use:** Compiler takes care of details of parallelism management and data movement
- **Portable:** The code is generic, not specific to any type of hardware and can be deployed into multiple languages
- **Uncertain:** Performance of code can vary across compiler versions

OpenACC

- Compiler directives for C, C++, and FORTRAN

```
#pragma acc parallel loop
copyin(input1[0:inputLength],input2[0:inputLength]),
copyout(output[0:inputLength])
for(i = 0; i < inputLength; ++i) {
    output[i] = input1[i] + input2[i];
}
```

Programming Languages: Most Performance and Flexible Acceleration

- **Performance:** Programmer has best control of parallelism and data movement
- **Flexible:** The computation does not need to fit into a limited set of library patterns or directive types
- **Verbose:** The programmer often needs to express more details

GPU Programming Languages

Numerical analytics ➤

MATLAB Mathematica, LabVIEW

Fortran ➤

CUDA Fortran

C ➤

CUDA C

C++ ➤

CUDA C++

Python ➤

PyCUDA, Copperhead, Numba

F# ➤

Alea.cuBase

CUDA - C

Applications

Libraries

Compiler
Directives

Programming
Languages

Easy to use
Most Performance

Easy to use
Portable code

Most Performance
Most Flexibility

GPU Programming

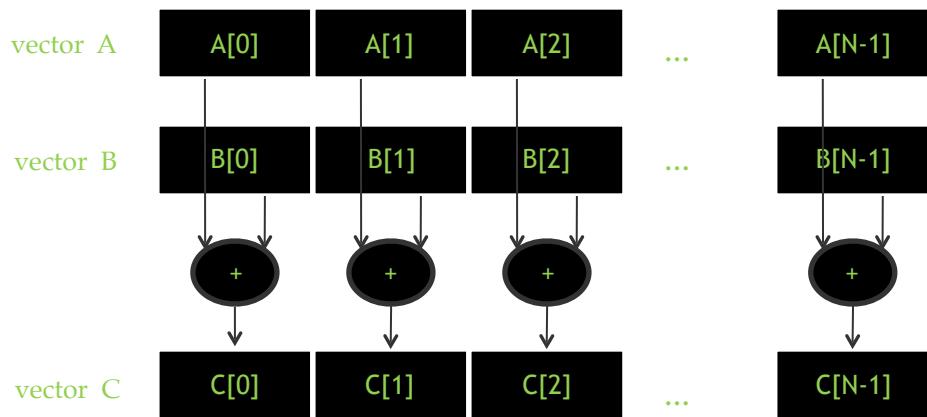
Lecture 2 - Introduction to CUDA C

Memory Allocation and Data Movement API Functions

Objective

- To learn the basic API functions in CUDA host code
- Device Memory Allocation
- Host-Device Data Transfer

Data Parallelism - Vector Addition Example



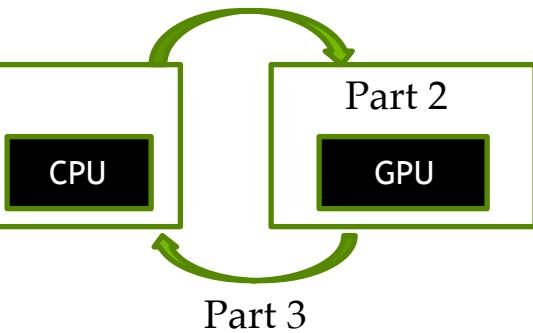
Vector Addition – Traditional C Code

```
// Compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int i;
    for (i = 0; i<n; i++) h_C[i] = h_A[i] + h_B[i];
}

int main()
{
    // Memory allocation for h_A, h_B, and h_C
    // I/O to read h_A and h_B, N elements
    ...
    vecAdd(h_A, h_B, h_C, N);
}
```

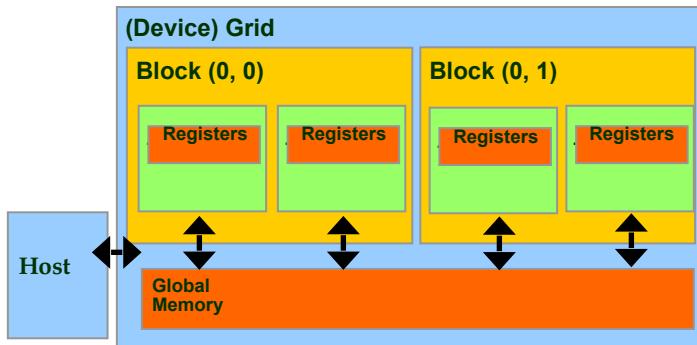
Heterogeneous Computing vecAdd CUDA Host Code

Part 1



```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n* sizeof(float);
    float *d_A, *d_B, *d_C;
    // Part 1
    // Allocate device memory for A, B, and C
    // copy A and B to device memory
    // Part 2
    // Kernel launch code – the device performs the actual vector addition
    // Part 3
    // copy C from the device memory
    // Free device vectors
}
```

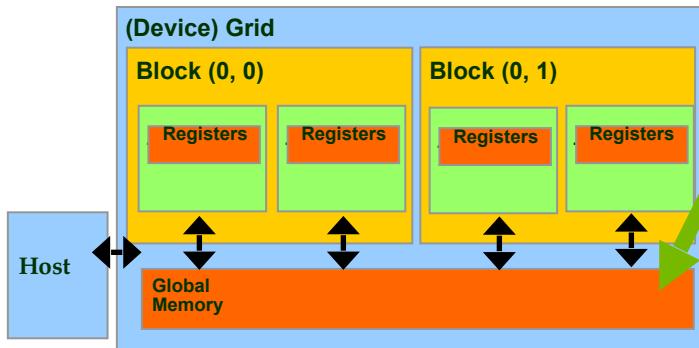
Partial Overview of CUDA Memories



- Device code can:
 - R/W per-thread **registers**
 - R/W all-shared **global memory**
- Host code can
 - Transfer data to/from per grid **global memory**

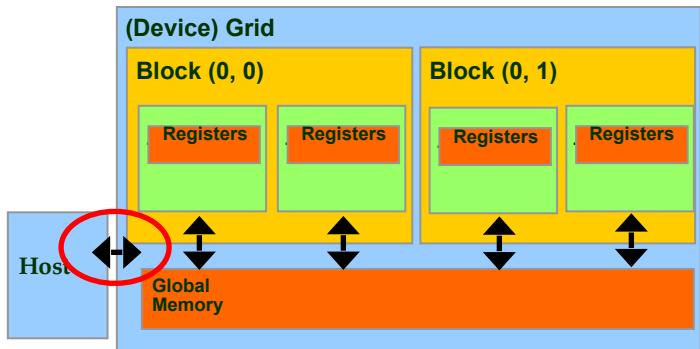
We will cover more memory types and more sophisticated memory models later.

CUDA Device Memory Management API functions



- `cudaMalloc()`
- Allocates an object in the device global memory
- Two parameters
- Address of a pointer to the allocated object
- Size of allocated object in terms of bytes
- `cudaFree()`
- Frees object from device global memory
- One parameter
- Pointer to freed object

Host-Device Data Transfer API functions



- `cudaMemcpy()`
- memory data transfer
- Requires four parameters
- Pointer to destination
- Pointer to source
- Number of bytes copied
- Type/Direction of transfer
- Transfer to device is asynchronous

Vector Addition Host Code

```
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
    int size = n * sizeof(float); float *d_A, *d_B, *d_C;
    cudaMalloc((void **) &d_A, size);
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_B, size);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
    cudaMalloc((void **) &d_C, size);

    // Kernel invocation code – to be shown later

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaFree(d_A); cudaFree(d_B); cudaFree (d_C);
}
```

In Practice, Check for API Errors in Host Code

```
cudaError_t err = cudaMalloc((void **) &d_A, size);

if (err != cudaSuccess) {
    printf("%s in %s at line %d\n", cudaGetErrorString(err),
    __FILE__,
    __LINE__);
    exit(EXIT_FAILURE);
}
```

GPU Programming

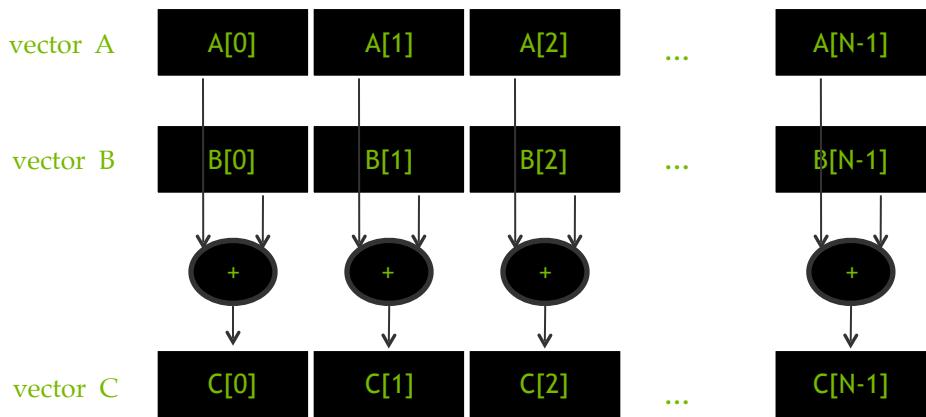
Lecture 3 – Introduction to CUDA C

Threads and Kernel Functions

Objective

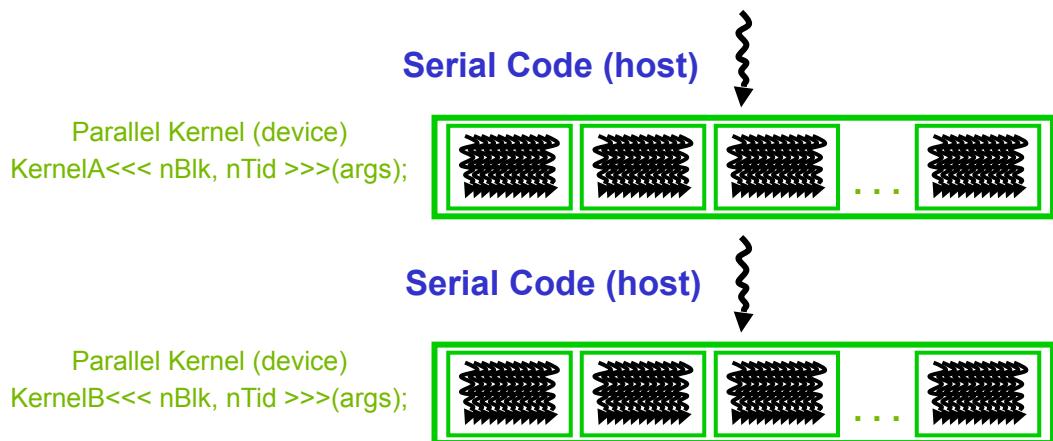
- To learn about CUDA threads, the main mechanism for exploiting of data parallelism
- Hierarchical thread organization
- Launching parallel execution
- Thread index to data index mapping

Data Parallelism - Vector Addition Example

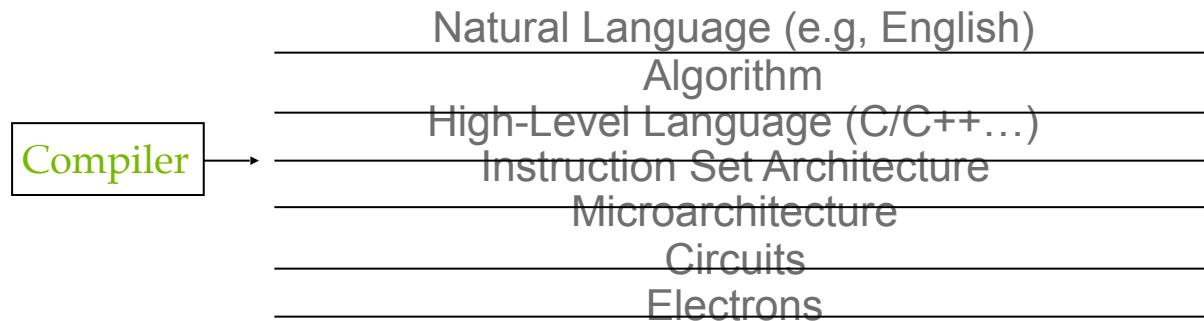


CUDA Execution Model

- Heterogeneous host (CPU) + device (GPU) application C program
 - Serial parts in **host** C code
 - Parallel parts in **device** SPMD kernel code



From Natural Language to Electrons



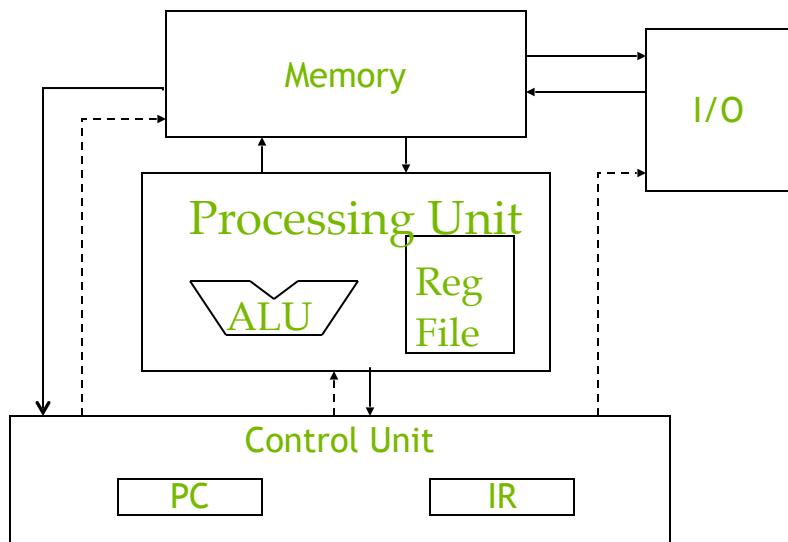
©Yale Patt and Sanjay Patel, *From bits and bytes to gates and beyond*

A program at the ISA level

- A program is a set of instructions stored in memory that can be read, interpreted, and executed by the hardware.
- Both CPUs and GPUs are designed based on (different) instruction sets
- Program instructions operate on data stored in memory and/or registers.

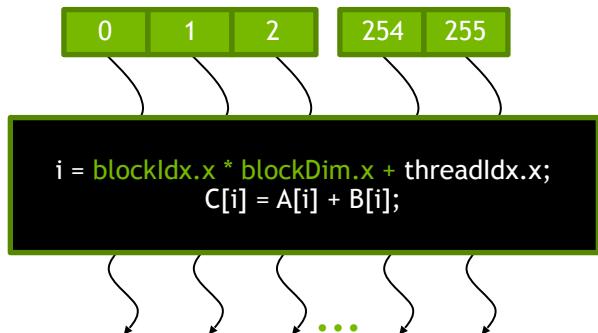
A Thread as a Von-Neumann Processor

A thread is a “virtualized” or “abstracted”
Von-Neumann Processor

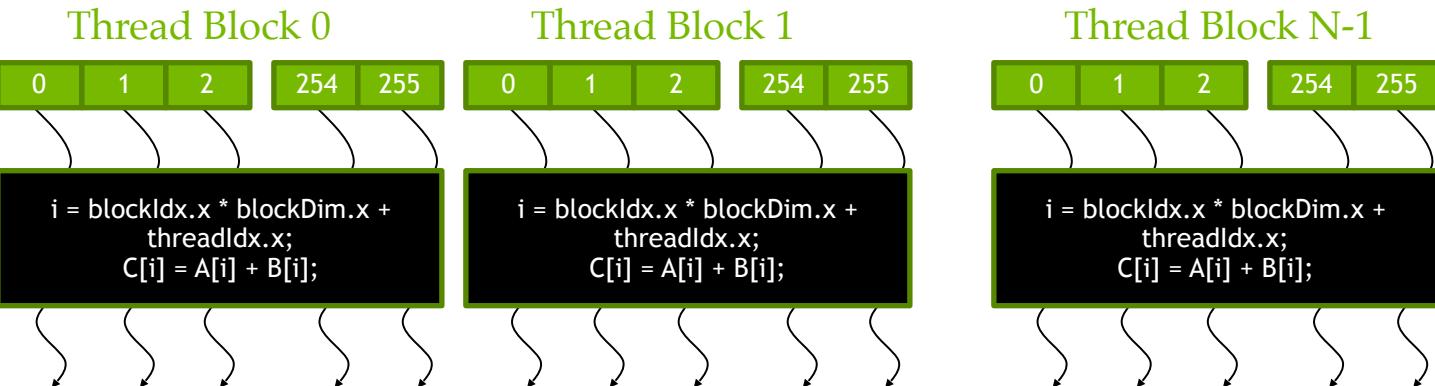


Arrays of Parallel Threads

- A CUDA kernel is executed by a **grid** (array) of threads
- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that it uses to compute memory addresses and make control decisions



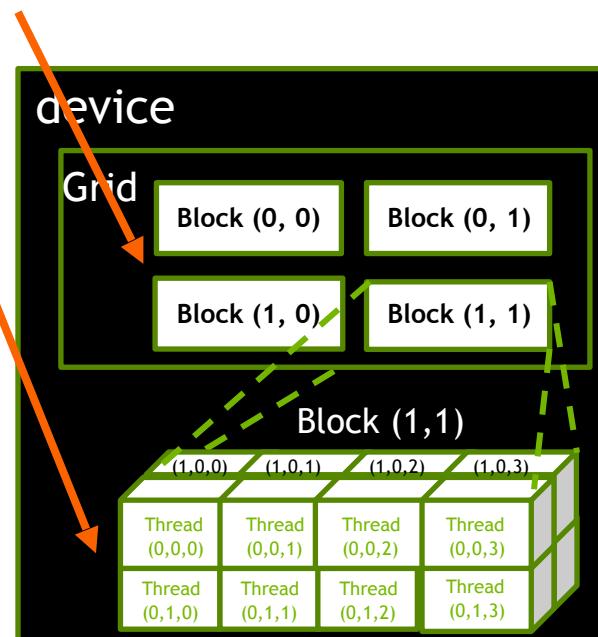
Thread Blocks: Scalable Cooperation



- Divide thread array into multiple blocks
 - Threads within a block cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
 - Threads in different blocks do not interact

blockIdx and threadIdx

- Each thread uses indices to decide what data to work on
 - blockIdx: 1D, 2D, or 3D (CUDA 4.0)
 - threadIdx: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes
 - ...



GPU Programming

Lecture 4 - CUDA Parallelism Model

Kernel-Based SPMD Parallel Programming

Objective

- To learn the basic concepts involved in a simple CUDA kernel function
- Declaration
- Built-in variables
- Thread index to data index mapping

Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

global
void vecAddKernel(float* A, float* B, float* C, int
n)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i < n) C[i] = A[i] + B[i];
}
```

Example: Vector Addition Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    // d_A, d_B, d_C allocations and copies omitted
    // Run ceil(n/256.0) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0),256>>>(d_A, d_B, d_C,
    n);
}
```



The ceiling function makes sure that there are enough threads to cover all elements.

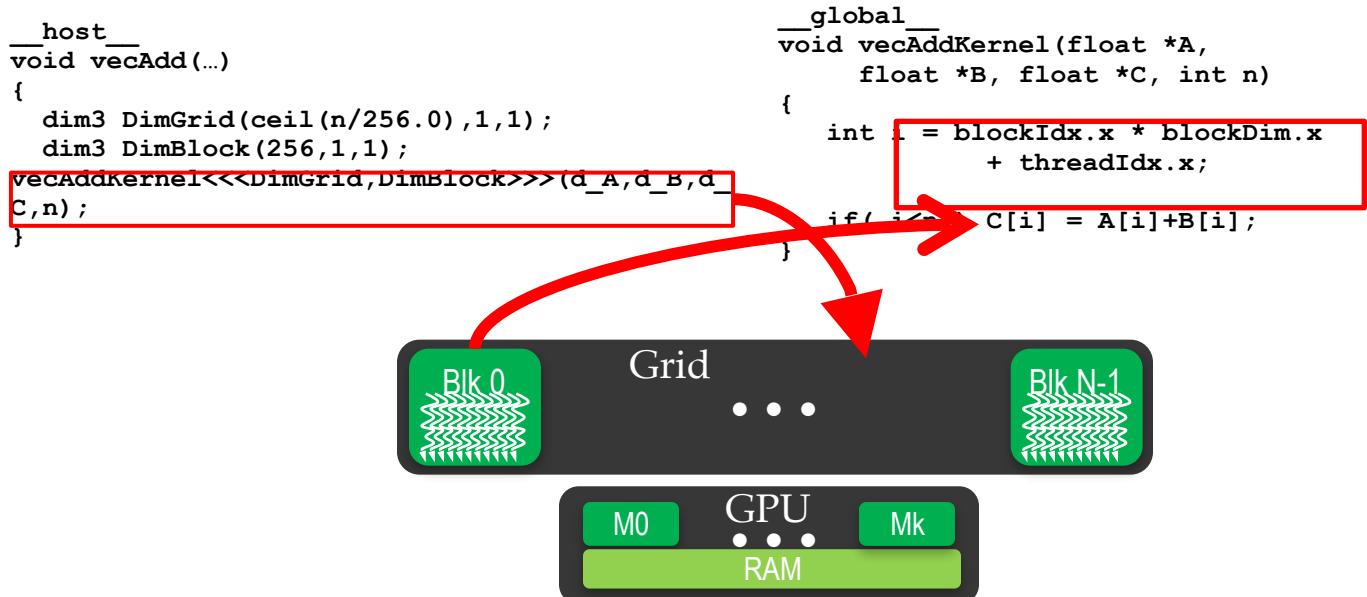
More on Kernel Launch (Host Code)

Host Code

```
void vecAdd(float* h_A, float* h_B, float* h_C, int n)
{
    dim3 DimGrid((n-1)/256 + 1, 1, 1);
    dim3 DimBlock(256, 1, 1);
    vecAddKernel<<<DimGrid,DimBlock>>>(d_A, d_B, d_C,
n);
}
```

This is an equivalent way to express the ceiling function.

Kernel execution in a nutshell



More on CUDA Function Declarations

	Executed on the:	Only callable from the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	host	host

- `__global__` defines a kernel function
 - Each “__” consists of two underscore characters
 - A kernel function must return `void`
- `__device__` and `__host__` can be used together
- `__host__` is optional if used alone

CSC 447 Parallel Programming

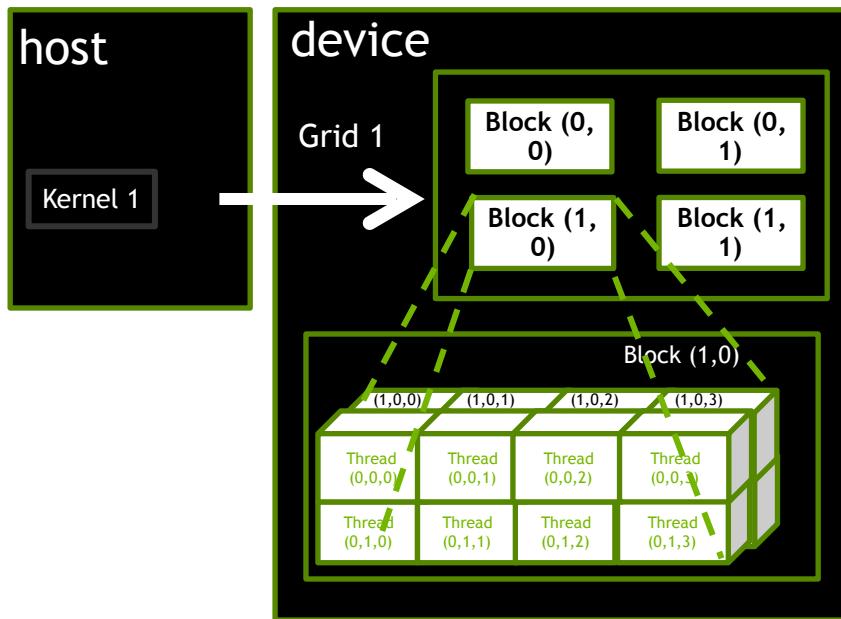
Lecture 6 – CUDA Parallelism Model

Multidimensional Kernel Configuration

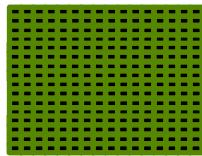
Objective

- To understand multidimensional Grids
- Multi-dimensional block and thread indices
- Mapping block/thread indices to data indices

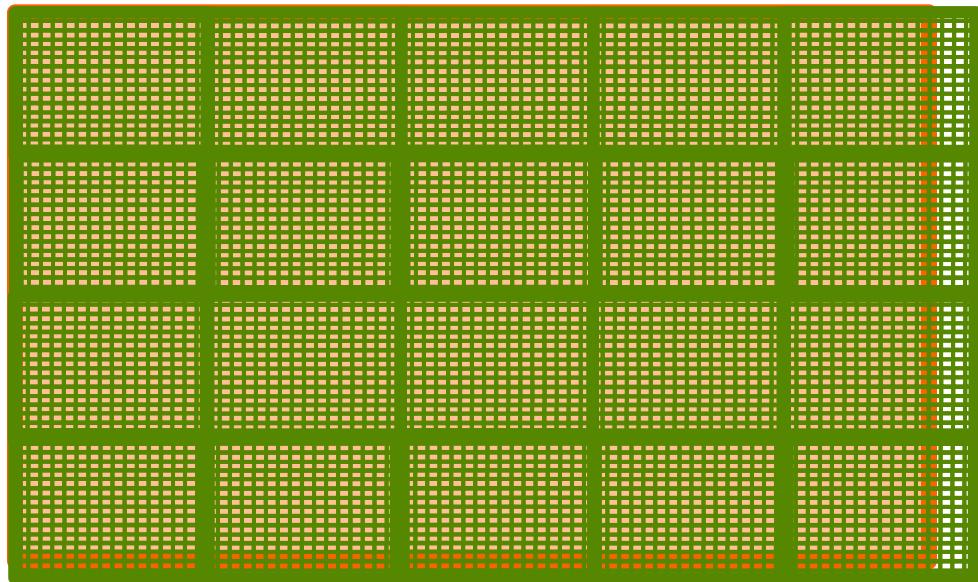
A Multi-Dimensional Grid Example



Processing a Picture with a 2D Grid



1616 blocks



6276 picture

Row-Major Layout in C/C++

M
↓

$$\text{Row} * \text{Width} + \text{Col} = 2 * 4 + 1 = 9$$

M ₀	M ₁	M ₂	M ₃	M ₄	M ₅	M ₆	M ₇	M ₈	M ₉	M ₁₀	M ₁₁	M ₁₂	M ₁₃	M ₁₄	M ₁₅
----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	----------------	-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

M
↓



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}	M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}	M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}	M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}
------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------	------------------



M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}

Source Code of a PictureKernel

```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
                               int height, int width)
{
    // Calculate the row # of the d_Pin and d_Pout element
    int Row = blockIdx.y*blockDim.y + threadIdx.y;

    // Calculate the column # of the d_Pin and d_Pout element
    int Col = blockIdx.x*blockDim.x + threadIdx.x;

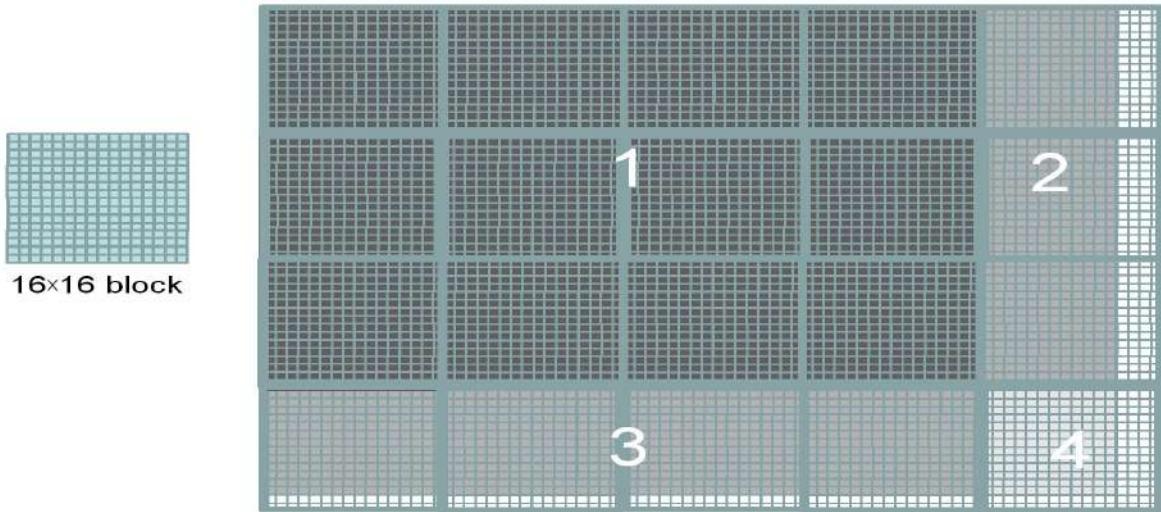
    // each thread computes one element of d_Pout if in range
    if ((Row < height) && (Col < width)) {
        d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
    }
}
```

Scale every pixel value by 2.0

Host Code for Launching PictureKernel

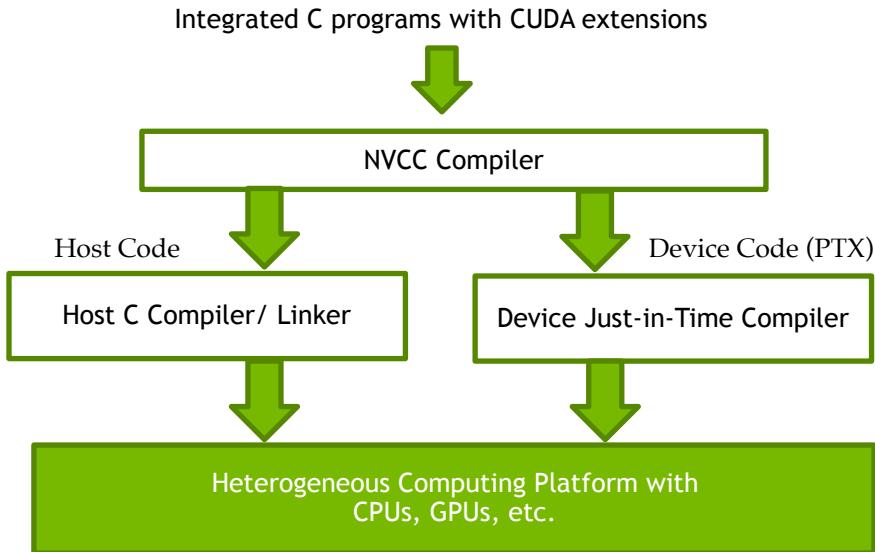
```
// assume that the picture is mn,  
// m pixels in y dimension and n pixels in x dimension  
// input d_Pin has been allocated on and copied to device  
// output d_Pout has been allocated on device  
...  
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);  
dim3 DimBlock(16, 16, 1);  
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);  
...
```

Covering a 6276 Picture with 1616 Blocks



Not all threads in a Block will follow the same control flow path.

Compiling A CUDA Program



CSC 447: Parallel Programming

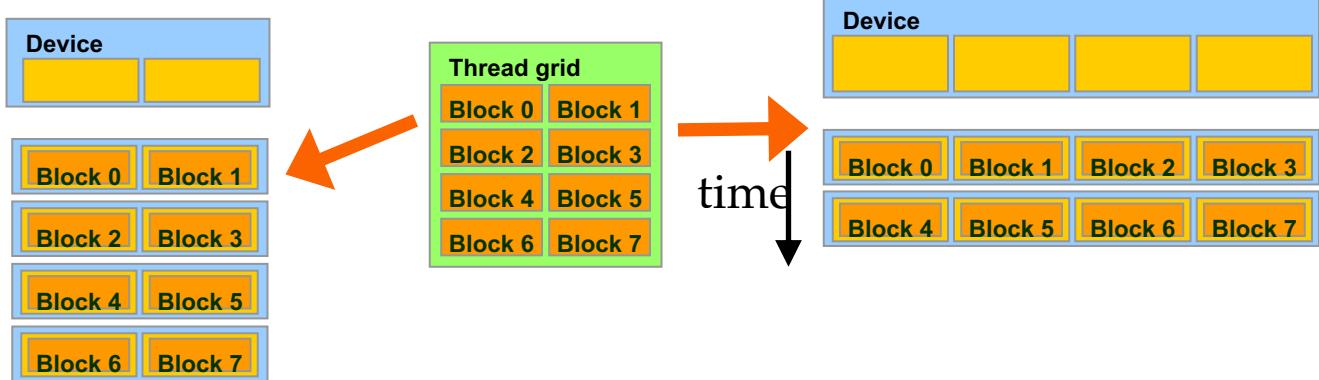
CUDA Parallelism Model

Thread Scheduling

Objective

- To learn how a CUDA kernel utilizes hardware execution resources
 - Assigning thread blocks to execution resources
 - Capacity constraints of execution resources
 - Zero-overhead thread scheduling

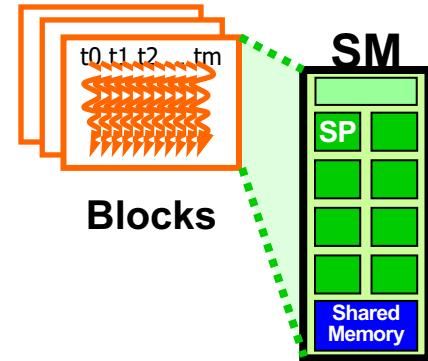
Transparent Scalability



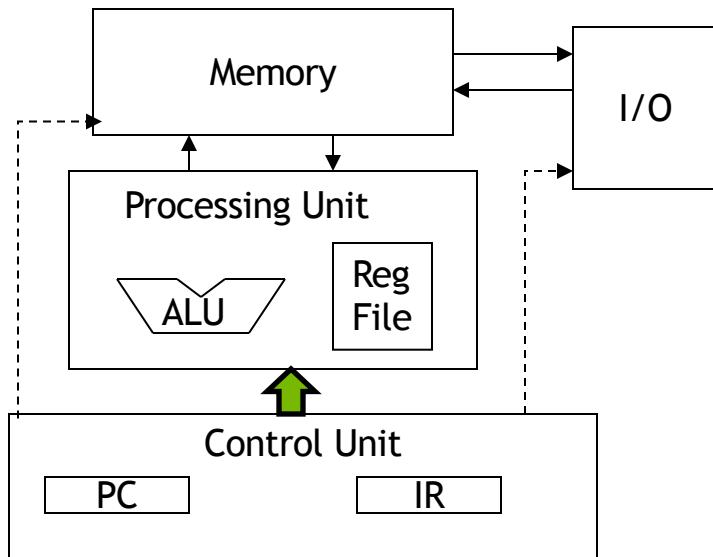
- Each block can execute in any order relative to others.
- Hardware is free to assign blocks to any processor at any time
 - A kernel scales to any number of parallel processors

Example: Executing Thread Blocks

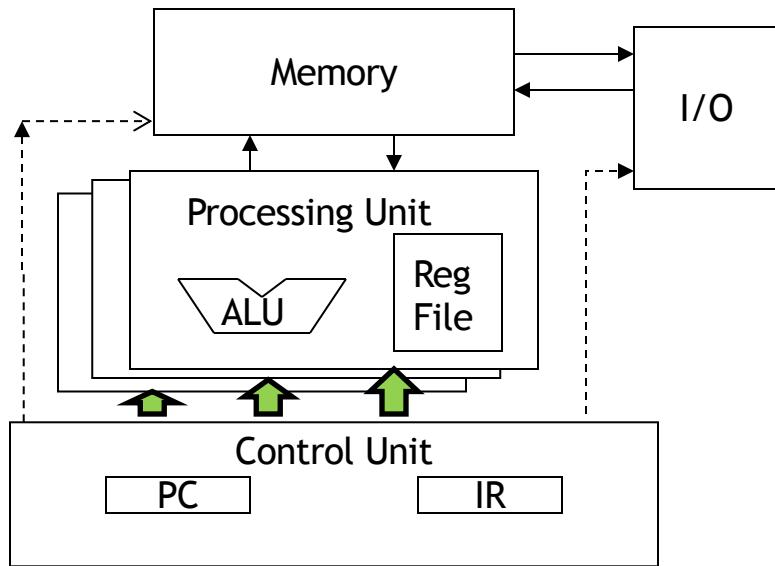
- Threads are assigned to **Streaming Multiprocessors (SM)** in block granularity
 - Up to **8** blocks to each SM as resource allows
 - Fermi SM can take up to **1536** threads
 - Could be $256 \text{ (threads/block)} * 6 \text{ blocks}$
 - Or $512 \text{ (threads/block)} * 3 \text{ blocks}$, etc.
- SM maintains thread/block idx #s
- SM manages/schedules thread execution



The Von-Neumann Model



The Von-Neumann Model with SIMD units



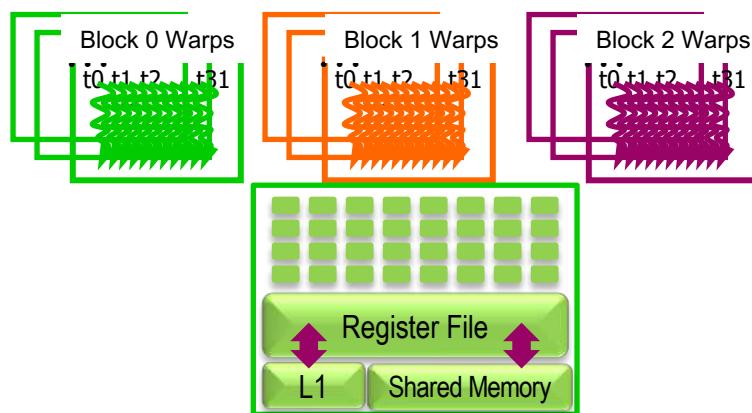
Single Instruction Multiple Data
(SIMD)

Warps as Scheduling Units

- Each Block is executed as 32-thread Warps
 - An implementation decision, not part of the CUDA programming model
 - Warps are scheduling units in SM
 - Threads in a warp execute in SIMD
 - Future GPUs may have different number of threads in each warp

Warp Example

- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
 - Each Block is divided into $256/32 = 8$ Warps
 - There are $8 * 3 = 24$ Warps



Example: Thread Scheduling (Cont.)

- SM implements zero-overhead warp scheduling
 - Warps whose next instruction has its operands ready for consumption are eligible for execution
 - Eligible Warps are selected for execution based on a prioritized scheduling policy
 - All threads in a warp execute the same instruction when selected

Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks for Fermi?
 - For 8X8, we have 64 threads per Block. Since each SM can take up to 1536 threads, which translates to 24 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
 - For 16X16, we have 256 threads per Block. Since each SM can take up to 1536 threads, it can take up to 6 Blocks and achieve full capacity unless other resource considerations overrule.
 - For 32X32, we would have 1024 threads per Block. Only one block can fit into an SM for Fermi. Using only 2/3 of the thread capacity of an SM.

CSC 447: Parallel Programming

Lecture 8– Memory and Data Locality

CUDA Memories

Objective

- To learn to effectively use the CUDA memory types in a parallel program
- Importance of memory access efficiency
- Registers, shared memory, global memory
- Scope and lifetime

Review: Image Blur Kernel.

```
// Get the average of the surrounding 2xBLUR_SIZE x 2xBLUR_SIZE box
for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
    for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

        int curRow = Row + blurRow;
        int curCol = Col + blurCol;
        // Verify we have a valid image pixel
        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
            pixVal += in[curRow * w + curCol];
            pixels++; // Keep track of number of pixels in the
accumulated total
        }
    }
}

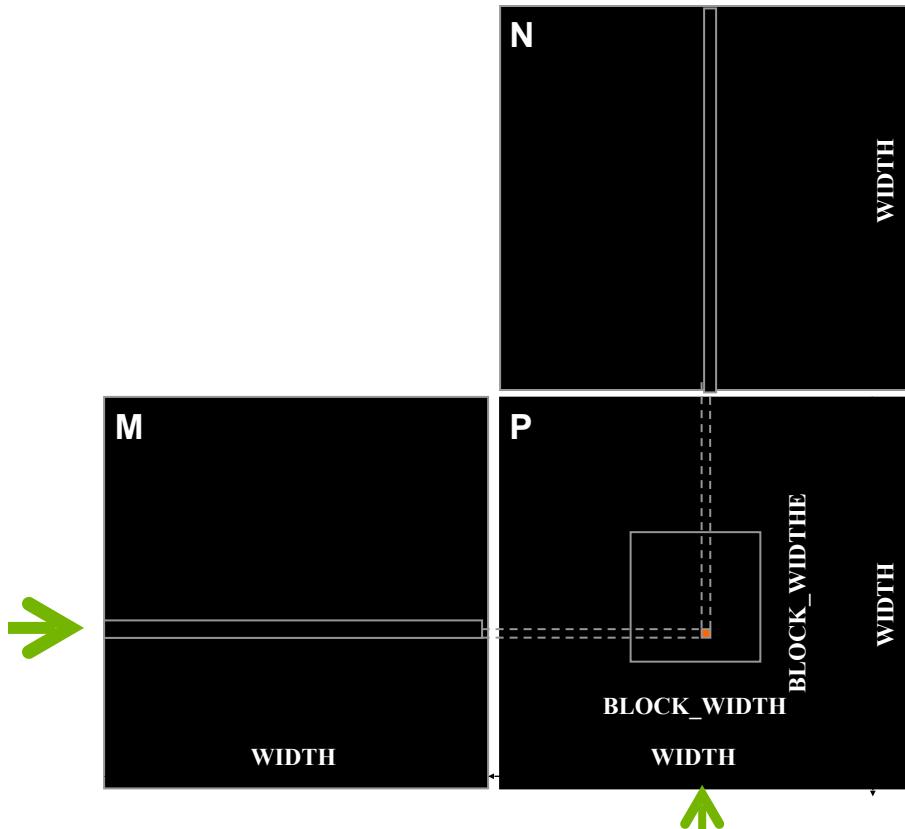
// Write our new pixel value out
out[Row * w + Col] = (unsigned char)(pixVal / pixels);
```



How about performance on a GPU

- All threads access global memory for their input matrix elements
- One memory accesses (4 bytes) per floating-point addition
- 4B/s of memory bandwidth/FLOPS
- Assume a GPU with
 - Peak floating-point rate 1,500 GFLOPS with 200 GB/s DRAM bandwidth
 - $4 \times 1,500 = 6,000$ GB/s required to achieve peak FLOPS rating
 - The 200 GB/s memory bandwidth limits the execution at 50 GFLOPS
- This limits the execution rate to 3.3% ($50/1500$) of the peak floating-point execution rate of the device!
- Need to drastically cut down memory accesses to get close to the 1,500 GFLOPS

Example – Matrix Multiplication



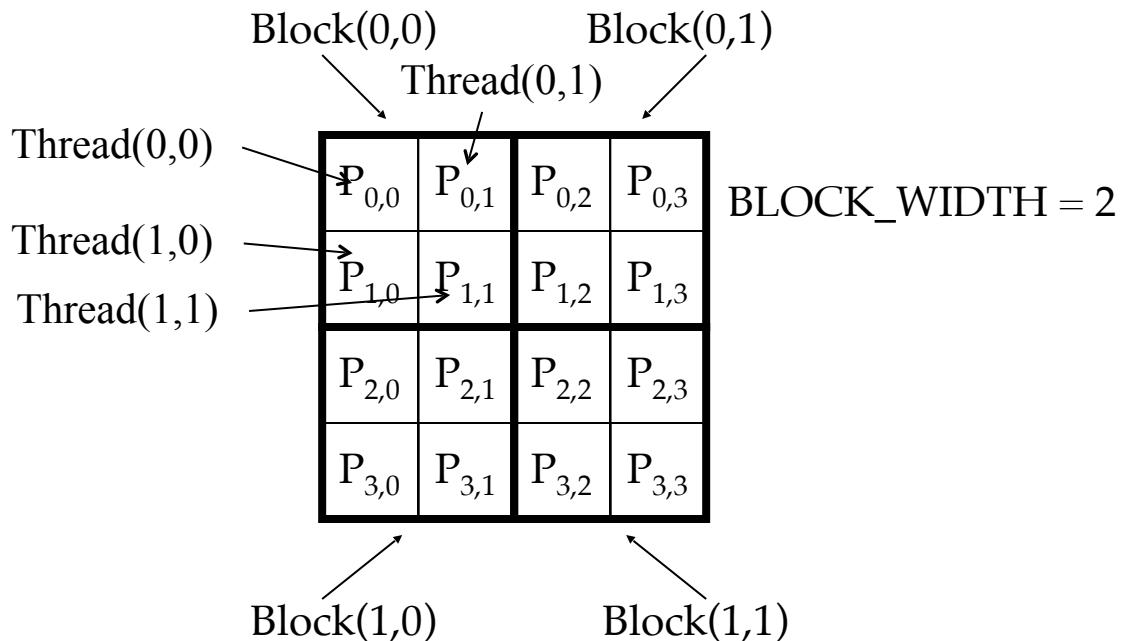
A Basic Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

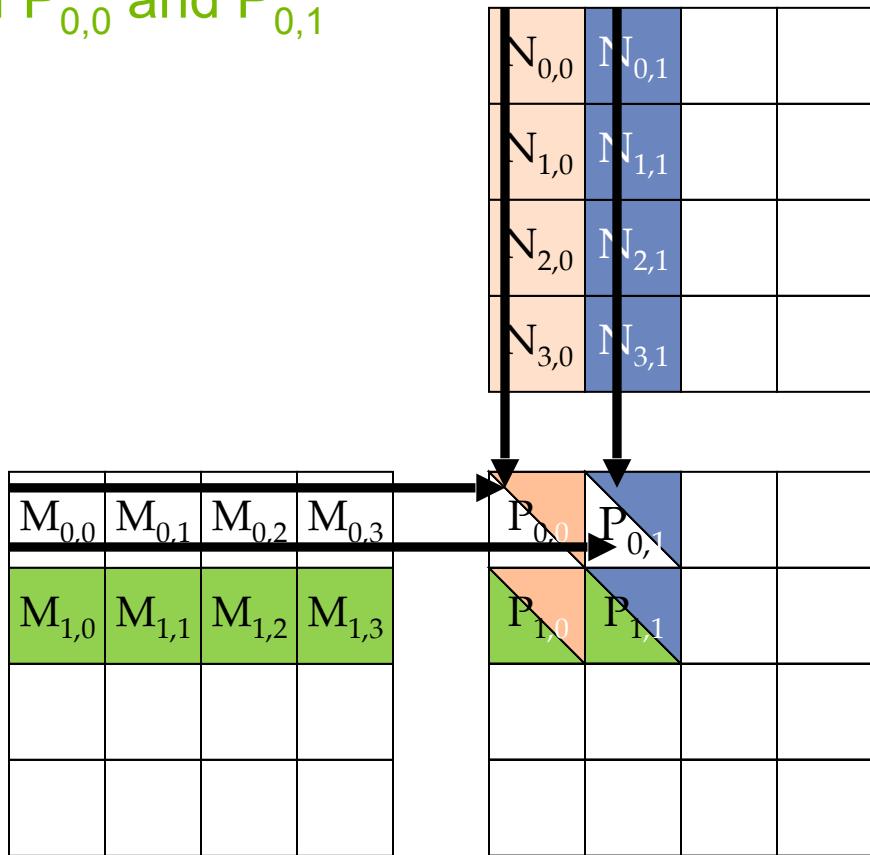
Example – Matrix Multiplication

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {  
    // Calculate the row index of the P element and M  
    int Row = blockIdx.y*blockDim.y+threadIdx.y;  
  
    // Calculate the column index of P and N  
    int Col = blockIdx.x*blockDim.x+threadIdx.x;  
  
    if ((Row < Width) && (Col < Width)) {  
        float Pvalue = 0;  
        // each thread computes one element of the block sub-matrix  
        for (int k = 0; k < Width; ++k) {  
            Pvalue += M[Row*Width+k]*N[k*Width+Col];  
        }  
        P[Row*Width+Col] = Pvalue;  
    }  
}
```

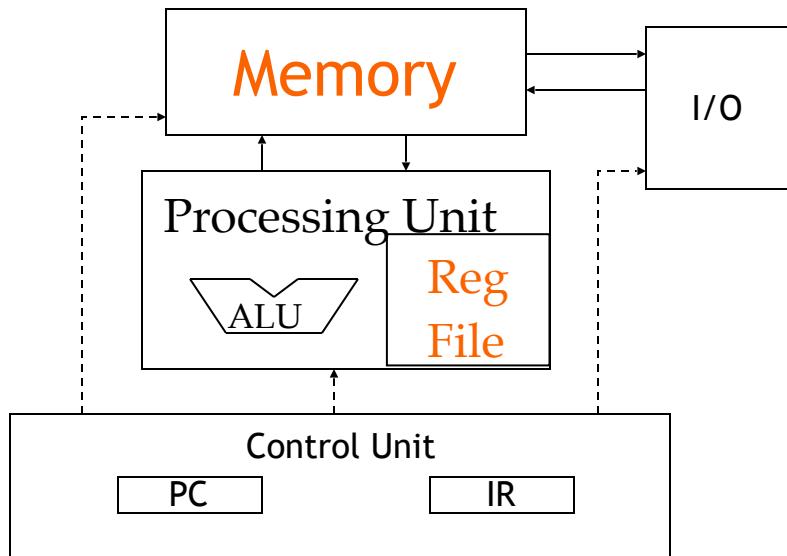
A Toy Example: Thread to P Data Mapping



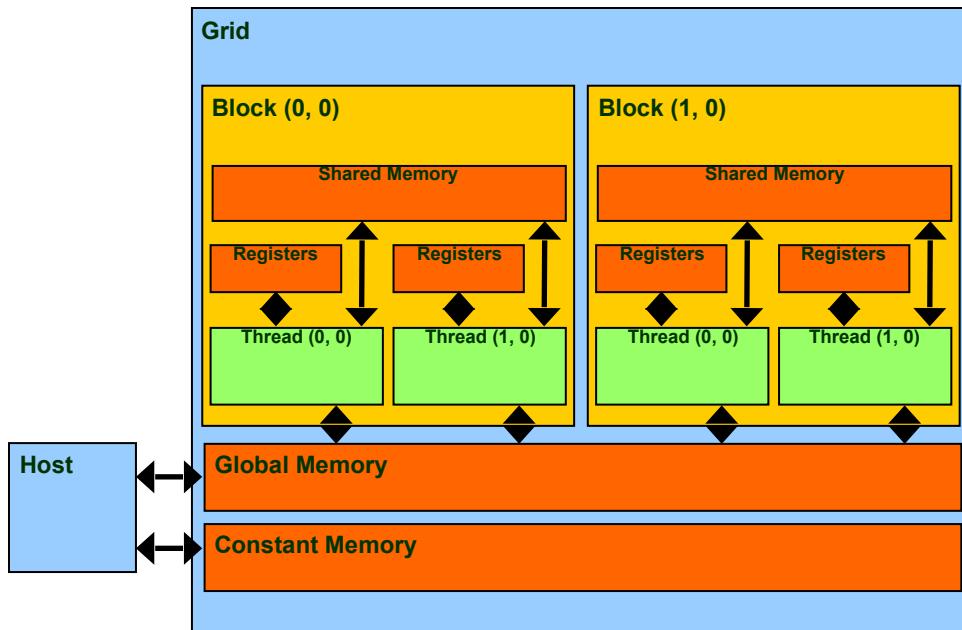
Calculation of $P_{0,0}$ and $P_{0,1}$



Memory and Registers in the Von-Neumann Model



Programmer View of CUDA Memories



Declaring CUDA Variables

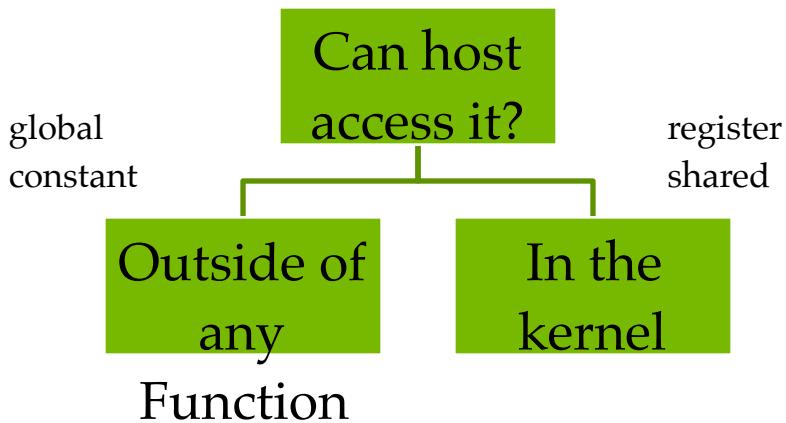
Variable declaration	Memory	Scope	Lifetime
int LocalVar;	register	thread	thread
<code>__device__ __shared__ int SharedVar;</code>	shared	block	block
<code>__device__ int GlobalVar;</code>	global	grid	application
<code>__device__ __constant__ int ConstantVar;</code>	constant	grid	application

- `__device__` is optional when used with `__shared__`, or `__constant__`
- Automatic variables reside in a register
- Except per-thread arrays that reside in global memory

Example: Shared Memory Variable Declaration

```
void blurKernel(unsigned char * in, unsigned char * out,  
int w, int h)  
{  
    __shared__ float ds_in[TILE_WIDTH] [TILE_WIDTH];  
    ...  
}
```

Where to Declare Variables?



Shared Memory in CUDA

- A special type of memory whose contents are explicitly defined and used in the kernel source code
- One in each SM
- Accessed at much higher speed (in both latency and throughput) than global memory
- Scope of access and sharing - thread blocks
- Lifetime – thread block, contents will disappear after the corresponding thread finishes terminates execution
- Accessed by memory load/store instructions
- A form of scratchpad memory in computer architecture

CSC447 Parallel Programming

Lecture 9 – Memory and Data Locality

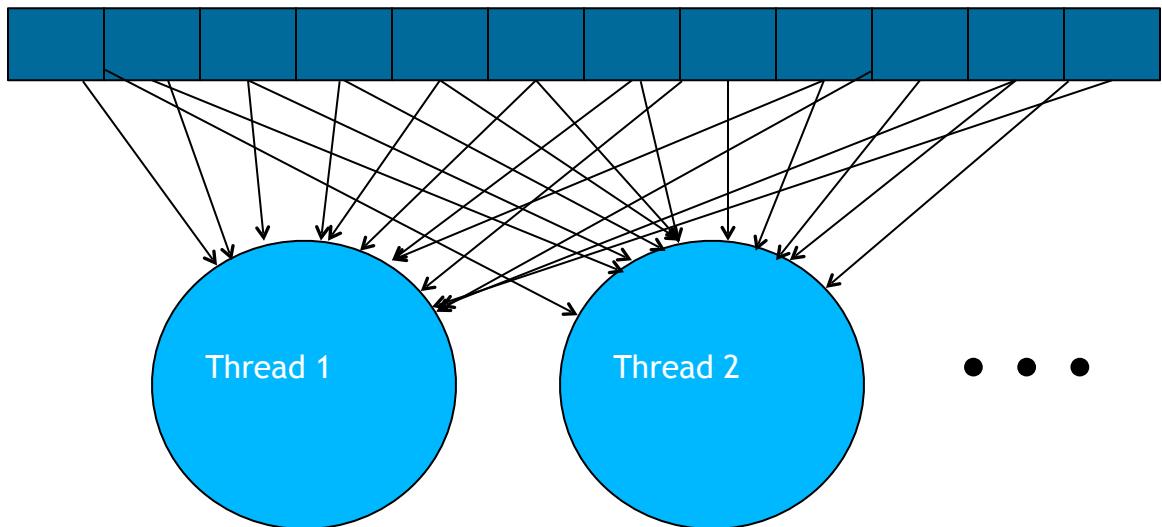
Tiled Parallel Algorithms

Objective

- To understand the motivation and ideas for tiled parallel algorithms
- Reducing the limiting effect of memory bandwidth on parallel kernel performance
- Tiled algorithms and barrier synchronization

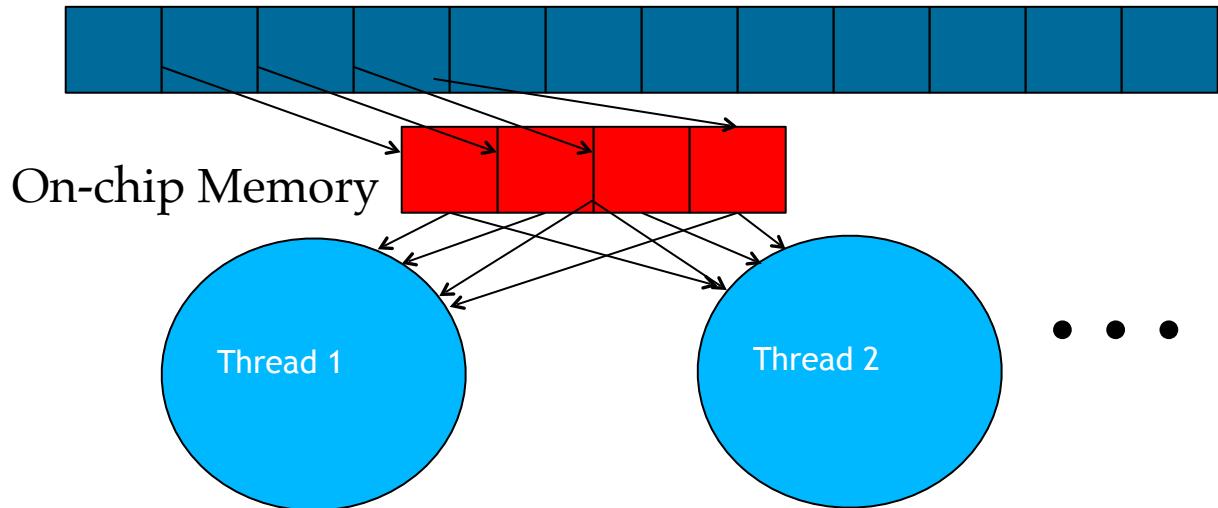
Global Memory Access Pattern of the Basic Matrix Multiplication Kernel

Global Memory



Tiling/Blocking - Basic Idea

Global Memory

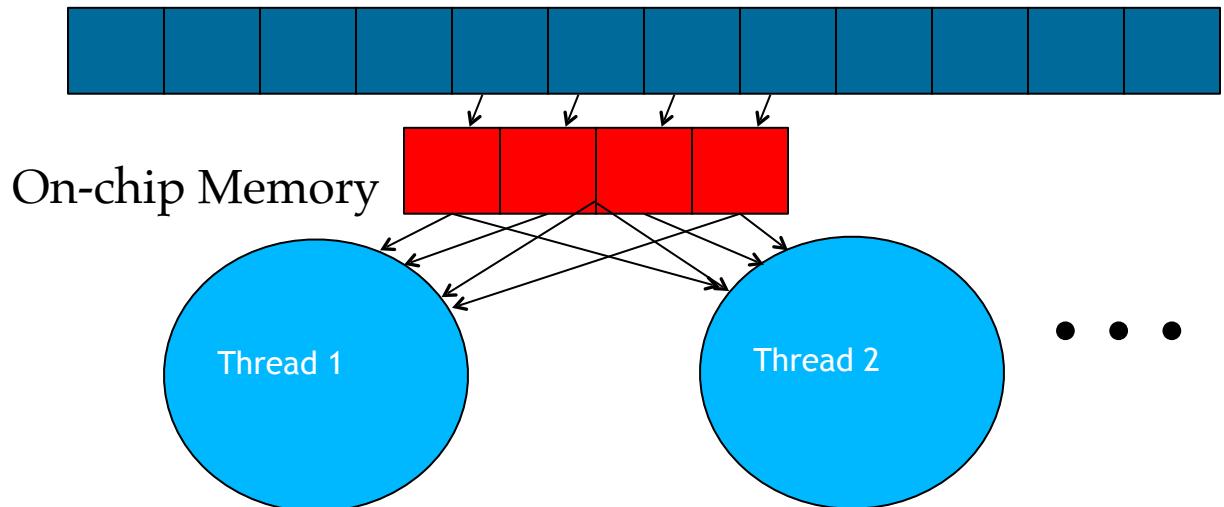


Divide the global memory content into tiles

Focus the computation of threads on one or a small number of tiles at each point in time

Tiling/Blocking - Basic Idea

Global Memory



Basic Concept of Tiling

- In a congested traffic system, significant reduction of vehicles can greatly improve the delay seen by all vehicles
- Carpooling for commuters
- Tiling for global memory accesses
- drivers = threads accessing their memory data operands
- cars = memory access requests



Some Computations are More Challenging to Tile

- Some carpools may be easier than others
- Car pool participants need to have similar work schedule
- Some vehicles may be more suitable for carpooling
- Similar challenges exist in tiling



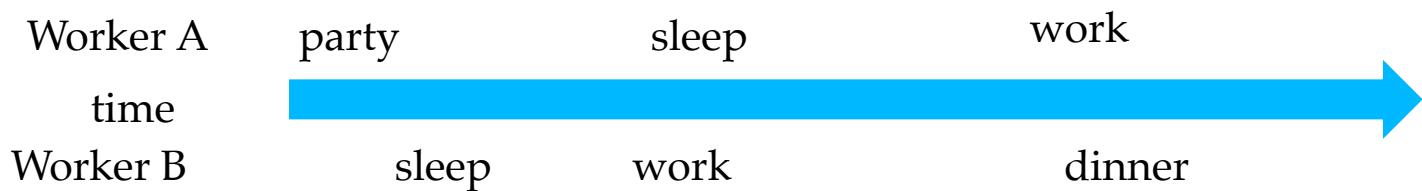
Carpools need synchronization.

- Good: when people have similar schedule



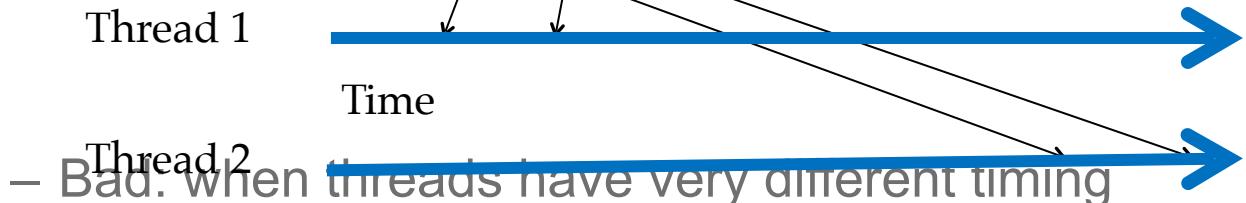
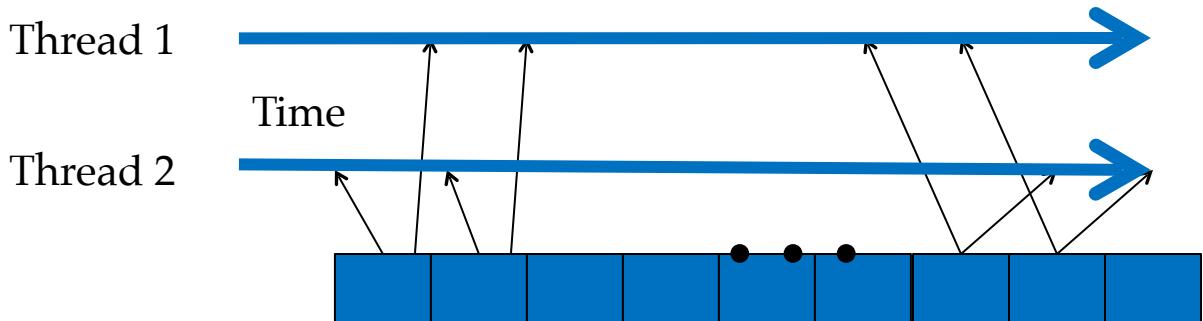
Carpools need synchronization.

- Bad: when people have very different schedule

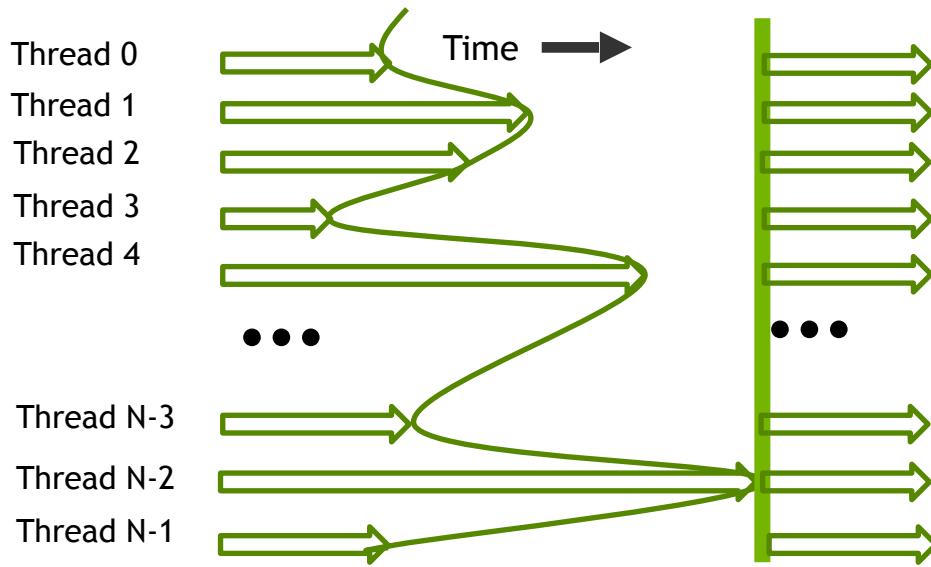


Same with Tiling

- Good: when threads have similar access timing



Barrier Synchronization for Tiling



Outline of Tiling Technique

- Identify a tile of global memory contents that are accessed by multiple threads
- Load the tile from global memory into on-chip memory
- Use barrier synchronization to make sure that all threads are ready to start the phase
- Have the multiple threads to access their data from the on-chip memory
- Use barrier synchronization to make sure that all threads have completed the current phase
- Move on to the next tile

CSC 447 Parallel Programming

Lecture 10 - Memory Model and Locality

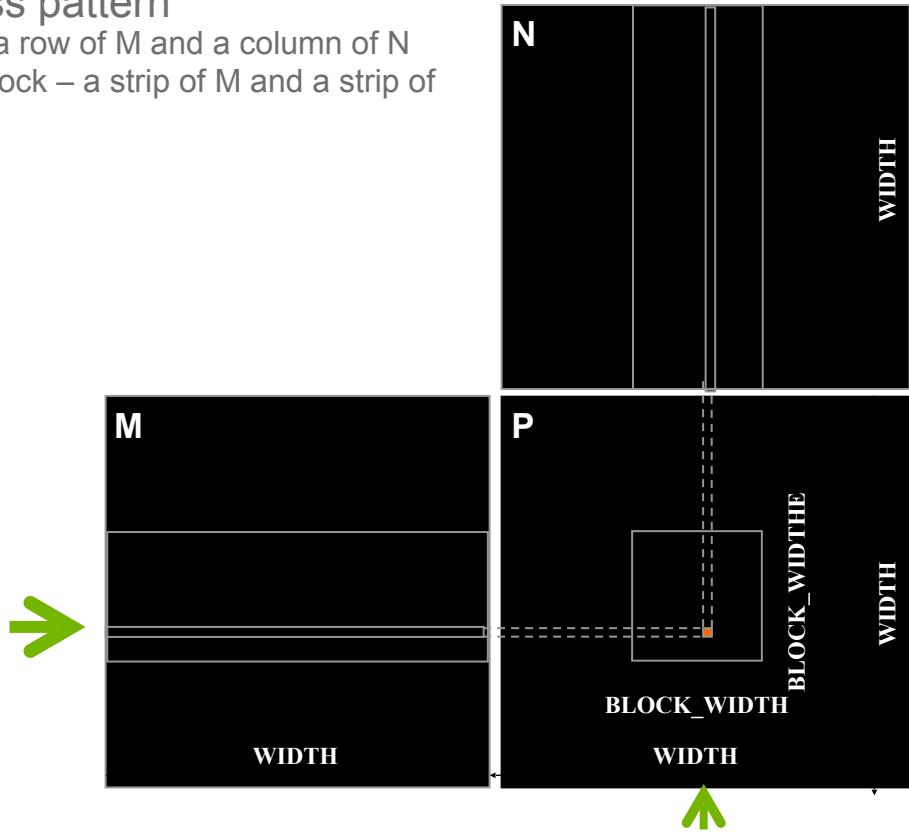
Tiled Matrix Multiplication

Objective

- To understand the design of a tiled parallel algorithm for matrix multiplication
- Loading a tile
- Phased execution
- Barrier Synchronization

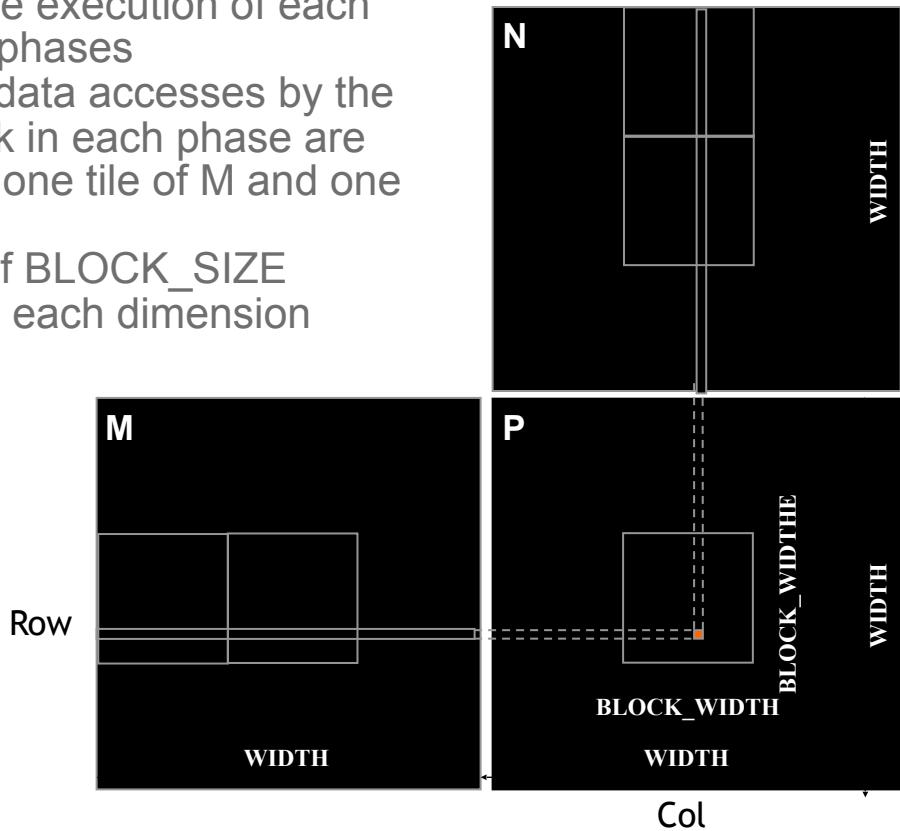
Matrix Multiplication

- Data access pattern
- Each thread - a row of M and a column of N
- Each thread block – a strip of M and a strip of N



Tiled Matrix Multiplication

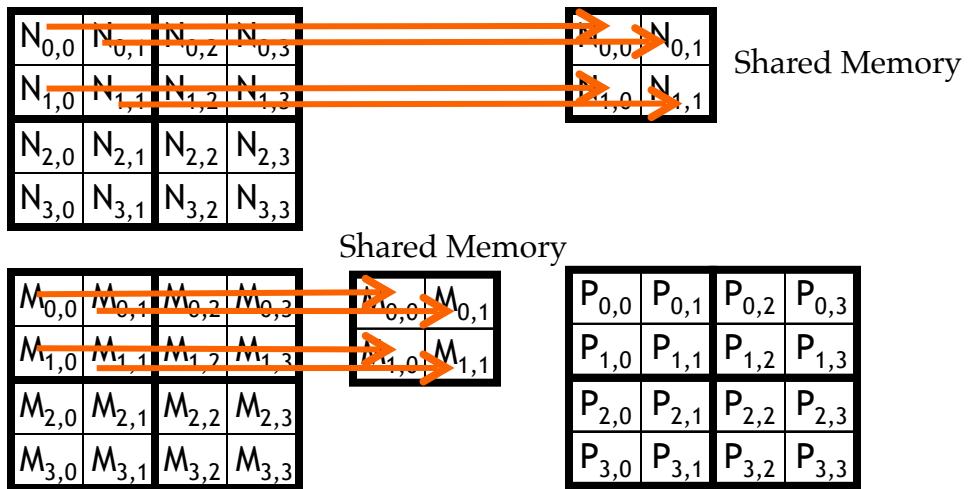
- Break up the execution of each thread into phases
- so that the data accesses by the thread block in each phase are focused on one tile of M and one tile of N
- The tile is of BLOCK_SIZE elements in each dimension



Loading a Tile

- All threads in a block participate
- Each thread loads one M element and one N element in tiled code

Phase 0 Load for Block (0,0)



Phase 0 Use for Block (0,0) (iteration 0)

N _{0,0}	N _{0,1}	N _{0,2}	N _{0,3}
N _{1,0}	N _{1,1}	N _{1,2}	N _{1,3}
N _{2,0}	N _{2,1}	N _{2,2}	N _{2,3}
N _{3,0}	N _{3,1}	N _{3,2}	N _{3,3}

N _{0,0}	N _{0,1}
N _{1,0}	N _{1,1}

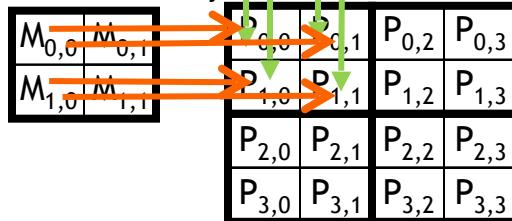
Shared Memory

M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}

Shared Memory

M _{0,0}	M _{0,1}
M _{1,0}	M _{1,1}

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}



Phase 0 Use for Block (0,0) (iteration 1)

N _{0,0}	N _{0,1}	N _{0,2}	N _{0,3}
N _{1,0}	N _{1,1}	N _{1,2}	N _{1,3}
N _{2,0}	N _{2,1}	N _{2,2}	N _{2,3}
N _{3,0}	N _{3,1}	N _{3,2}	N _{3,3}

M _{0,0}	M _{0,1}	M _{0,2}	M _{0,3}
M _{1,0}	M _{1,1}	M _{1,2}	M _{1,3}
M _{2,0}	M _{2,1}	M _{2,2}	M _{2,3}
M _{3,0}	M _{3,1}	M _{3,2}	M _{3,3}

N _{0,0}	N _{0,1}
N _{1,0}	N _{1,1}

Shared Memory

Shared Memory

M _{0,0}	M _{0,1}
M _{1,0}	M _{1,1}

M _{0,0}	M _{0,1}
M _{1,0}	M _{1,1}

P _{0,0}	P _{0,1}	P _{0,2}	P _{0,3}
P _{1,0}	P _{1,1}	P _{1,2}	P _{1,3}
P _{2,0}	P _{2,1}	P _{2,2}	P _{2,3}
P _{3,0}	P _{3,1}	P _{3,2}	P _{3,3}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

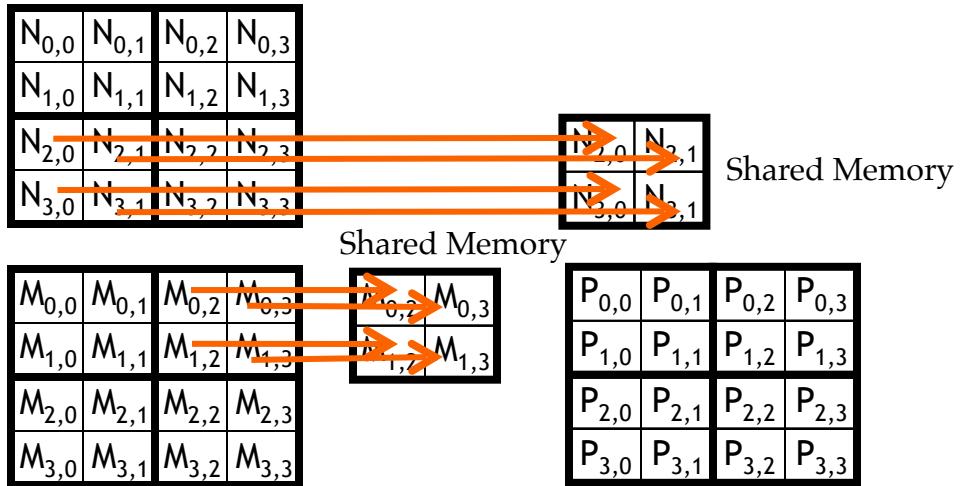
P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0}	P _{1,1}

P _{0,0}	P _{0,1}
P _{1,0} </	

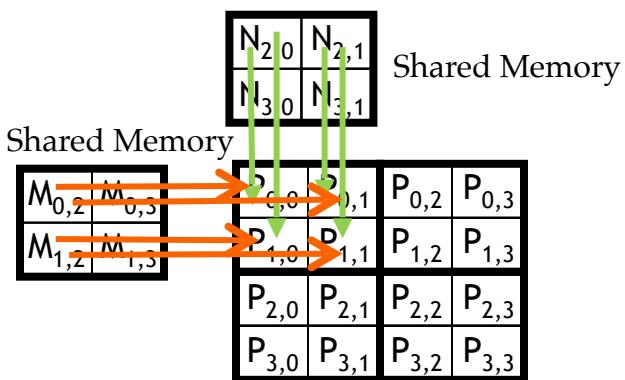
Phase 1 Load for Block (0,0)



Phase 1 Use for Block (0,0) (iteration 0)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

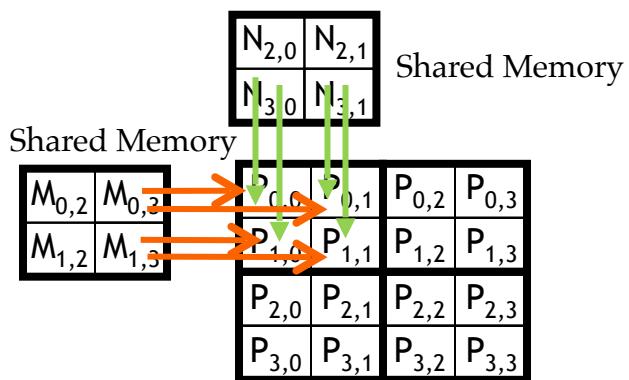
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Phase 1 Use for Block (0,0) (iteration 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	$N_{0,3}$
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	$N_{1,3}$
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$
$N_{3,0}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$
$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$



Execution Phases of Toy Example

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

time



Execution Phases of Toy Example (cont.)

	Phase 0			Phase 1		
thread _{0,0}	M_{0,0} ↓ Mds _{0,0}	N_{0,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}	M_{0,2} ↓ Mds _{0,0}	N_{2,0} ↓ Nds _{0,0}	PValue _{0,0} += Mds _{0,0} *Nds _{0,0} + Mds _{0,1} *Nds _{1,0}
thread _{0,1}	M_{0,1} ↓ Mds _{0,1}	N_{0,1} ↓ Nds _{1,0}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}	M_{0,3} ↓ Mds _{0,1}	N_{2,1} ↓ Nds _{0,1}	PValue _{0,1} += Mds _{0,0} *Nds _{0,1} + Mds _{0,1} *Nds _{1,1}
thread _{1,0}	M_{1,0} ↓ Mds _{1,0}	N_{1,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}	M_{1,2} ↓ Mds _{1,0}	N_{3,0} ↓ Nds _{1,0}	PValue _{1,0} += Mds _{1,0} *Nds _{0,0} + Mds _{1,1} *Nds _{1,0}
thread _{1,1}	M_{1,1} ↓ Mds _{1,1}	N_{1,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}	M_{1,3} ↓ Mds _{1,1}	N_{3,1} ↓ Nds _{1,1}	PValue _{1,1} += Mds _{1,0} *Nds _{0,1} + Mds _{1,1} *Nds _{1,1}

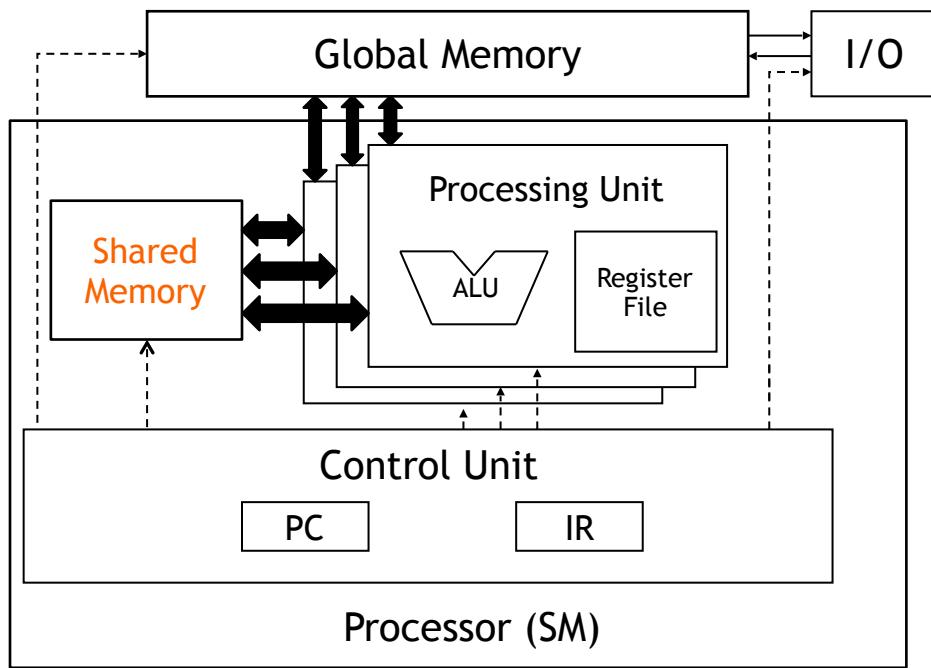
time →

Shared memory allows each value to be accessed by multiple threads

Barrier Synchronization

- Synchronize all threads in a block
- `__syncthreads()`
- All threads in the same block must reach the `__syncthreads()` before any of them can move on
- Best used to coordinate the phased execution tiled algorithms
- To ensure that all elements of a tile are loaded at the beginning of a phase
- To ensure that all elements of a tile are consumed at the end of a phase

Hardware View of CUDA Memories



CSC 447 Parallel Programming

Lecture 11- Memory and Data Locality

Tiled Matrix Multiplication Kernel

Objective

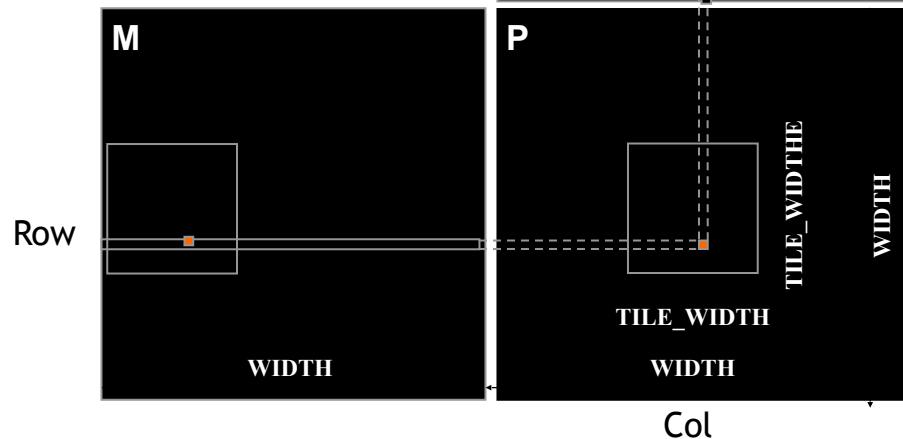
- To learn to write a tiled matrix-multiplication kernel
- Loading and using tiles for matrix multiplication
- Barrier synchronization, shared memory
- Resource Considerations
- Assume that Width is a multiple of tile size for simplicity

Loading Input Tile 0 of M (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;  
2D indexing for accessing Tile 0:
```

M[Row][tx]
N[ty][Col]

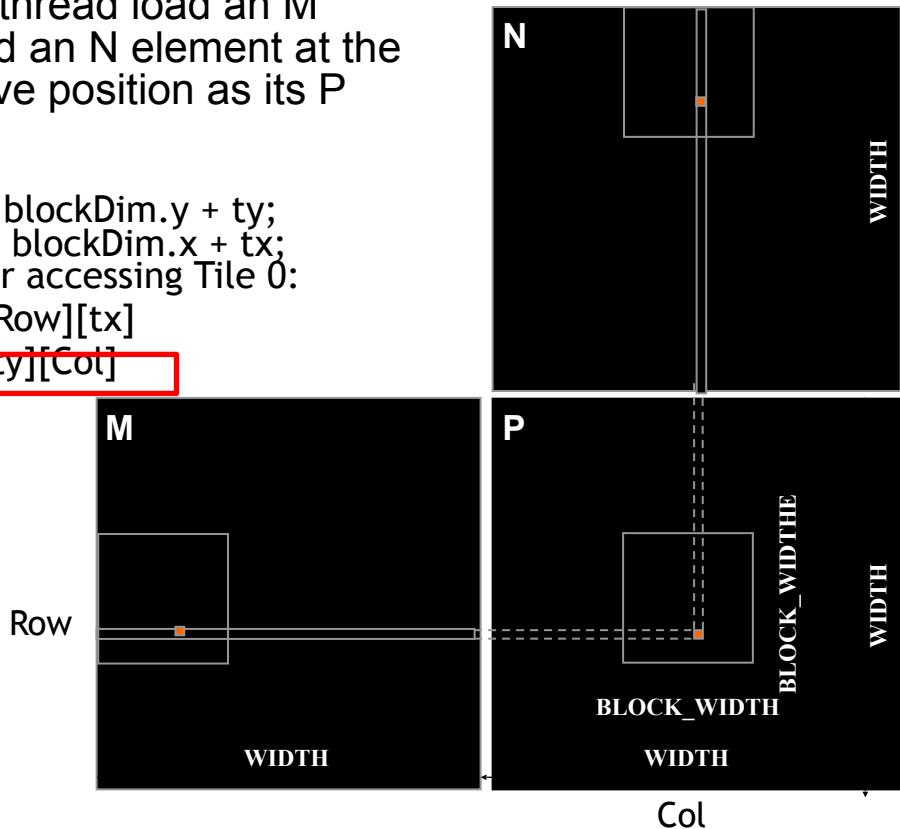


Loading Input Tile 0 of N (Phase 0)

- Have each thread load an M element and an N element at the same relative position as its P element.

```
int Row = by * blockDim.y + ty;  
int Col = bx * blockDim.x + tx;  
2D indexing for accessing Tile 0:
```

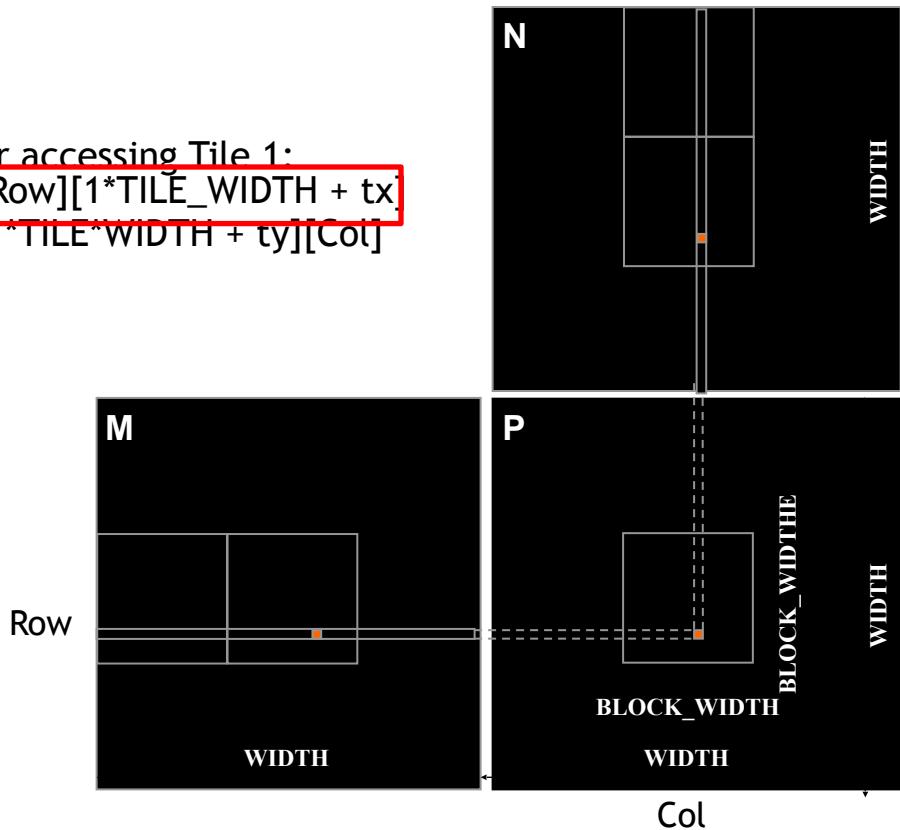
M[Row][tx]
N[ty][Col]



Loading Input Tile 1 of M (Phase 1)

2D indexing for accessing Tile 1:

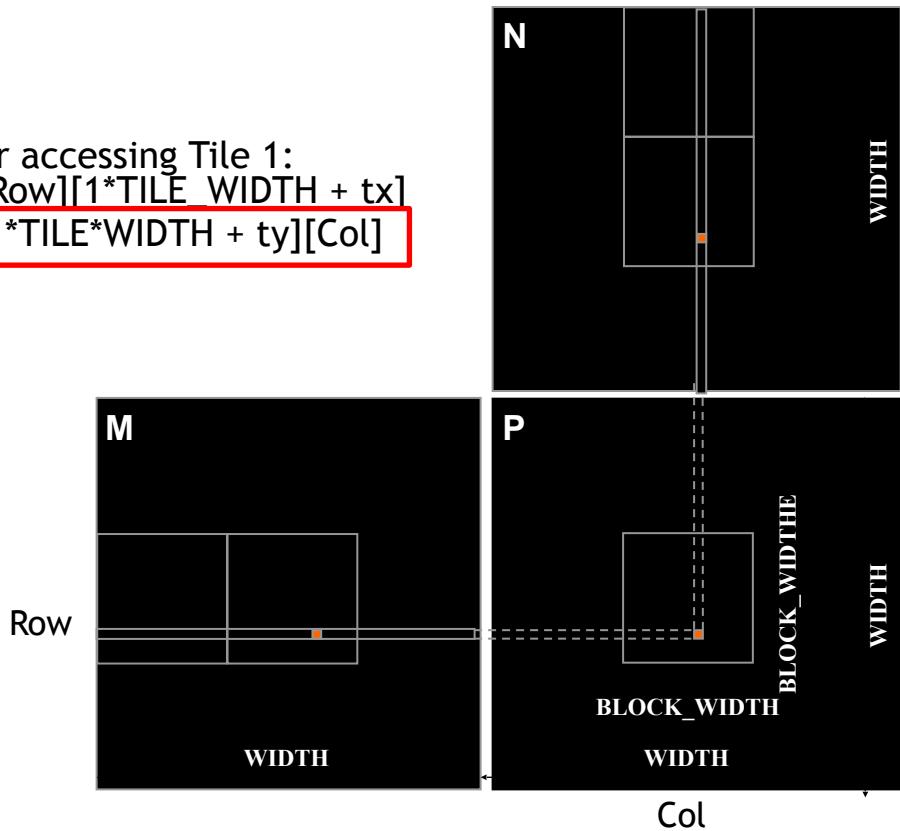
$M[\text{Row}][1 * \text{TILE_WIDTH} + tx]$
 $N[1 * \text{TILE_WIDTH} + ty][\text{Col}]$



Loading Input Tile 1 of N (Phase 1)

2D indexing for accessing Tile 1:

$M[\text{Row}][1 * \text{TILE_WIDTH} + tx]$
 $N[1 * \text{TILE_WIDTH} + ty][\text{Col}]$



M and N are dynamically allocated - use 1D indexing

- $M[\text{Row}][p * \text{TILE_WIDTH} + tx]$
 $M[\text{Row} * \text{Width} + p * \text{TILE_WIDTH} + tx]$

- $N[p * \text{TILE_WIDTH} + ty][\text{Col}]$
 $N[(p * \text{TILE_WIDTH} + ty) * \text{Width} + \text{Col}]$

where p is the sequence number of the current phase

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH][TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH][TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH] [TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH] [TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Tiled Matrix Multiplication Kernel

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, Int Width)
{
    __shared__ float ds_M[TILE_WIDTH] [TILE_WIDTH];
    __shared__ float ds_N[TILE_WIDTH] [TILE_WIDTH];

    int bx = blockIdx.x;  int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    int Row = by * blockDim.y + ty;
    int Col = bx * blockDim.x + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    for (int p = 0; p < n/TILE_WIDTH; ++p) {
        // Collaborative loading of M and N tiles into shared memory
        ds_M[ty][tx] = M[Row*Width + p*TILE_WIDTH+tx];
        ds_N[ty][tx] = N[(t*TILE_WIDTH+ty)*Width + Col];
        __syncthreads();

        for (int i = 0; i < TILE_WIDTH; ++i) Pvalue += ds_M[ty][i] * ds_N[i][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```



Tile (Thread Block) Size Considerations

- Each **thread block** should have many threads
- TILE_WIDTH of 16 gives $16 \times 16 = 256$ threads
- TILE_WIDTH of 32 gives $32 \times 32 = 1024$ threads
- For 16, in each phase, each block performs $2 \times 256 = 512$ float loads from global memory for $256 \times (2 \times 16) = 8,192$ mul/add operations. (16 floating-point operations for each memory load)
- For 32, in each phase, each block performs $2 \times 1024 = 2048$ float loads from global memory for $1024 \times (2 \times 32) = 65,536$ mul/add operations. (32 floating-point operation for each memory load)

Shared Memory and Threading

- For an SM with 16KB shared memory
- Shared memory size is implementation dependent!
- For `TILE_WIDTH = 16`, each thread block uses $2*256*4B = 2KB$ of shared memory.
- For 16KB shared memory, one can potentially have up to 8 thread blocks executing
- This allows up to $8*512 = 4,096$ pending loads. (2 per thread, 256 threads per block)
- The next `TILE_WIDTH 32` would lead to $2*32*32*4$ Byte= 8K Byte shared memory usage per thread block, allowing 2 thread blocks active at the same time
- However, in a GPU where the thread count is limited to 1536 threads per SM, the number of blocks per SM is reduced to one!
- Each `_syncthread()` can reduce the number of active threads for a block
- More thread blocks can be advantageous

CSC 447 Parallel Programming

Lecture 12- Memory and Data Locality

Handling Arbitrary Matrix Sizes in Tiled Algorithms

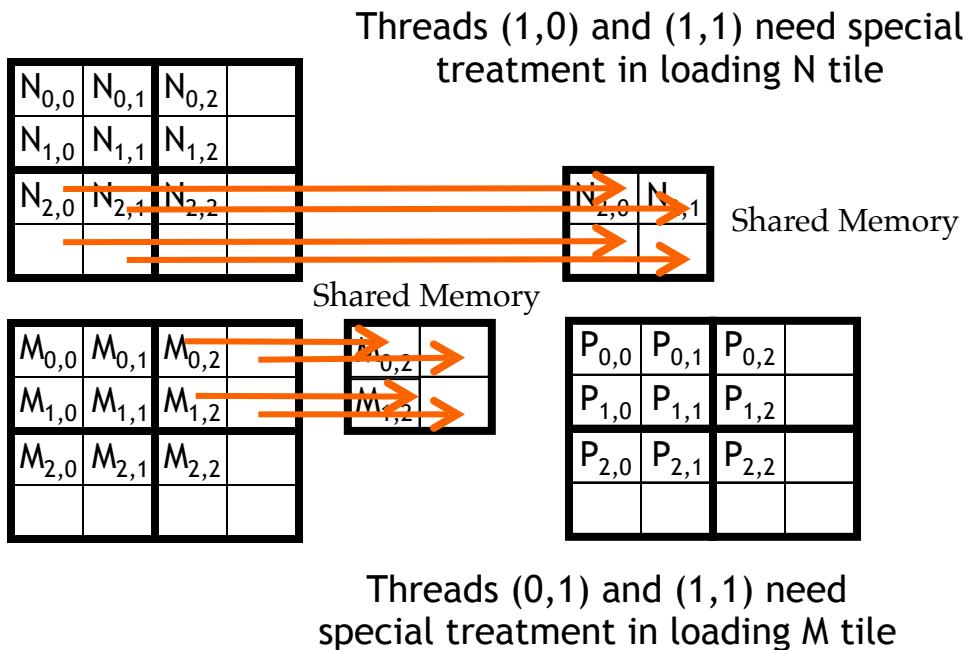
Objective

- To learn to handle arbitrary matrix sizes in tiled matrix multiplication
- Boundary condition checking
- Regularizing tile contents
- Rectangular matrices

Handling Matrix of Arbitrary Size

- The tiled matrix multiplication kernel we presented so far can handle only square matrices whose dimensions (Width) are multiples of the tile width (TILE_WIDTH)
- However, real applications need to handle arbitrary sized matrices.
- One could pad (add elements to) the rows and columns into multiples of the tile size, but would have significant space and data transfer time overhead.
- We will take a different approach.

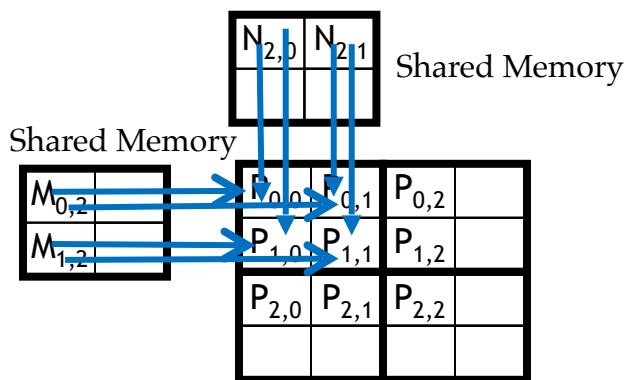
Phase 1 Loads for Block (0,0) for a 3x3 Example



Phase 1 Use for Block (0,0) (iteration 0)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

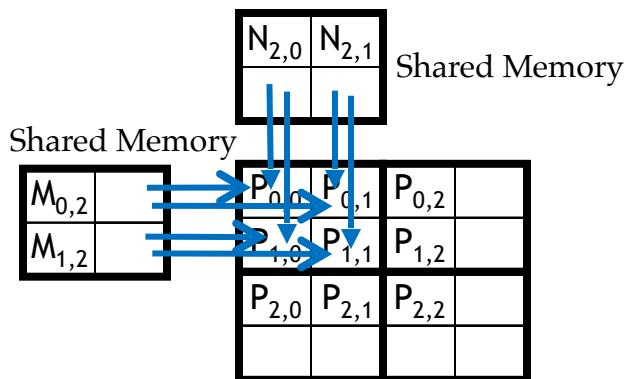
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



Phase 1 Use for Block (0,0) (iteration 1)

$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

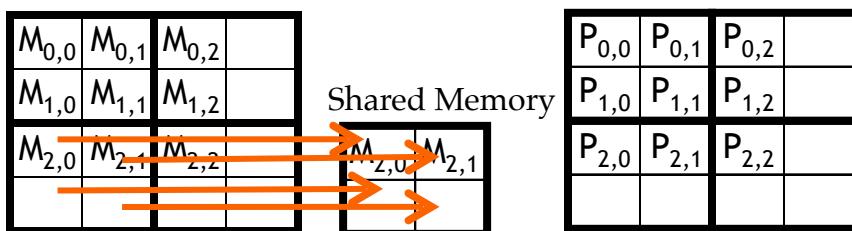
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



All Threads need special treatment.
None of them should introduce
invalidate contributions to their P
elements.

Phase 0 Loads for Block (1,1) for a 3x3 Example

Threads (0,1) and (1,1) need special treatment in loading N tile



Threads (1,0) and (1,1) need special treatment in loading M tile

Major Cases in Toy Example

- Threads that do not calculate valid P elements but still need to participate in loading the input tiles
- Phase 0 of Block(1,1), Thread(1,0), assigned to calculate non-existent P[3,2] but need to participate in loading tile element N[1,2]
- Threads that calculate valid P elements may attempt to load non-existing input elements when loading input tiles
- Phase 0 of Block(0,0), Thread(1,0), assigned to calculate valid P[1,0] but attempts to load non-existing N[3,0]

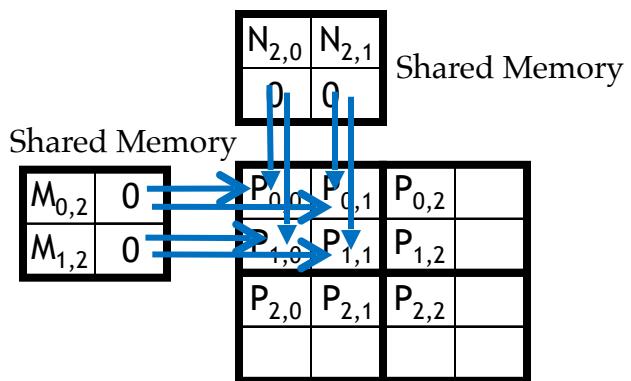
A “Simple” Solution

- When a thread is to load any input element, test if it is in the valid index range
- If valid, proceed to load
- Else, do not load, just write a 0
- Rationale: a 0 value will ensure that that the multiply-add step does not affect the final value of the output element
- The condition tested for loading input elements is different from the test for calculating output P element
- A thread that does not calculate valid P element can still participate in loading input tile elements

Phase 1 Use for Block (0,0) (iteration 1)

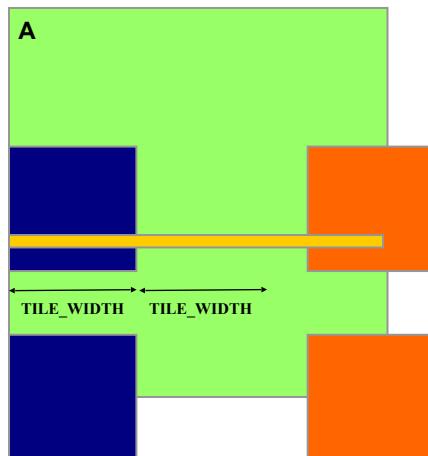
$N_{0,0}$	$N_{0,1}$	$N_{0,2}$	
$N_{1,0}$	$N_{1,1}$	$N_{1,2}$	
$N_{2,0}$	$N_{2,1}$	$N_{2,2}$	

$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	
$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	
$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	



Boundary Condition for Input M Tile

- Each thread loads
- $M[\text{Row}][p * \text{TILE_WIDTH} + tx]$
- $M[\text{Row} * \text{Width} + p * \text{TILE_WIDTH} + tx]$
- Need to test
- $(\text{Row} < \text{Width}) \&\& (p * \text{TILE_WIDTH} + tx < \text{Width})$
- If true, load M element
- Else , load 0



Boundary Condition for Input N Tile

- Each thread loads
- $N[p \cdot \text{TILE_WIDTH} + ty][\text{Col}]$
- $N[(p \cdot \text{TILE_WIDTH} + ty) \cdot \text{Width} + \text{Col}]$
- Need to test
- $(p \cdot \text{TILE_WIDTH} + ty < \text{Width}) \&\& (\text{Col} < \text{Width})$
- If true, load N element
- Else , load 0



Loading Elements – with boundary check

```
- 8  for (int p = 0; p < (Width-1) / TILE_WIDTH + 1; ++p) {  
-  
-  ++      if(Row < Width && t * TILE_WIDTH+tx < Width) {  
-  9          ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];  
-  ++      } else {  
-  ++          ds_M[ty][tx] = 0.0;  
-  ++      }  
-  ++      if (p*TILE_WIDTH+ty < Width && Col < Width) {  
-  10         ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];  
-  ++      } else {  
-  ++          ds_N[ty][tx] = 0.0;  
-  ++      }  
-  11      __syncthreads();  
-
```

Inner Product – Before and After

```
- 11  ++ if(Row < Width && Col < Width) {  
- 12    for (int i = 0; i < TILE_WIDTH; ++i) {  
- 13      Pvalue += ds_M[ty][i] * ds_N[i][tx];  
- 14    }  
- 15  } /* end of outer for loop */  
- 16  ++ if (Row < Width && Col < Width)  
- 17    P[Row*Width + Col] = Pvalue;  
- 18 } /* end of kernel */
```

Some Important Points

- For each thread the conditions are different for
- Loading M element
- Loading N element
- Calculating and storing output elements
- The effect of control divergence should be small for large matrices

Handling General Rectangular Matrices

- In general, the matrix multiplication is defined in terms of rectangular matrices
- A $j \times k$ M matrix multiplied with a $k \times l$ N matrix results in a $j \times l$ P matrix
- We have presented square matrix multiplication, a special case
- The kernel function needs to be generalized to handle general rectangular matrices
- The Width argument is replaced by three arguments: j, k, l
- When Width is used to refer to the height of M or height of P, replace it with j
- When Width is used to refer to the width of M or height of N, replace it with k
- When Width is used to refer to the width of N or width of P, replace it with l