Introduction to High Performance Computing

Gregory G. Howes
Department of Physics and Astronomy
University of Iowa

Iowa High Performance Computing Summer School
University of Iowa
Iowa City, Iowa
1-3 June 2011

Thank you

Jerry Protheroe	Information Technology Services
Ben Rogers	Information Technology Services
Glenn Johnson	Information Technology Services
Mary Grabe	Information Technology Services
Greg Johnson	Information Technology Services
Amir Bozorgzadeh	Information Technology Services
Bill Whitson	Purdue University

and

National Science Foundation

Rosen Center for Advanced Computing, Purdue University

Great Lakes Consortium for Petascale Computing

This presentation borrows heavily from information freely available on the web by

Ian Foster and Blaise Barney

(see references)

Thank you

Jerry Protheroe	Information Technology Services
Ben Rogers	Information Technology Services
Glenn Johnson	Information Technology Services
Mary Grabe	Information Technology Services
Bill Whitson	Purdue University

and

National Science Foundation

Rosen Center for Advanced Computing, Purdue University

Great Lakes Consortium for Petascale Computing

This presentation borrows heavily from information freely available on the web by Ian Foster and Blaise Barney (see references)

Outline

- Introduction
- Thinking in Parallel
- Parallel Computer Architectures
- Parallel Programming Models
- Design of Parallel Algorithms
- References

Introduction

Disclaimer: High Performance Computing (HPC) is valuable to a variety of applications over a very wide range of fields. Many of my examples will come from the world of physics, but I will try to present them in a general sense

Why Use Parallel Computing?

- Single processor speeds are reaching their ultimate limits
- Multi-core processors and multiple processors are the most promising paths to performance improvements

Definition of a parallel computer:

A set of independent processors that can work cooperatively to solve a problem.

Introduction

THE UNIVERSITY OF IOWA

The March towards Petascale Computing

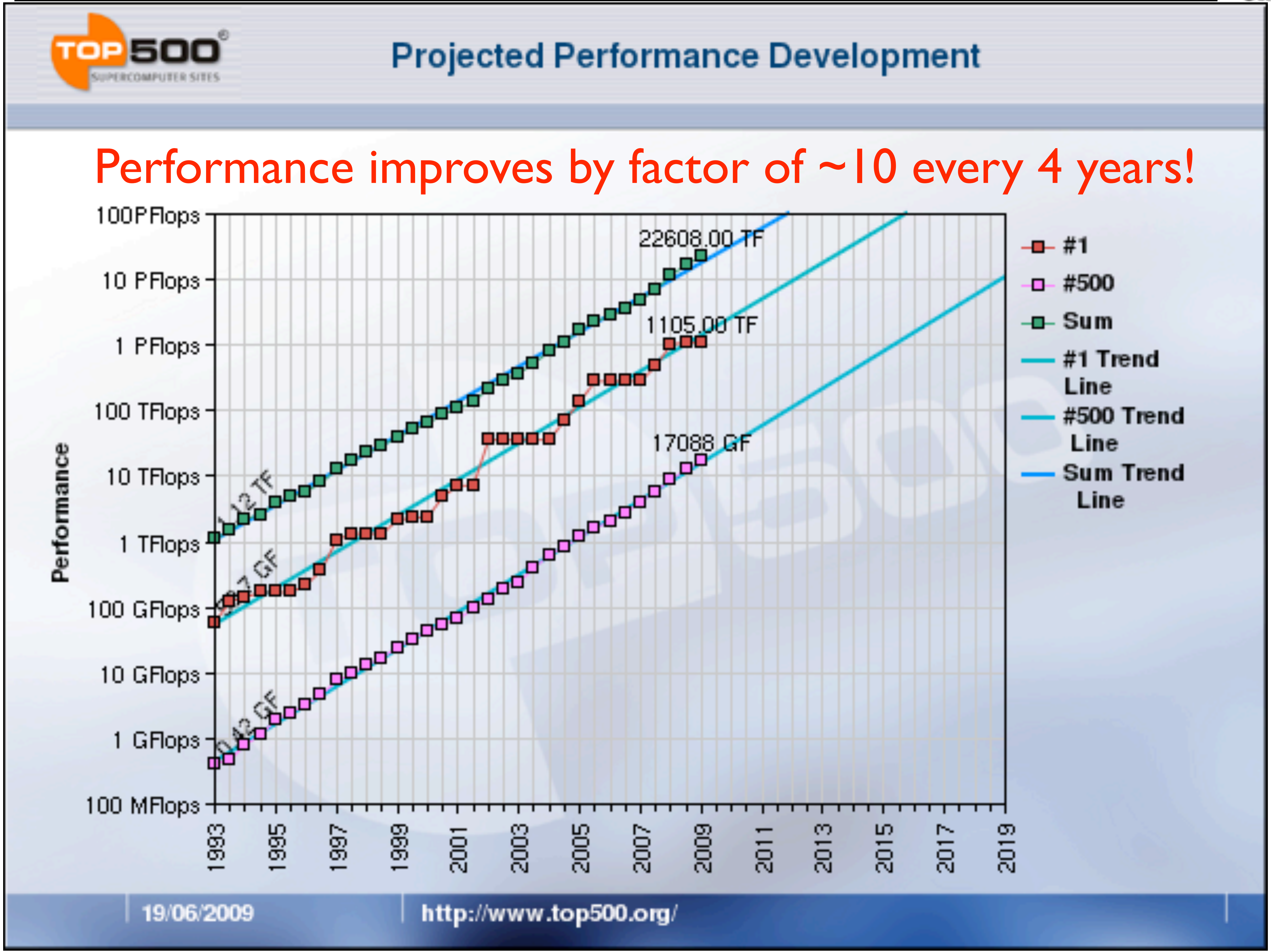
- Computing performance is defined in terms of FLoating-point OPerations per Second (FLOPS)

GigaFLOP	$1 \text{ GF} = 10^9 \text{ FLOPS}$
TeraFLOP	$1 \text{ TF} = 10^{12} \text{ FLOPS}$
PetaFLOP	$1 \text{ PF} = 10^{15} \text{ FLOPS}$

- Petascale computing also refers to extremely large data sets

PetaByte	$1 \text{ PB} = 10^{15} \text{ Bytes}$

Introduction



Outline

- Introduction
- Thinking in Parallel
- Parallel Computer Architectures
- Parallel Programming Models
- Design of Parallel Algorithms
- References

Thinking in Parallel

DEFINITION Concurrency: The property of a parallel algorithm that a number of operations can be performed by separate processors at the same time.

Concurrency is the key concept in the design of parallel algorithms:

- Requires a different way of looking at the strategy to solve a problem
- May require a very different approach from a serial program to achieve high efficiency

Thinking in Parallel

DEFINITION Scalability: The ability of a parallel algorithm to demonstrate a speedup proportional to the number of processors used.

DEFINITION Speedup: The ratio of the serial wallclock time to the parallel wallclock time required for execution.

$$S = \frac{\text{wallclock time}_{serial}}{\text{wallclock time}_{parallel}}$$

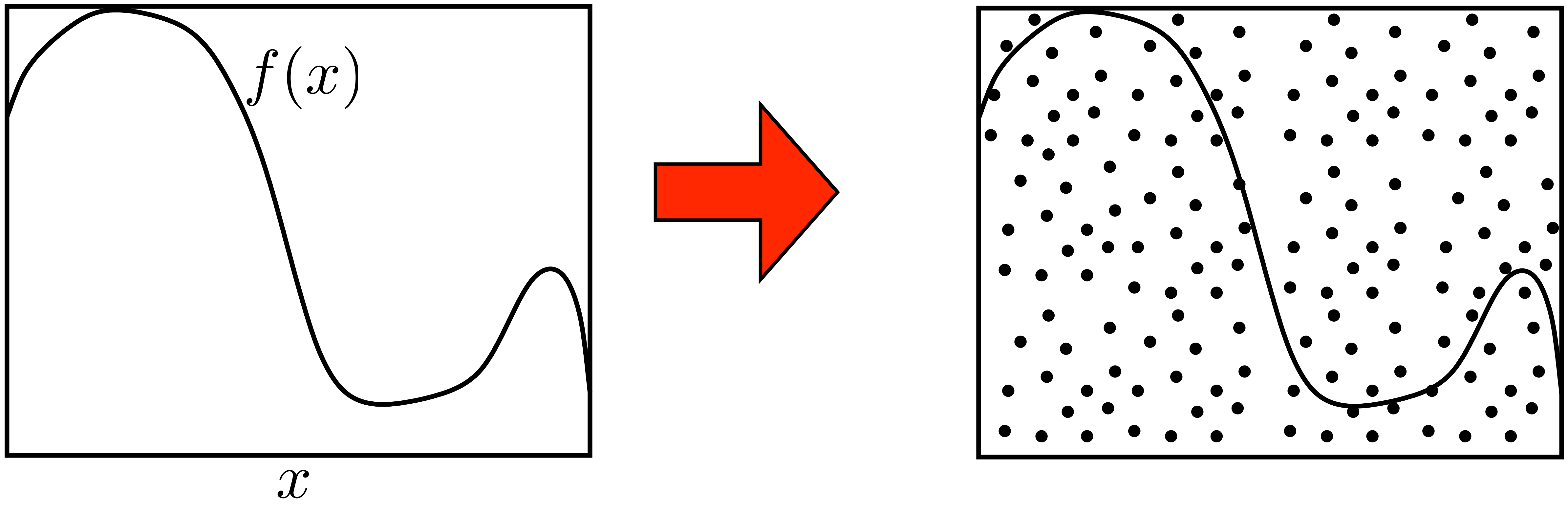
- An algorithm that has good scalability will take half the time with double the number of processors

- Parallel Overhead, the time required to coordinate parallel tasks and communicate information between processors, degrades scalability.

Example: Numerical Integration

Numerical Integration: Monte Carlo Method

- Choose N points within the box of total area A
- Determine the number of points n falling below $f(x)$
- Integral value is $I = \frac{n}{N} A$



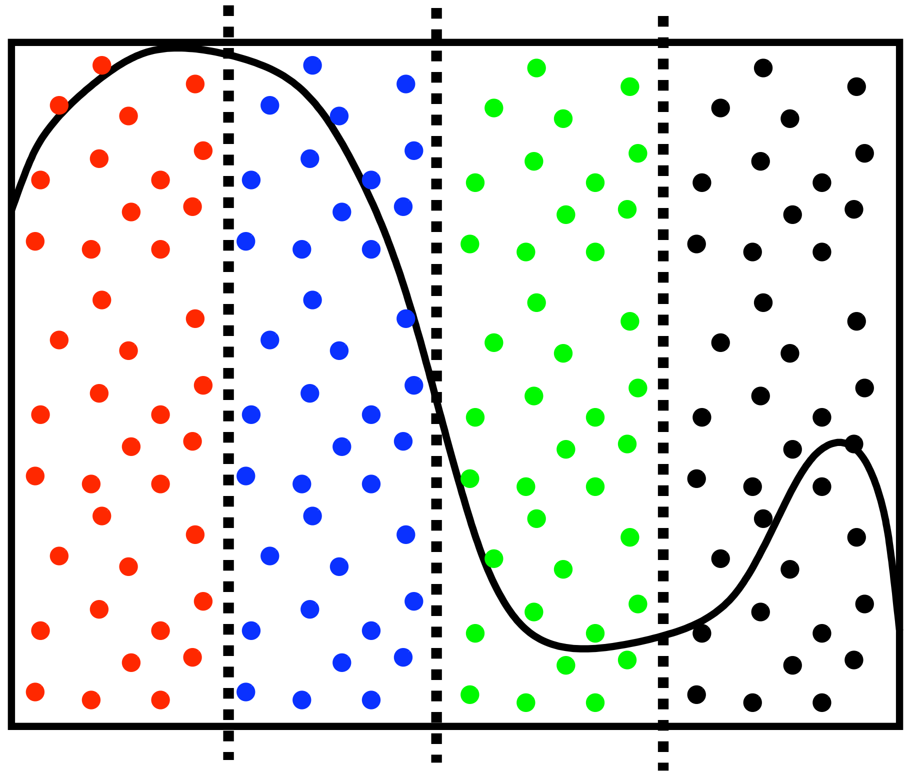
How do we do this computation in parallel?

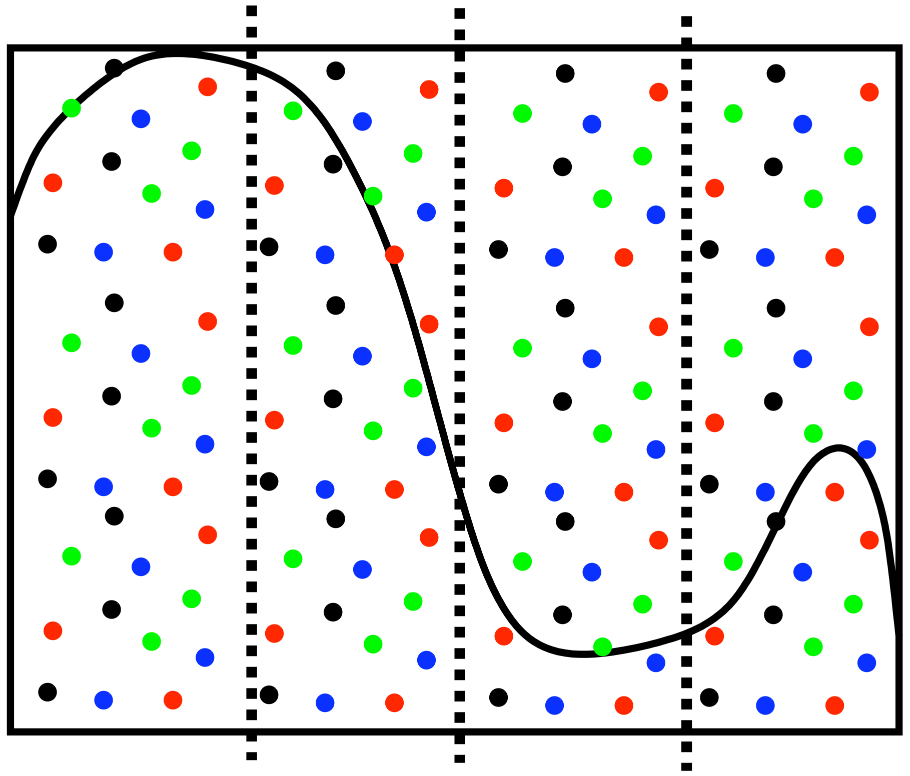
Example: Numerical Integration

Strategies for Parallel Computation of the Numerical Integral:

1) Give different ranges of x to different processors and sum results

2) Give $N/4$ points to each processor and sum results





Example: Fibonacci Series

The Fibonacci series is defined by:

$f(k+2) = f(k+1) + f(k)$ with $f(1) = f(2) = 1$

The Fibonacci series is therefore $(1, 1, 2, 3, 5, 8, 13, 21, \ldots)$

The Fibonacci series can be calculated using the loop

```
f(1)=1
f(2)=1
do i=3, N
     f(i)=f(i-1)+f(i-2)
enddo
```

How do we do this computation in parallel?

This calculation cannot be made parallel.
- We cannot calculate $f(k+2)$ until we have $f(k+1)$ and $f(k)$

- This is an example of data dependence that results in a non-parallelizable problem

Example: Protein Folding

- Protein folding problems involve a large number of independent calculations that do not depend on data from other calculations
- Concurrent calculations with no dependence on the data from other calculations are termed Embarrassingly Parallel
- These embarrassingly parallel problems are ideal for solution by HPC methods, and can realize nearly ideal concurrency and scalability

Unique Problems Require Unique Solutions

• Each scientific or mathematical problem will, in general, require a unique strategy for efficient parallelization

Thus, each of you may require a different parallel implementation of your numerical problem to achieve good performance.

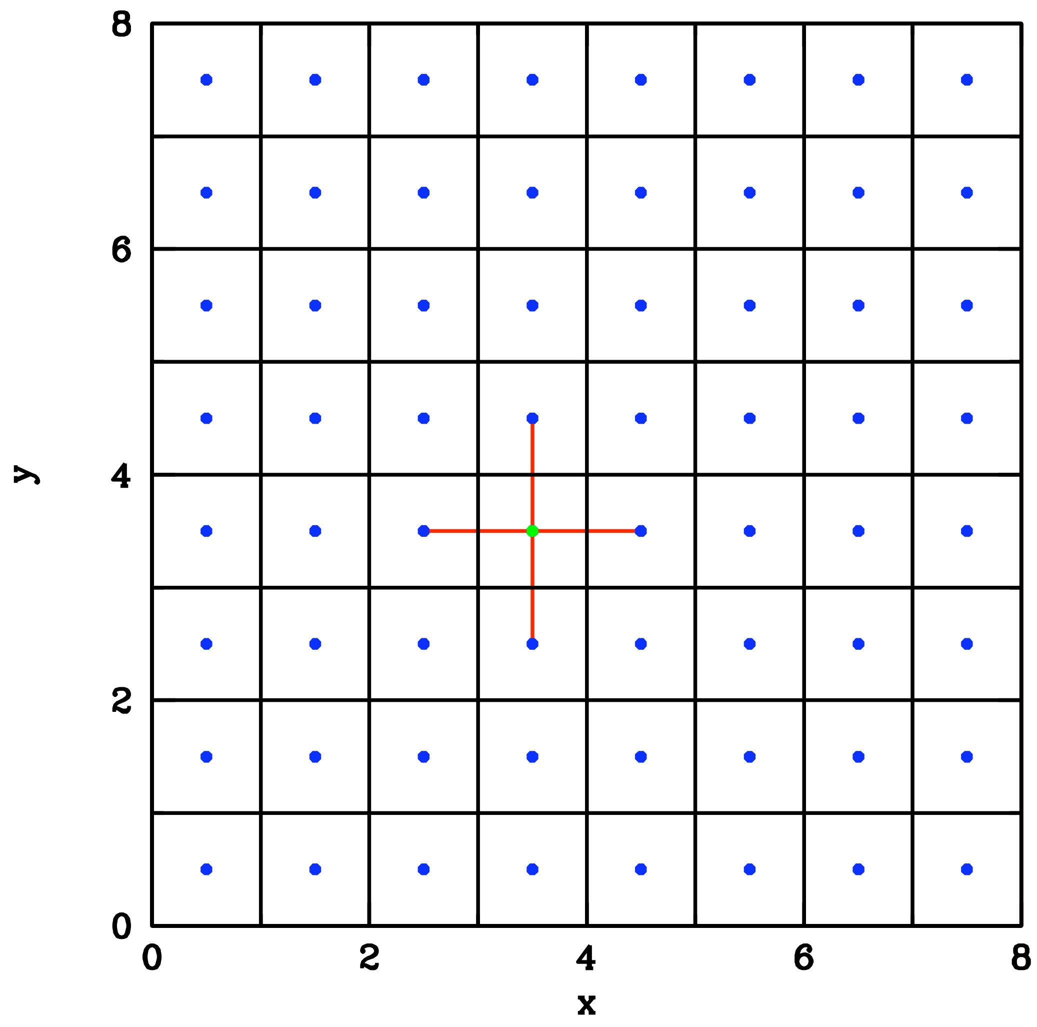
• Flexibility in the way a problem is solved is beneficial to finding a parallel algorithm that yields a good parallel scaling.

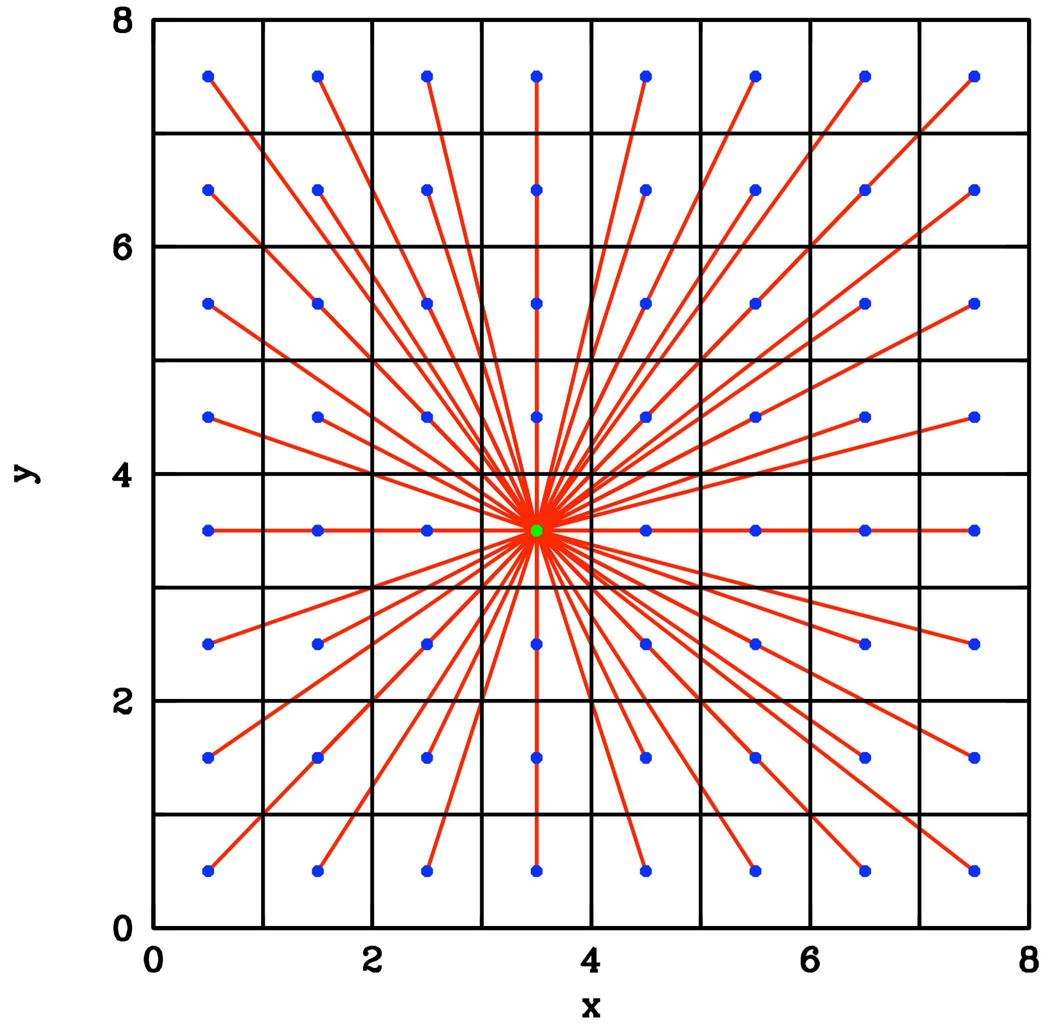
• Often, one has to employ substantial creativity in the way a parallel algorithm is implemented to achieve good scalability.

Understand the Dependencies

- One must understand all aspects of the problem to be solved, in particular the possible dependencies of the data.
- It is important to understand fully all parts of a serial code that you wish to parallelize.

Example: Pressure Forces (Local) vs. Gravitational Forces (Global)





Rule of Thumb

When designing a parallel algorithm, always remember:

Computation is FAST

Communication is SLOW

Input/Output (I/O) is INCREDIBLY SLOW

Other Issues

In addition to concurrency and scalability, there are a number of other important factors in the design of parallel algorithms:

Locality

Granularity

Modularity

Flexibility

Load balancing

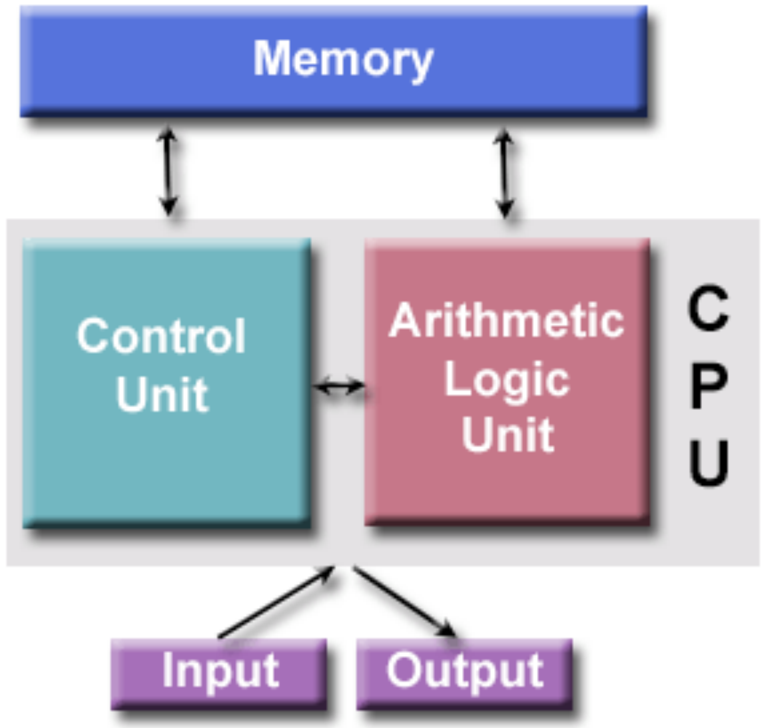
We'll learn about these as we discuss the design of parallel algorithms.

Outline

- Introduction
- Thinking in Parallel
- Parallel Computer Architectures
- Parallel Programming Models
- Design of Parallel Algorithms
- References

The Von Neumann Architecture

Virtually all computers follow this basic design



- Memory stores both instructions and data
- Control unit fetches instructions from memory, decodes instructions, and then sequentially performs operations to perform programmed task
- Arithmetic Unit performs mathematical operations
- Input/Output is interface to the user

Flynn’s Taxonomy

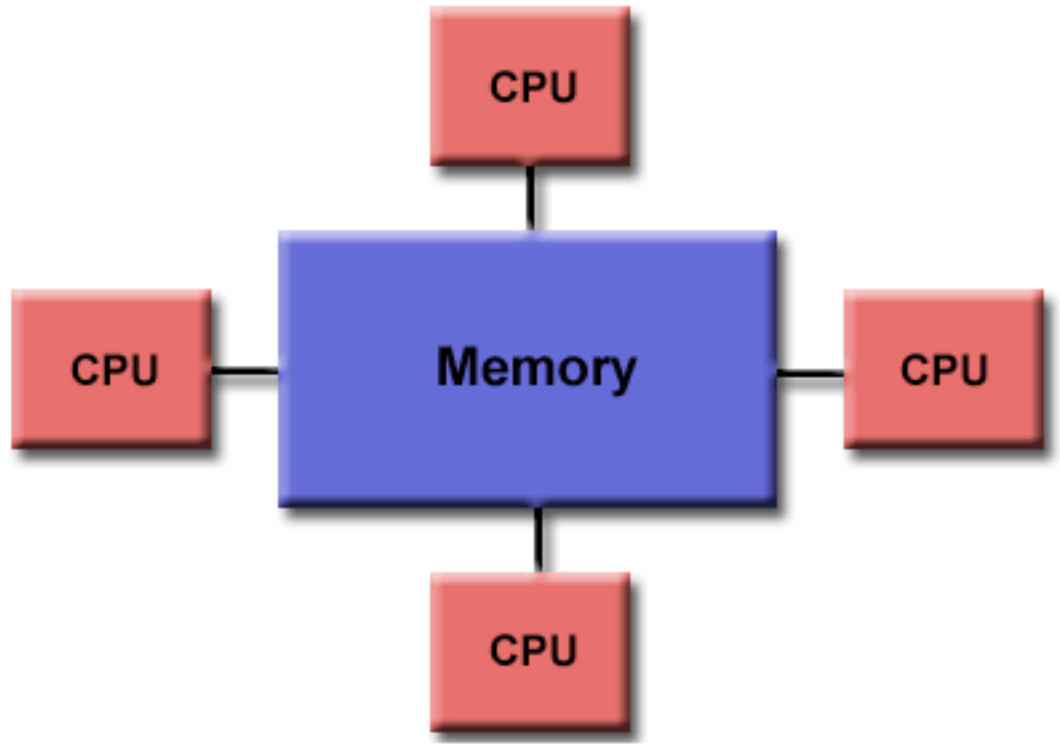
SISD Single Instruction, Single Data	SIMD Single Instruction, Multiple Data
MISD Multiple Instruction, Single Data	MIMD Multiple Instruction, Multiple Data

- SISD: This is a standard serial computer: one set of instructions, one data stream
- SIMD: All units execute same instructions on different data streams (vector)
  - Useful for specialized problems, such as graphics/image processing
  - Old Vector Supercomputers worked this way, also moderns GPUs
- MISD: Single data stream operated on by different sets of instructions, not generally used for parallel computers
- MIMD: Most common parallel computer, each processor can execute different instructions on different data streams
  - Often constructed of many SIMD subcomponents

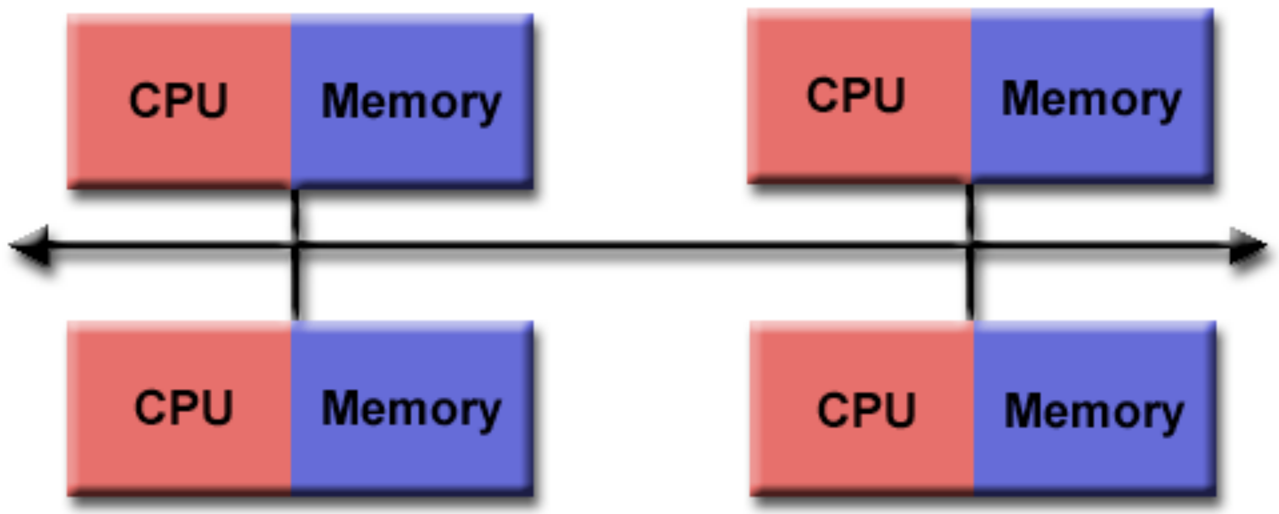
Parallel Computer Memory Architectures

The University of Iowa

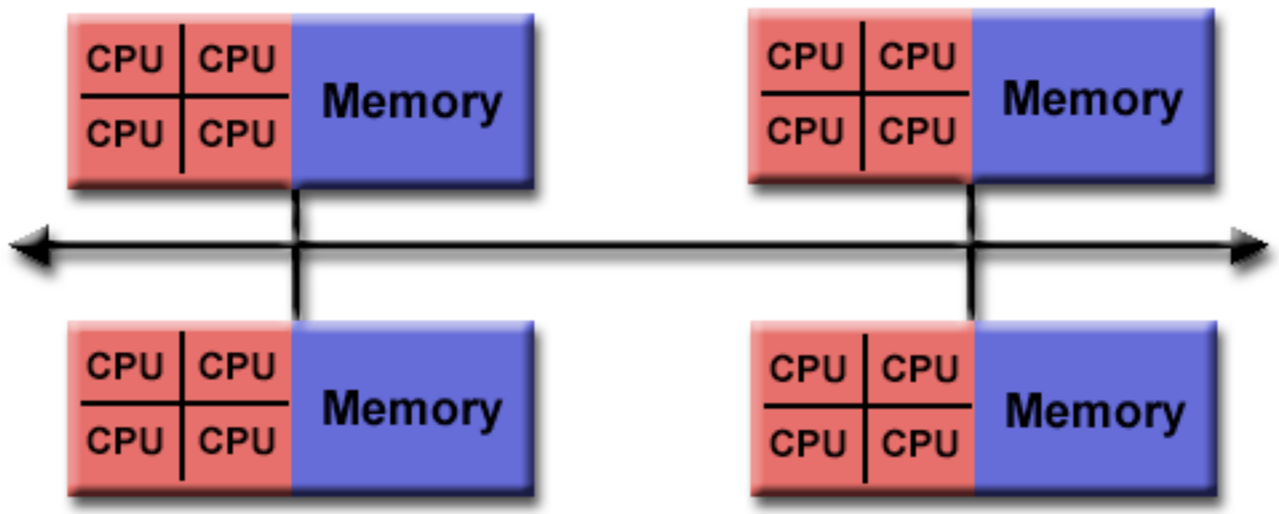
Shared Memory



Distributed Memory



Hybrid Distributed Shared Memory



Relation to Parallel Programming Models

• OpenMP: Multi-threaded calculations occur within shared-memory components of systems, with different threads working on the same data.

• MPI: Based on a distributed-memory model, data associated with another processor must be communicated over the network connection.

• GPUs: Graphics Processing Units (GPUs) incorporate many (hundreds) of computing cores with single Control Unit, so this is a shared-memory model.

• Processors vs. Cores: Most common parallel computer, each processor can execute different instructions on different data streams

-Often constructed of many SIMD subcomponents

Outline

- Introduction
- Thinking in Parallel
- Parallel Computer Architectures
- Parallel Programming Models
- Design of Parallel Algorithms
- References

Parallel Programming Models

- Embarrassingly Parallel
- Master/Slave
- Threads
- Message Passing
- Single Program-Multiple Data (SPMD) vs. Multiple Program-Multiple Data (MPMD)
- Other Parallel Implementations: GPUs and CUDA

Embarrassingly Parallel

- Refers to an approach that involves solving many similar but independent tasks simultaneously

- Little to no coordination (and thus no communication) between tasks

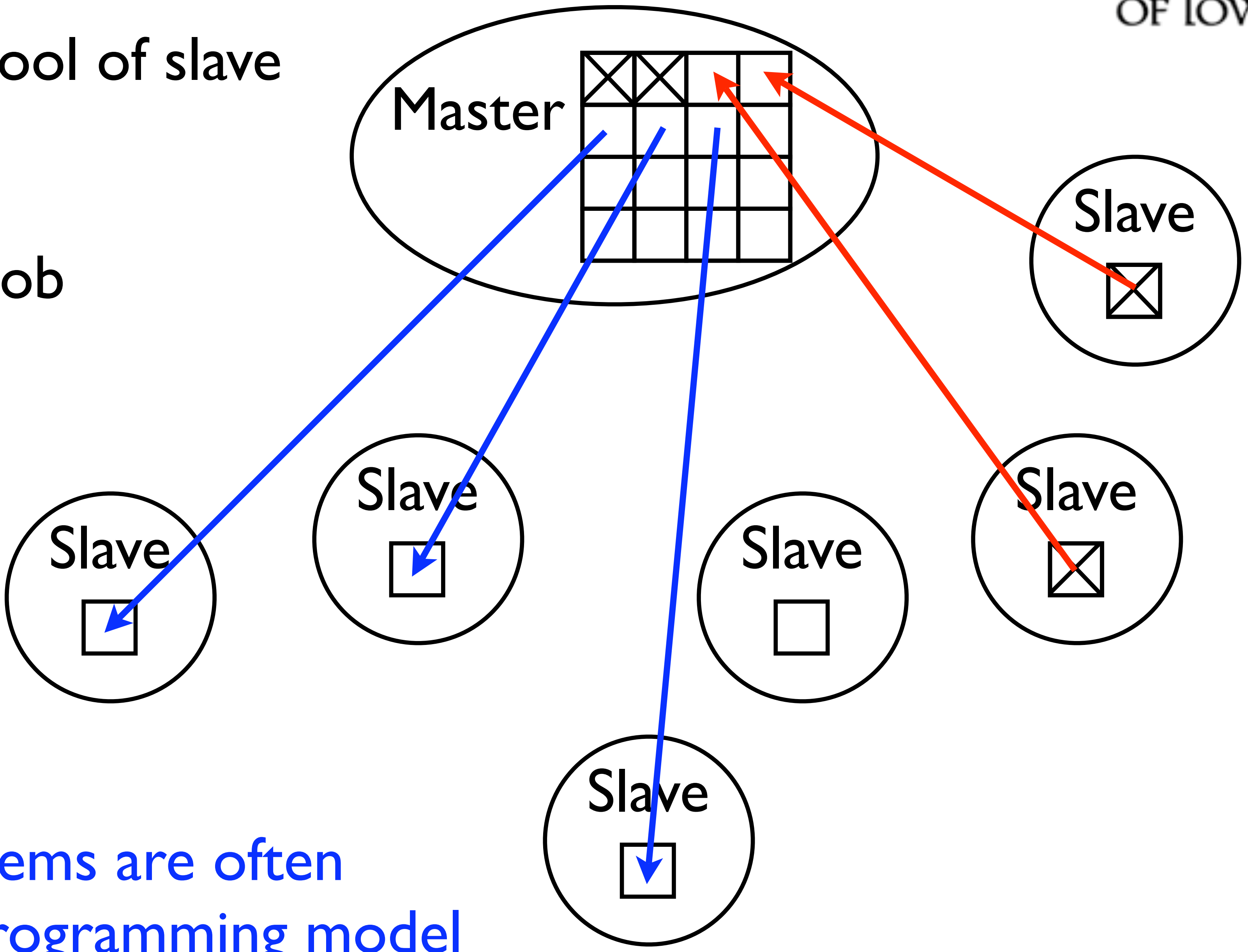
- Each task can be a simple serial program

- This is the "easiest" type of problem to implement in a parallel manner. Essentially requires automatically coordinating many independent calculations and possibly collating the results.

- Examples:
  - Computer Graphics and Image Processing
  - Protein Folding Calculations in Biology
  - Geographic Land Management Simulations in Geography
  - Data Mining in numerous fields
  - Event simulation and reconstruction in Particle Physics

Master/Slave

- Master Task assigns jobs to pool of slave tasks
- Each slave task performs its job independently
- When completed, each slave returns its results to the master, awaiting a new job
- Emabarrasingly parallel problems are often well suited to this parallel programming model



Multi-Threading

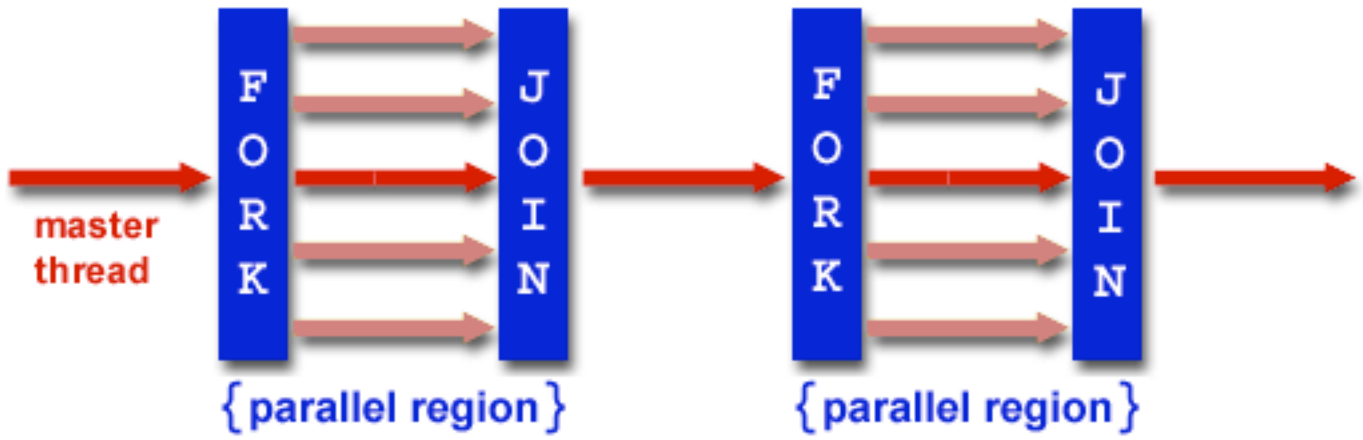
• Threading involves a single process that can have multiple, concurrent execution paths

• Works in a shared memory architecture

• Most common implementation is OpenMP (Open Multi-Processing)

```
serial code
       .
       .
       .

!$OMP PARALLEL DO
do i = 1,N
  A(i)=B(i)+C(i)
enddo
!$OMP END PARALLEL DO
       .
       .
       .
serial code
```



• Relatively easy to make inner loops of a serial code parallel and achieve substantial speedups with modern multi-core processors

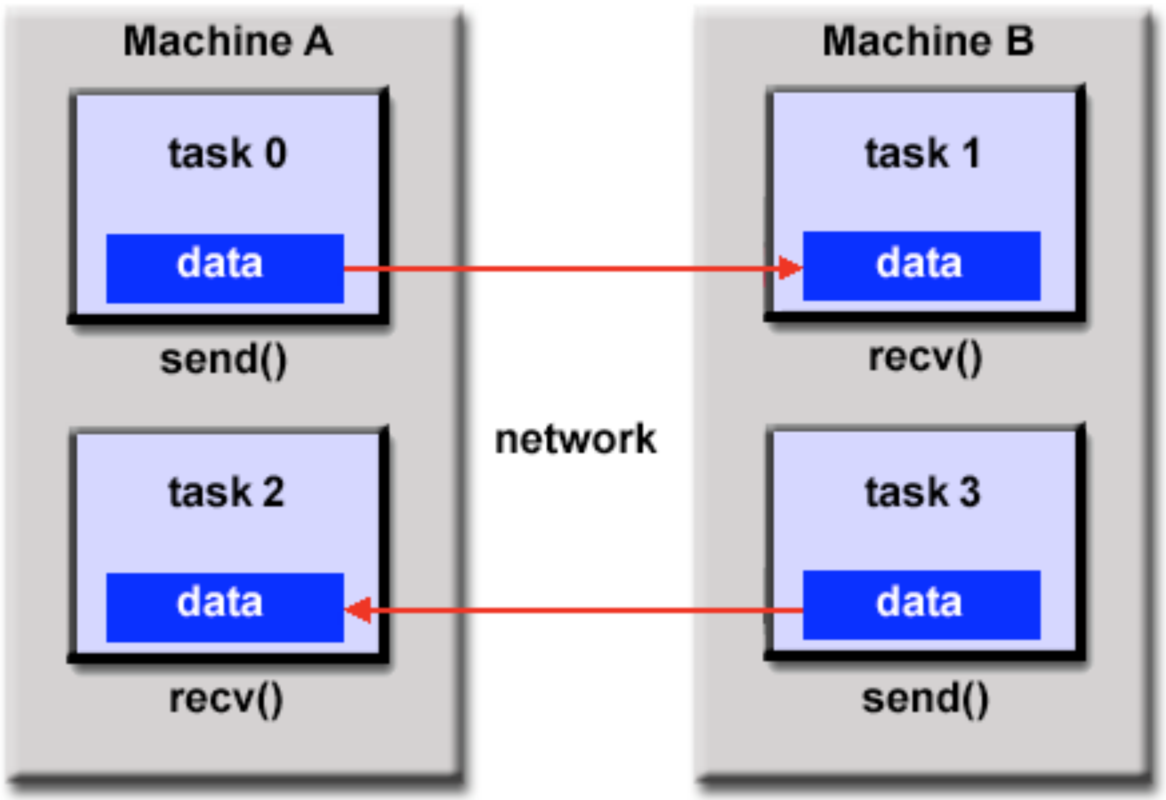
Message Passing

- The most widely used model for parallel programming

- Message Passing Interface (MPI) is the most widely used implementation

- A set of tasks have their own local memory during the computation (distributed-memory, but can also be used on shared-memory machines)

- Tasks exchange data by sending and receiving messages, requires programmer to coordinate explicitly all sends and receives.

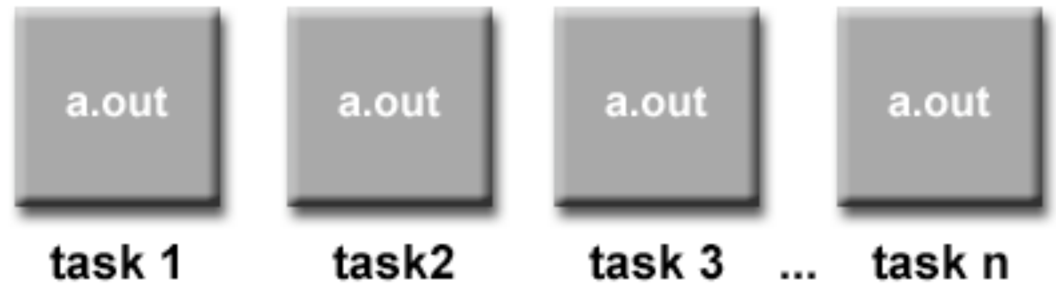


- One aim of this summer school will focus on the use of MPI to write parallel programs.

SPMD vs. MPMD

<u>Single Program-Multiple Data</u> (SPMD)

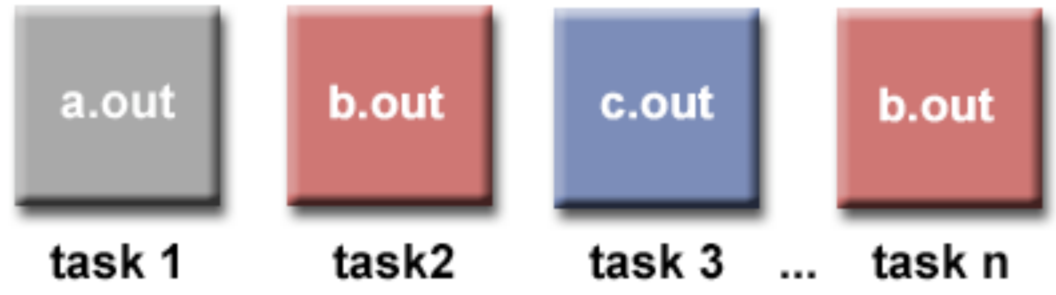
- A single program executes on all tasks simultaneously



- At a single point in time, different tasks may be executing the same or different instructions (logic allows different tasks to execute different parts of the code)

<u>Multiple Program-Multiple Data</u> (MPMD)

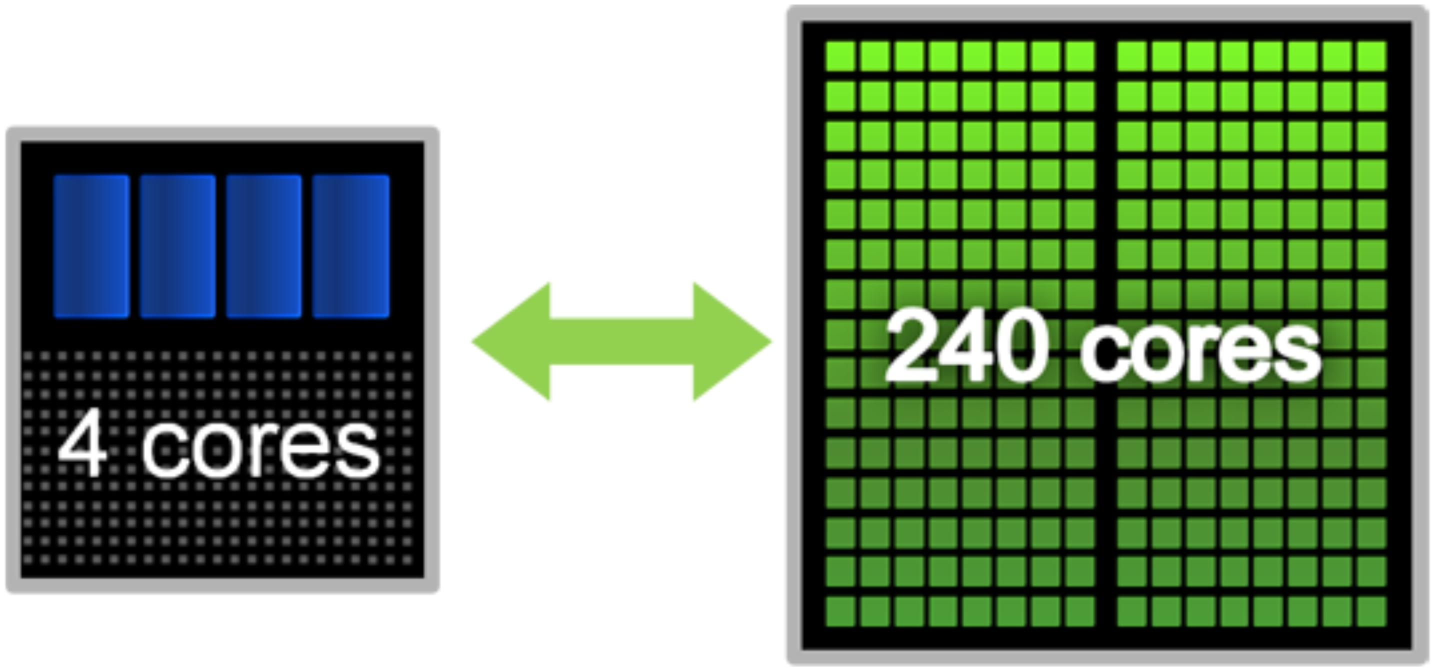
- Each task may be executing the same or different programs than other tasks



- The different executable programs may communicate to transfer data

Other Parallel Programming Models

- GPUs (Graphics Processing Units) contain many (hundreds) of processing cores, allowing for rapid vector processing (Single Instruction, Multiple Data)



• CUDA (Compute Unified Device Architecture) programming allows one to call on this powerful computing engine from codes written in C, Fortran, Python, Java, and Matlab.

• This is an exciting new way to achieve massive computing power for little hardware cost, but memory access bandwidth limitations constrain the possible applications.

Outline

- Introduction
- Thinking in Parallel
- Parallel Computer Architectures
- Parallel Programming Models
- Design of Parallel Algorithms
- References

Design of Parallel Algorithms

- Ensure that you understand fully the problem and/or the serial code that you wish to make parallel

- Identify the program hotspots
  - These are places where most of the computational work is being done
  - Making these sections parallel will lead to the most improvement
  - Profiling can help to determine the hotspots (more on this tomorrow)

- Identify bottlenecks in the program
  - Some sections of the code are disproportionately slow
  - It is often possible to restructure a code to minimize the bottlenecks

- Sometimes it is possible to identify a different computational algorithm that has much better scaling properties

P C A M

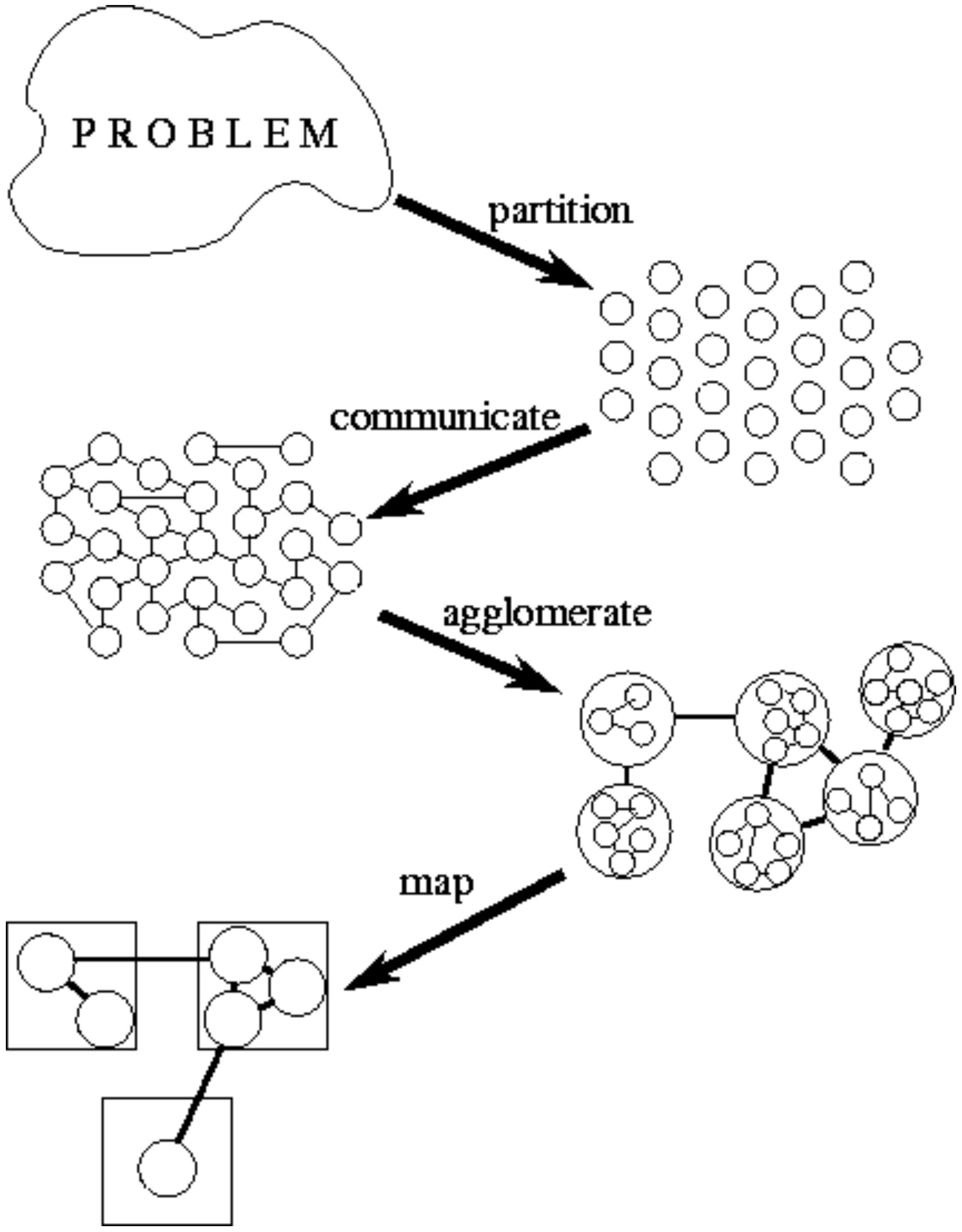
Methodological Approach to Parallel Algorithm Design:

1) Partitioning

2) Communication

3) Agglomeration

4) Mapping



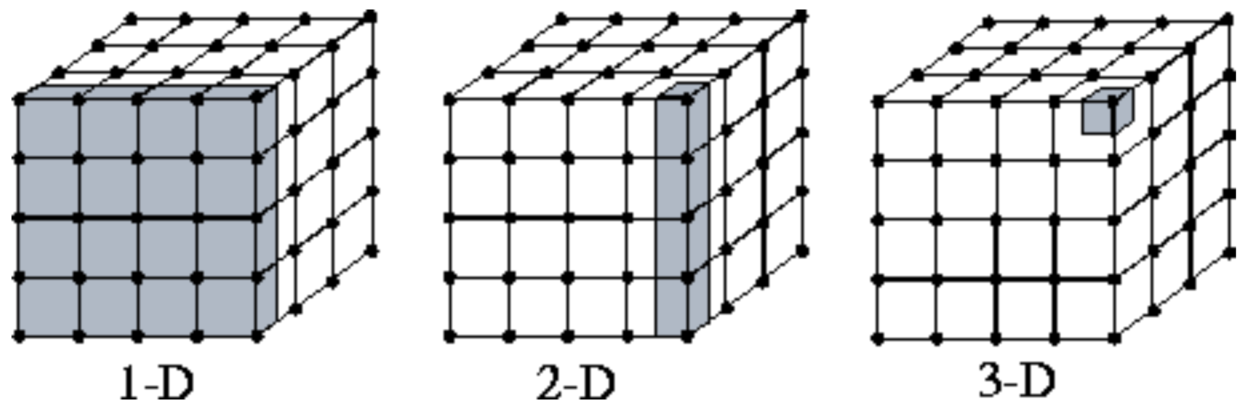
Partitioning

The University of Iowa

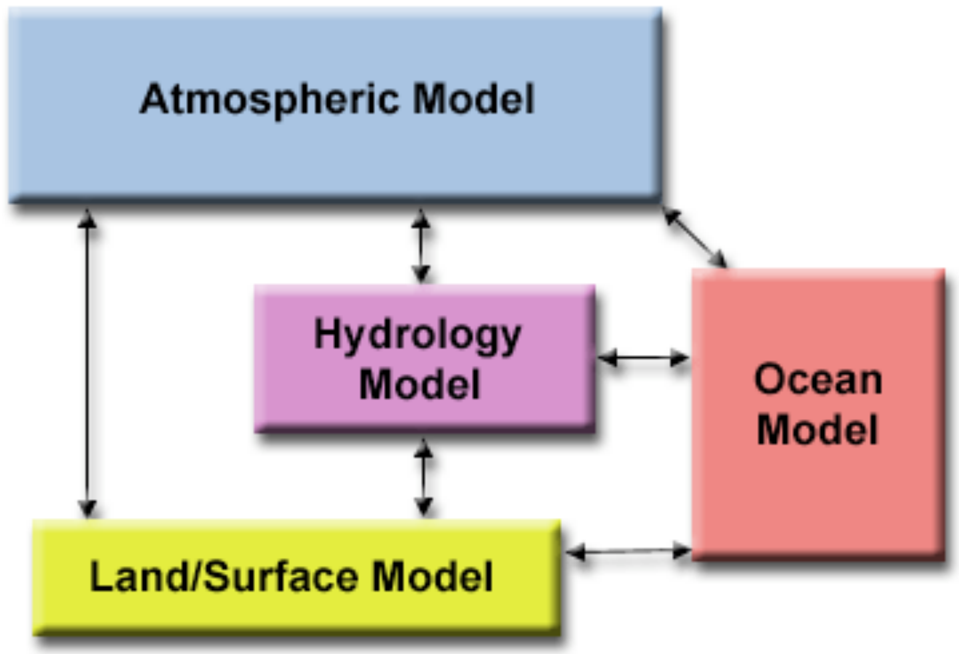
• Split both the computation to be performed and the data into a large number of small tasks (fine-grained)

Two primary ways of decomposing the problem:

• Domain Decomposition



• Functional Decomposition



Communication

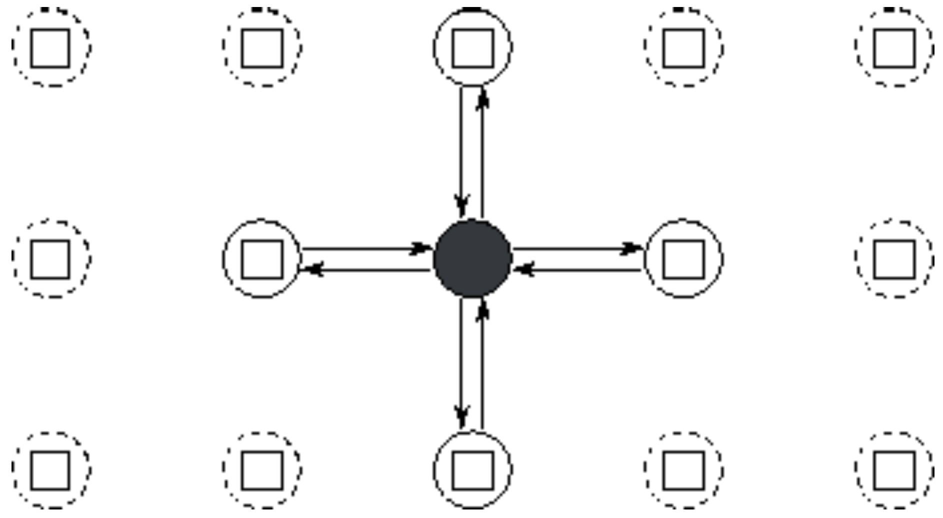
- Identify the necessary communication between the fine-grained tasks to perform the necessary computation

- For functional decomposition, this tasks is often relatively straightforward

- For domain decomposition, this can a challenging task. We'll consider some examples:

Finite Difference Relaxation:

$$f_{i,j}^{t+1} = \frac{4f_{i,j}^t + f_{i-1,j}^t + f_{i+1,j}^t + f_{i,j-1}^t + f_{i,j+1}^t}{8}$$

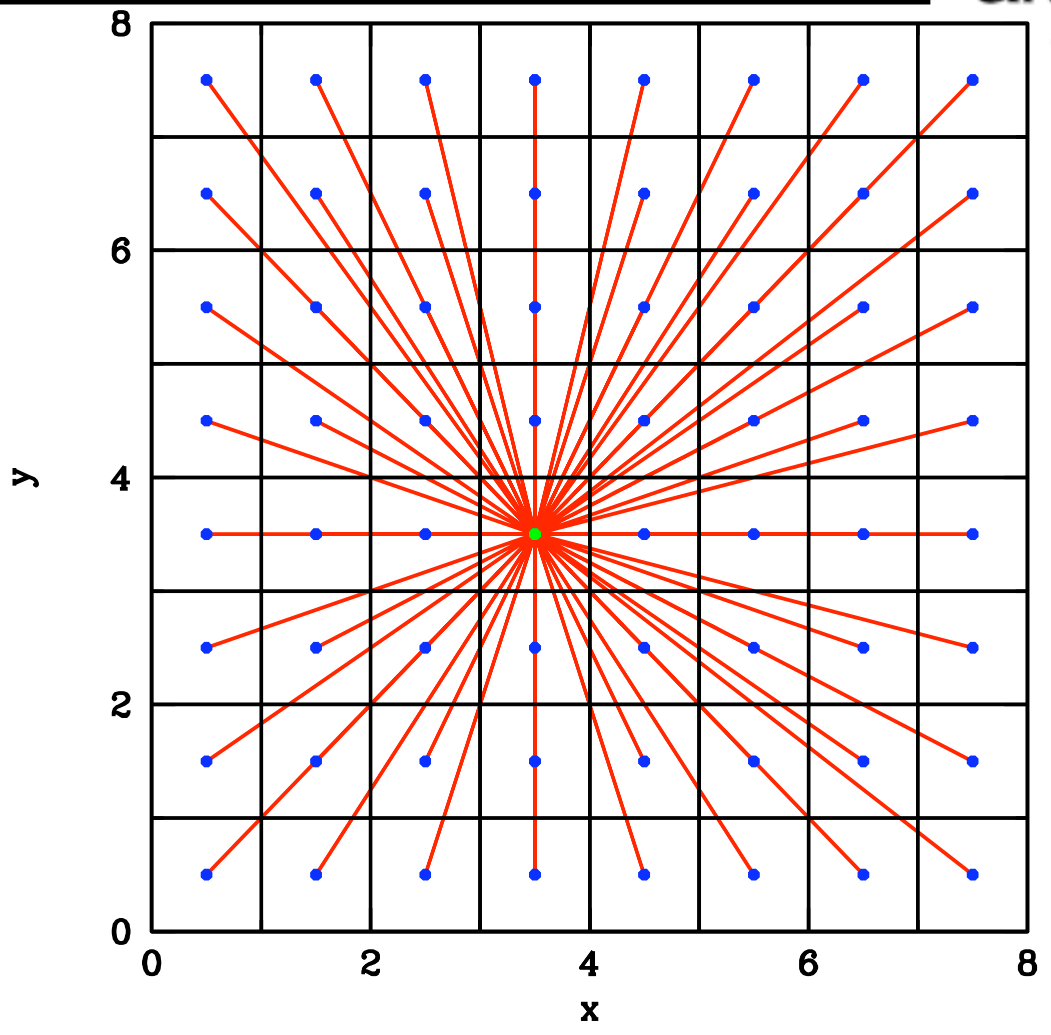


- This is a local communication, involving only neighboring tasks

Communication

Gravitational N-Body Problems:

- This is a global communication, requiring information from all tasks



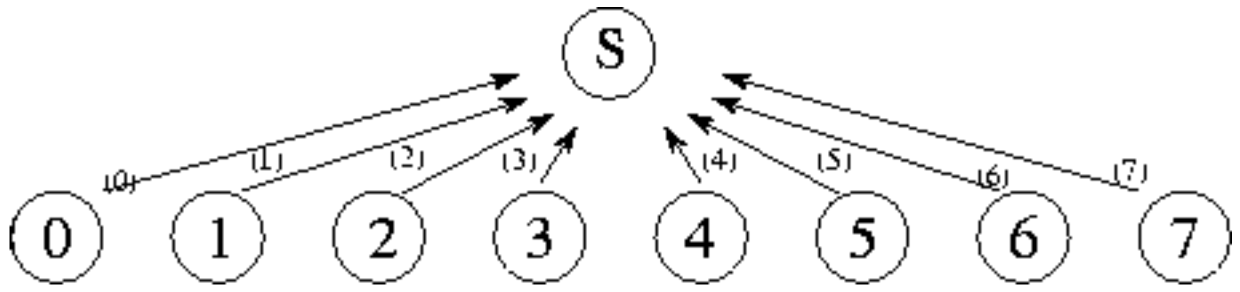
When communication is necessary, it is important to employ a scheme that executes the communications between different tasks concurrently.

Schemes for Global Communication

Consider the problem of summing the values on N=8 different processors

• This is an example of a parallel process generically called reduction.

Method 1: Summing by a Manager task, S



• Requires N=8 communications

• If all processors require the sum, it will require 2N=16 communications

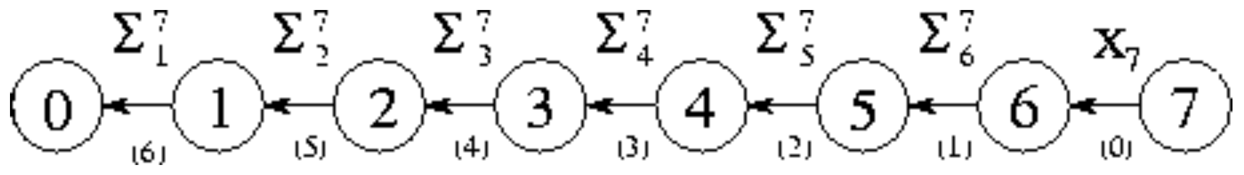
This is a poor parallel algorithm!

• Two properties of this method hinder parallel execution:
  - The algorithm is centralized, the manager participation in all interactions
  - The algorithm is sequential, without communications occurring concurrently

Schemes for Global Communication

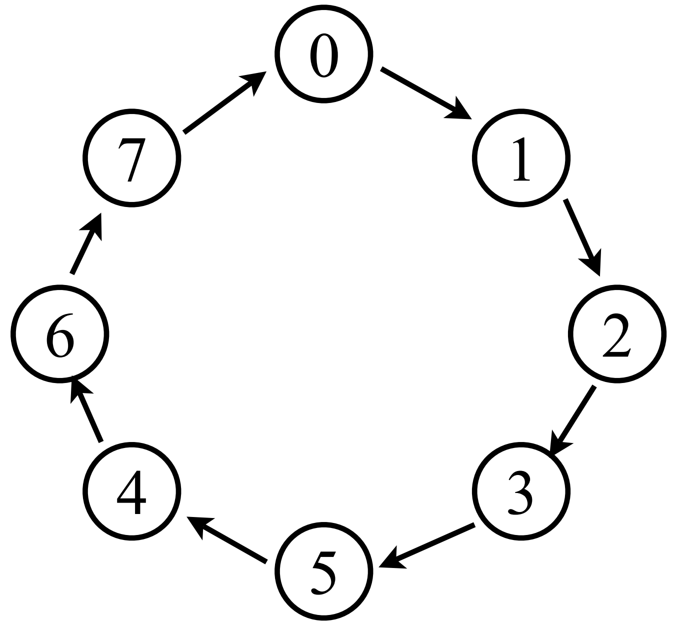
Method II: Line or Ring Communications

- By decentralizing, one can achieve some savings



- Requires N-1=7 communications, but it is still sequential

- If all processors require the sum, we can achieve this result with the same number of concurrent communications
  - By arranging the communcations in a ring, we can distribute the sum at all processors in N-1=7 communication steps.



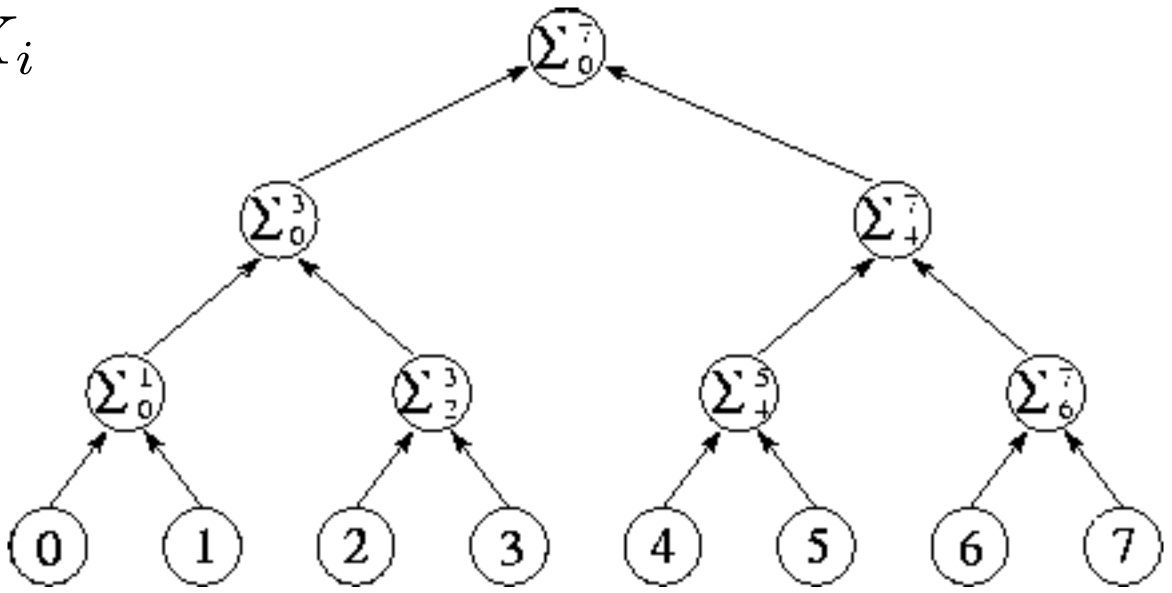
Schemes for Global Communication

THE UNIVERSITY OF IOWA

Method III: Tree Communications

- But we can do better by using a divide and conquer approach to the problem
  -Split problem into two of equivalent size, to be performed concurrently

$$\sum_{i=0}^{N-1} X_i = \sum_{i=0}^{N/2-1} X_i + \sum_{i=N/2}^{N-1} X_i$$



- Recursive application of this principle leads to a tree approach

- Requires $\log_2 N=3$ communication steps

- Distribution of the sum to all processors can be accomplished with the same $\log_2 N=3$ communication steps.

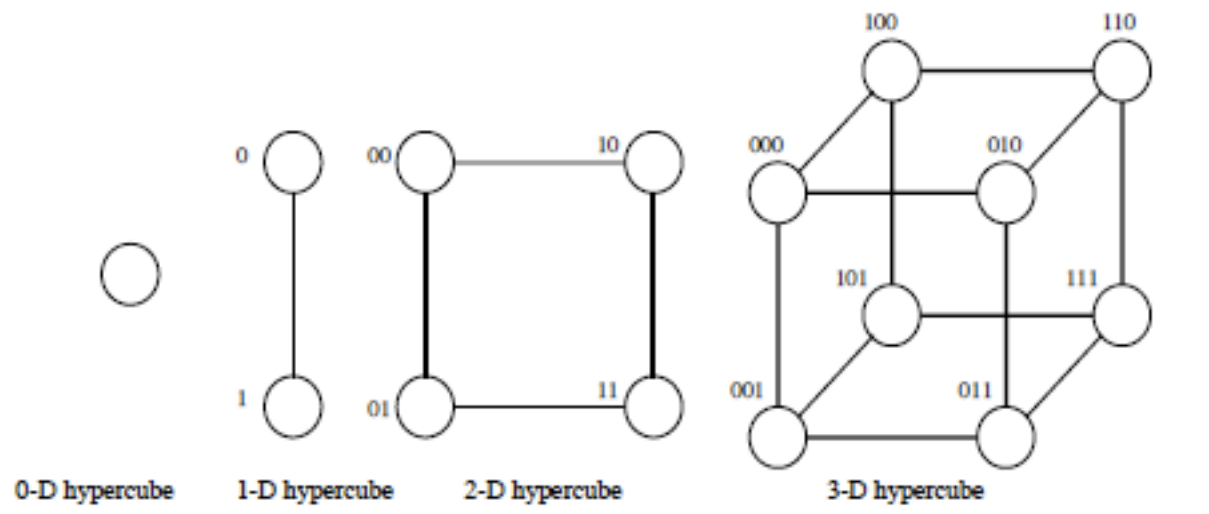
This is called a hypercube communication scheme

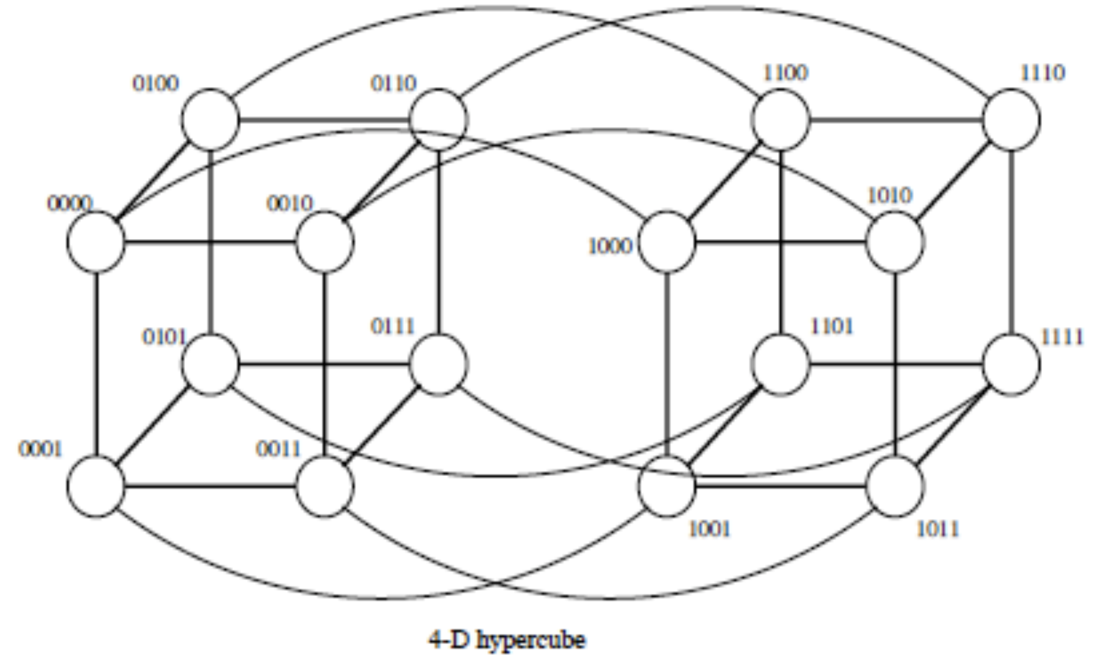
Hypercube Communication

In Hypercube Communications,

-All tasks communicate with one other tasks at each step,

-At each step, the task passes along all of the information it has gathered up to that point





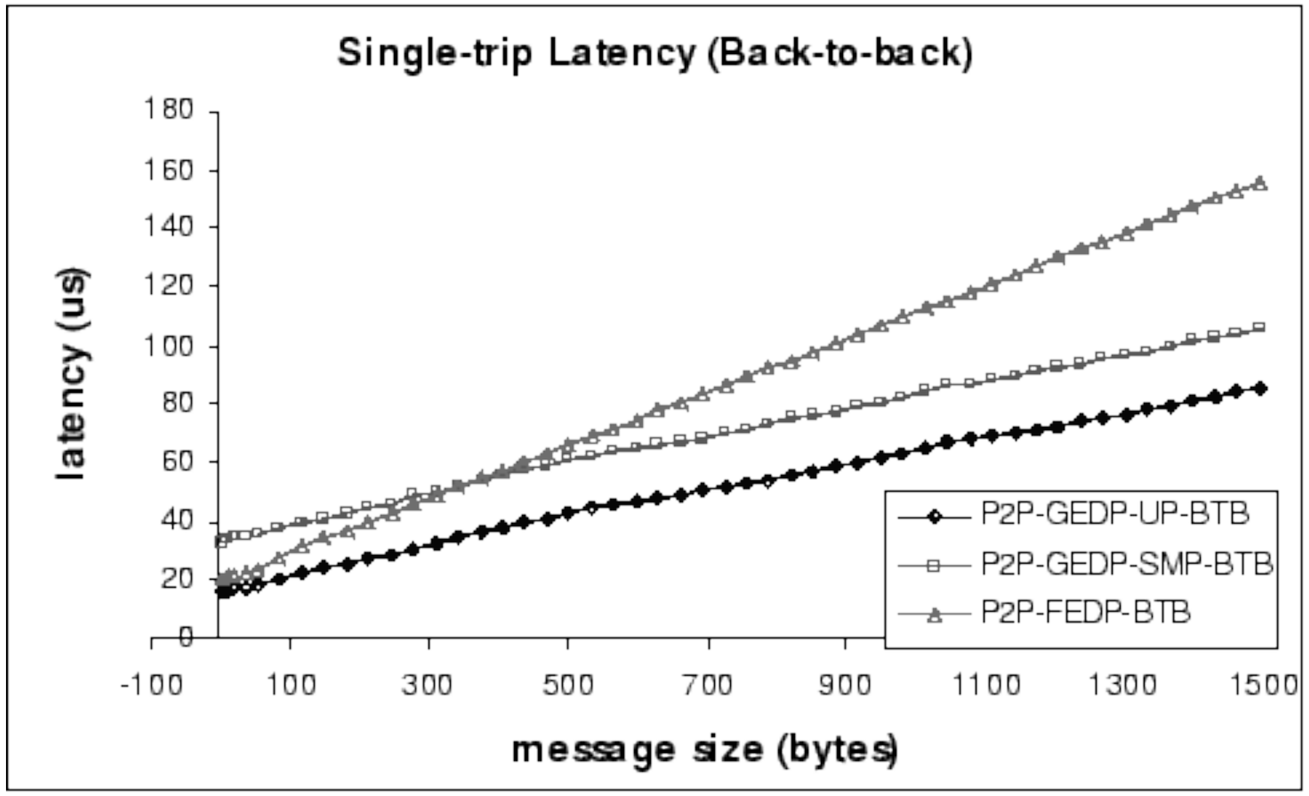
Communication: Latency vs. Bandwidth

The University of Iowa

Cost of Communications (Overhead):

- Latency: The time it takes to send a minimal message (1 bit) from A to B

- Bandwidth: The amount of data that can be communicated per unit of time



Factors to consider:

- Sending many small messages will cause latency to dominate the communications overhead
  - It is better to package many small messages into one large message

- The less information that needs to be transmitted, the less time the communications will require.

- It is often best to have all necessary communication occur at the same time

Synchronous vs. Asynchronous Communication

Consider a communication involving a message sent from task A to task B

Synchronous Communication:

• Task A sends the message, and must wait until task B receives message to move on

• Also known as blocking communication

Asynchronous Communication:

• After task A has sent the message, it can move on to do other work. When task B receives the message doesn't matter to task A.

• Also known as non-blocking communication

• Requires care to insure that different tasks don't get wildly out of step, possibly leading to race conditions or deadlocks.

P C A M

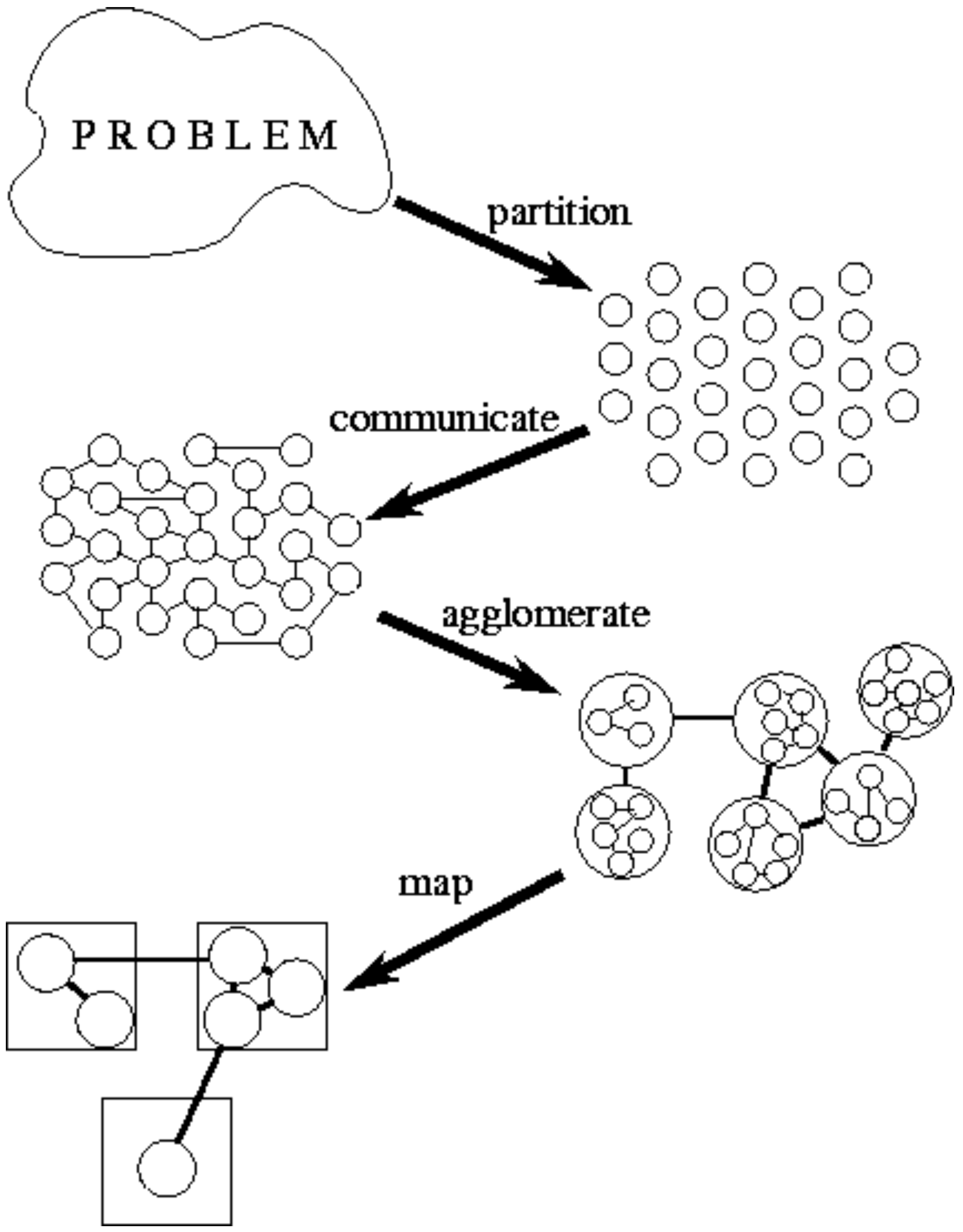
Methodological Approach to Parallel Algorithm Design:

1) Partitioning

2) Communication

3) Agglomeration

4) Mapping



Agglomeration

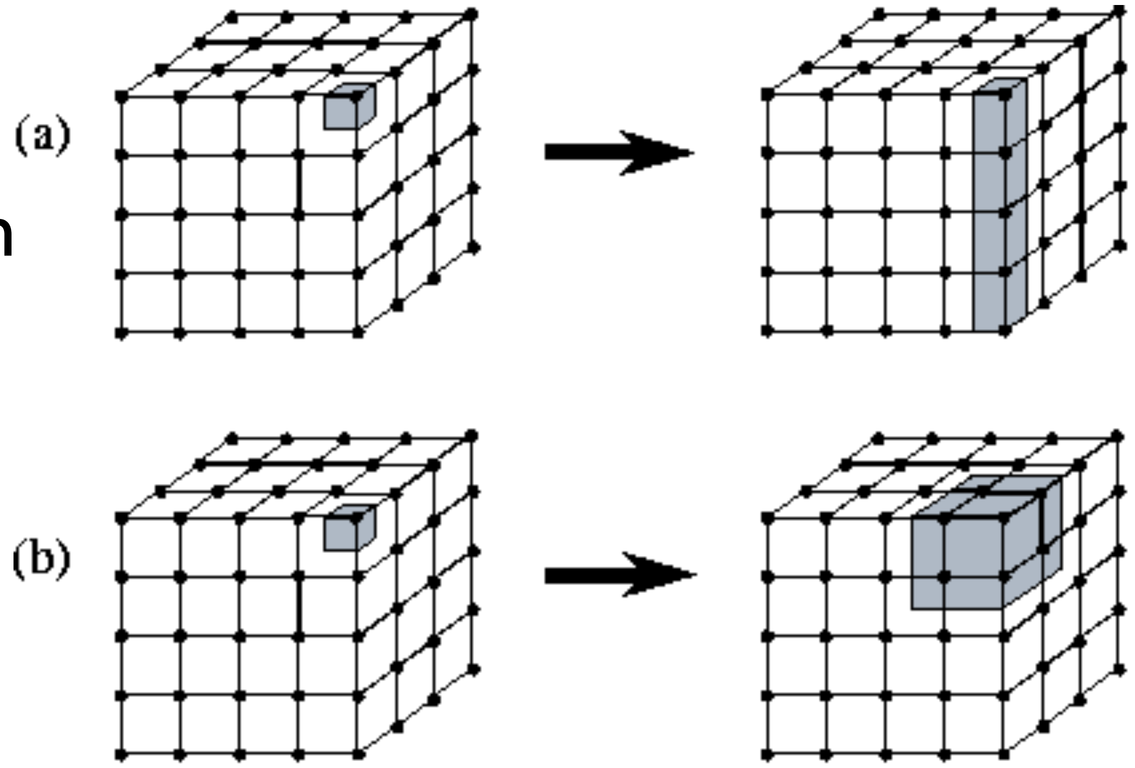
- Fine-grained partitioning of a problem is generally not an efficient parallel design
  - Requires too much communication of data to be efficient

- Agglomeration is required to achieve data locality and good performance

Agglomeration:

- Combine the many fine-grained tasks from partitioning into fewer coarse-grained tasks of larger size

- This task must take into account the details of the problem in order to achieve an algorithm with good scaling properties and good efficiency



Granularity

Granularity is the ratio of local computation to communication.

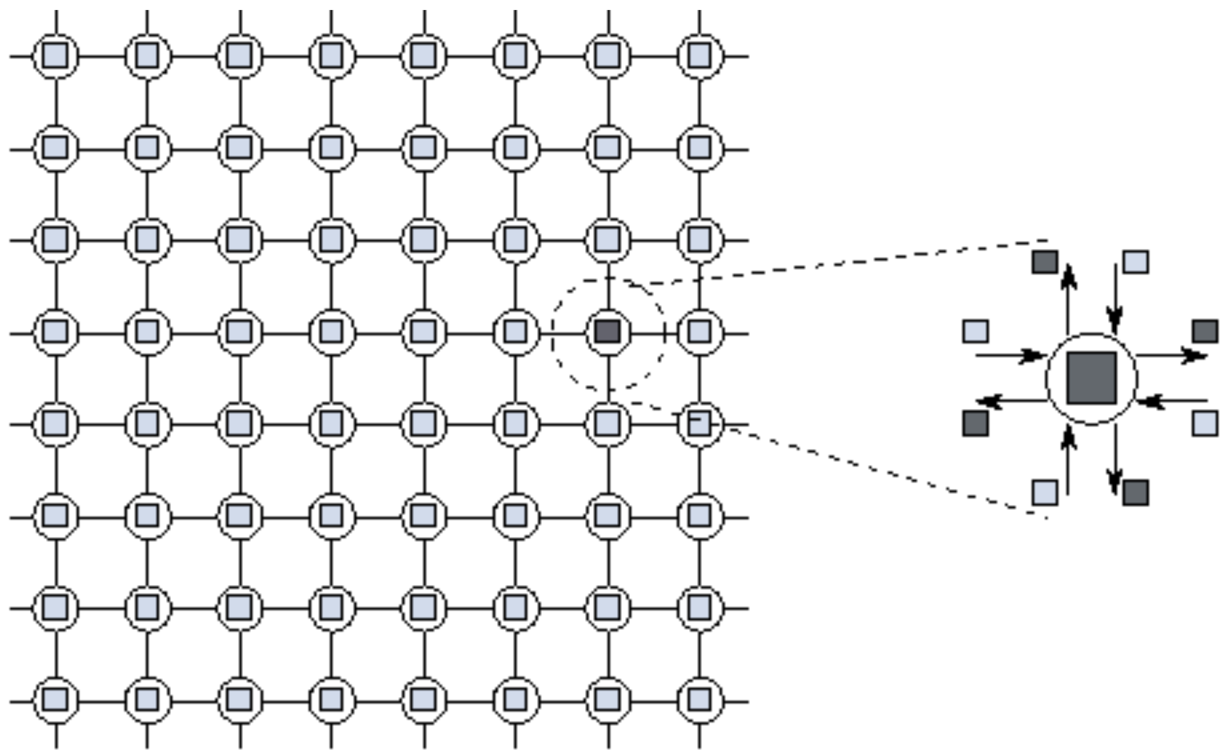
- Agglomeration is used to increase the granularity, improving performance since communication is slow compared to computation.

- By combining many finely grained tasks, we reduce both:
  (i) number of communications
  (ii) size of communications

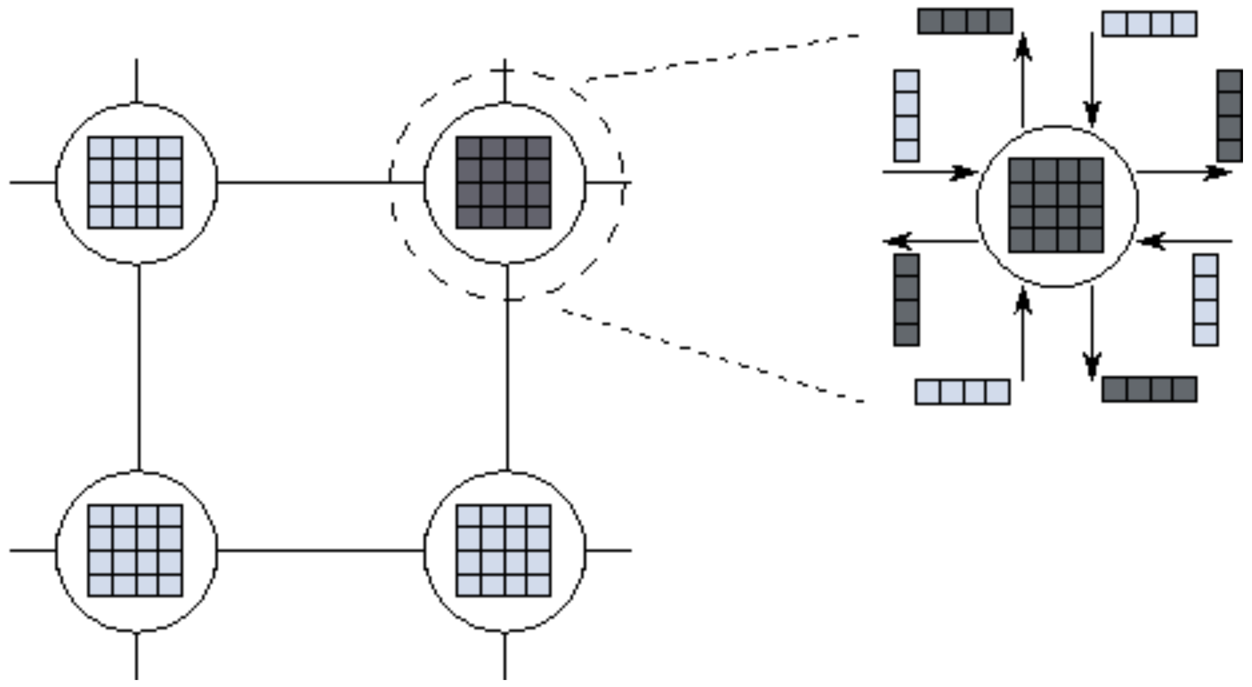
- In (a), updating 16 points requires
  (i) 16x4=64 communications
  (ii) passing 64 "bits"

- In (b), updating 16 points requires
  (i) 4 communications
  (ii) passing 16 "bits"

(a)



(b)



Surface-to-Volume in Domain Decomposition

For domain decomposition in problems with local data dependency, (ex. finite difference):

- Communication is proportional to subdomain surface area
- Computation is proportional to volume of the subdomain

For this 2-D problem:

(a) Surface $S = 4d$ & Area $A = d^2$

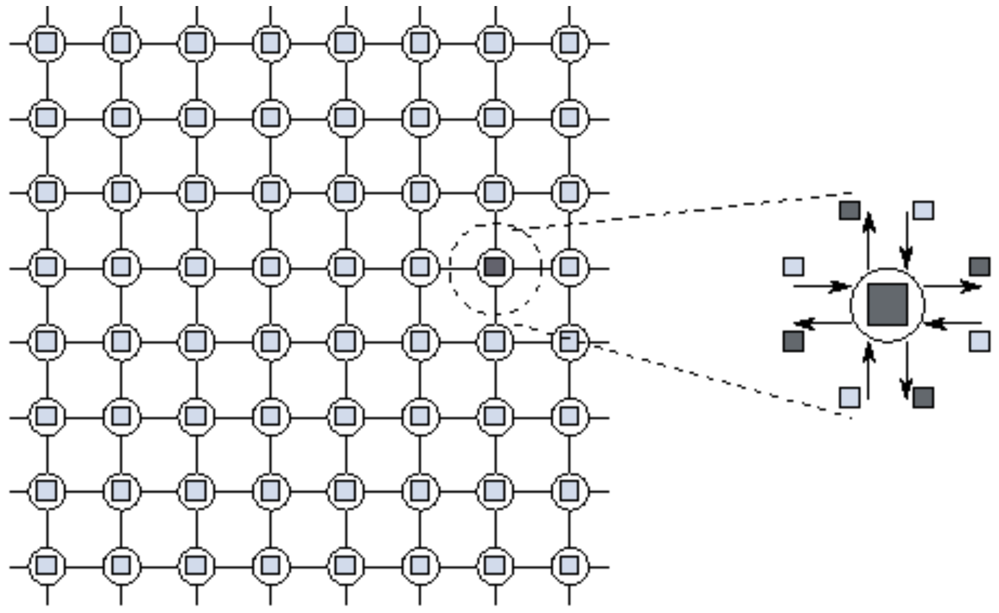
Thus, $$\frac{S}{A} = \frac{4}{d}$$

(b) Surface $S = 16d$ & Area $A = 16d^2$

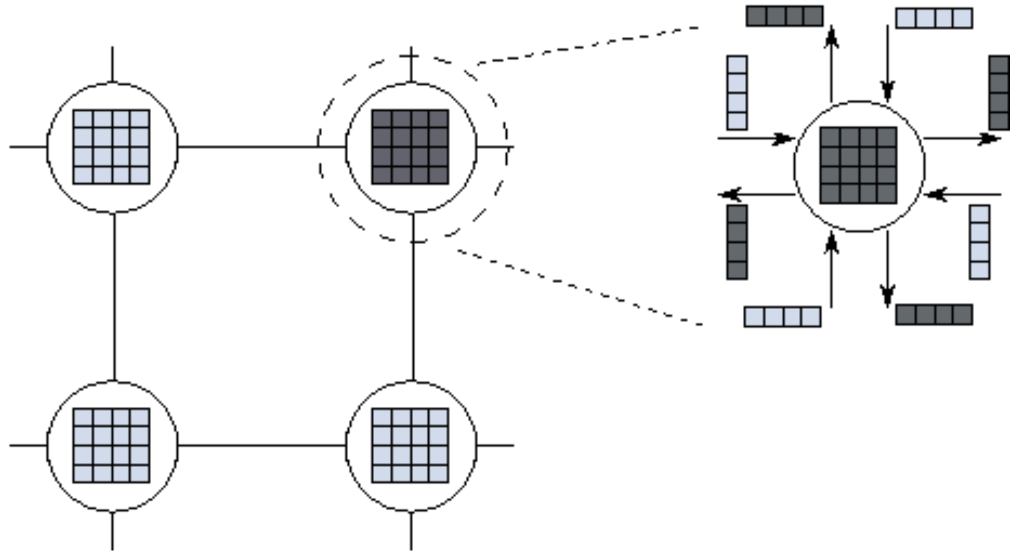
Thus, $$\frac{S}{A} = \frac{1}{d}$$

Decrease of surface-to-volume ratio is equivalent to increased granularity

(a)



(b)

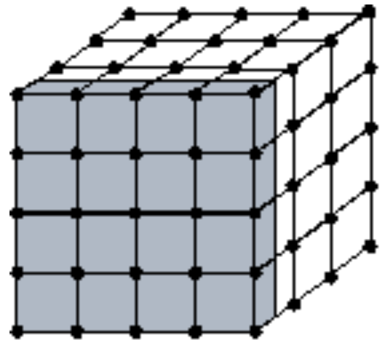


Other Factors in Agglomeration

Maintaining flexibility:

• It is possible to make choices in designing a parallel algorithm that limit flexibility

• For example, if 3-D data is decomposed in only 1-D, it will limit the scalability of the application


1-D

We'll see this later in the weak scaling example of HYDRO

Replication of Data and/or Computation:

• Sometimes significant savings in communication can be made by replicating either data or computation

• Although from a serial point of view this seems inefficient and wasteful, because communication is much slower than computation, it can often lead to significant improvements in performance.

P C A M

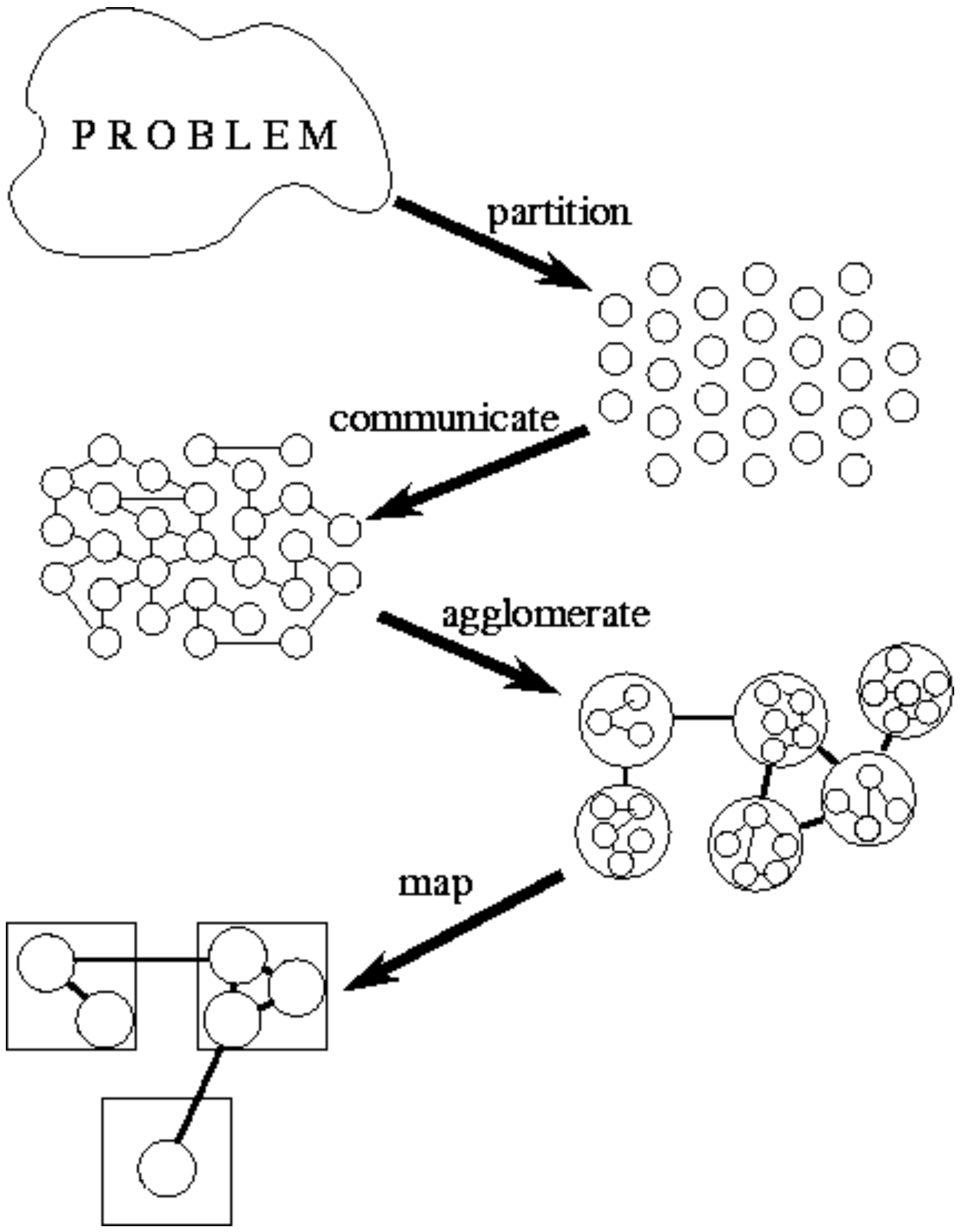
Methodological Approach to Parallel Algorithm Design:

1) Partitioning

2) Communication

3) Agglomeration

4) Mapping

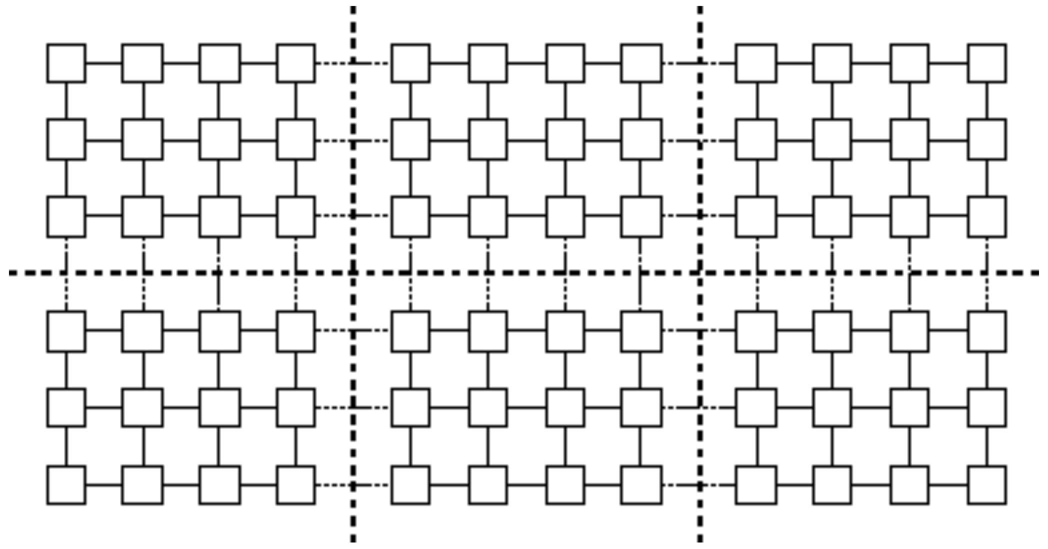


Mapping

Mapping Coarse-grained Tasks to Processors:

- Goal: To minimize total execution time

- Guidelines:
  - Tasks that can execute concurrently map to different processors
  - Tasks that communicate frequently map to the same processor

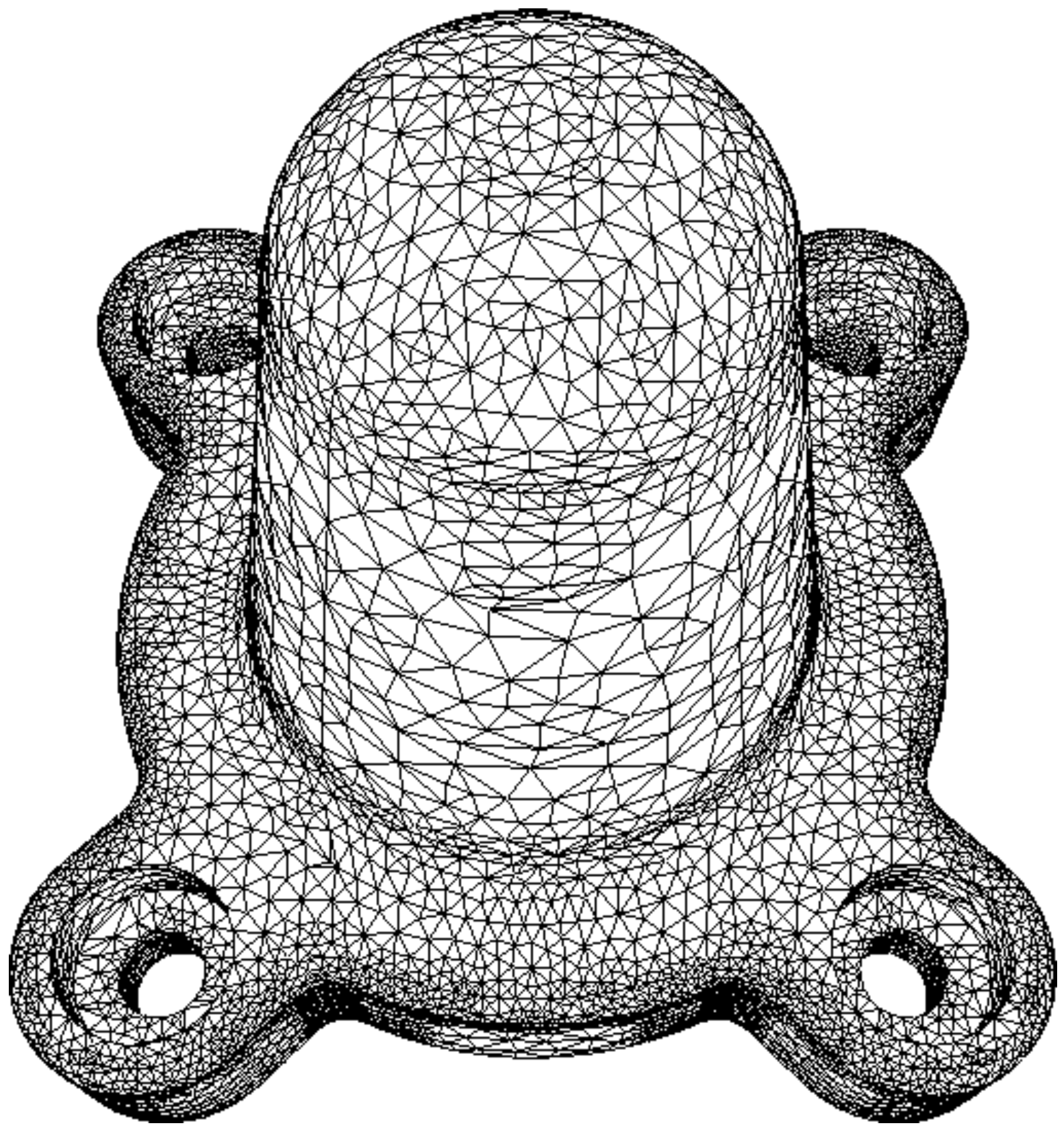


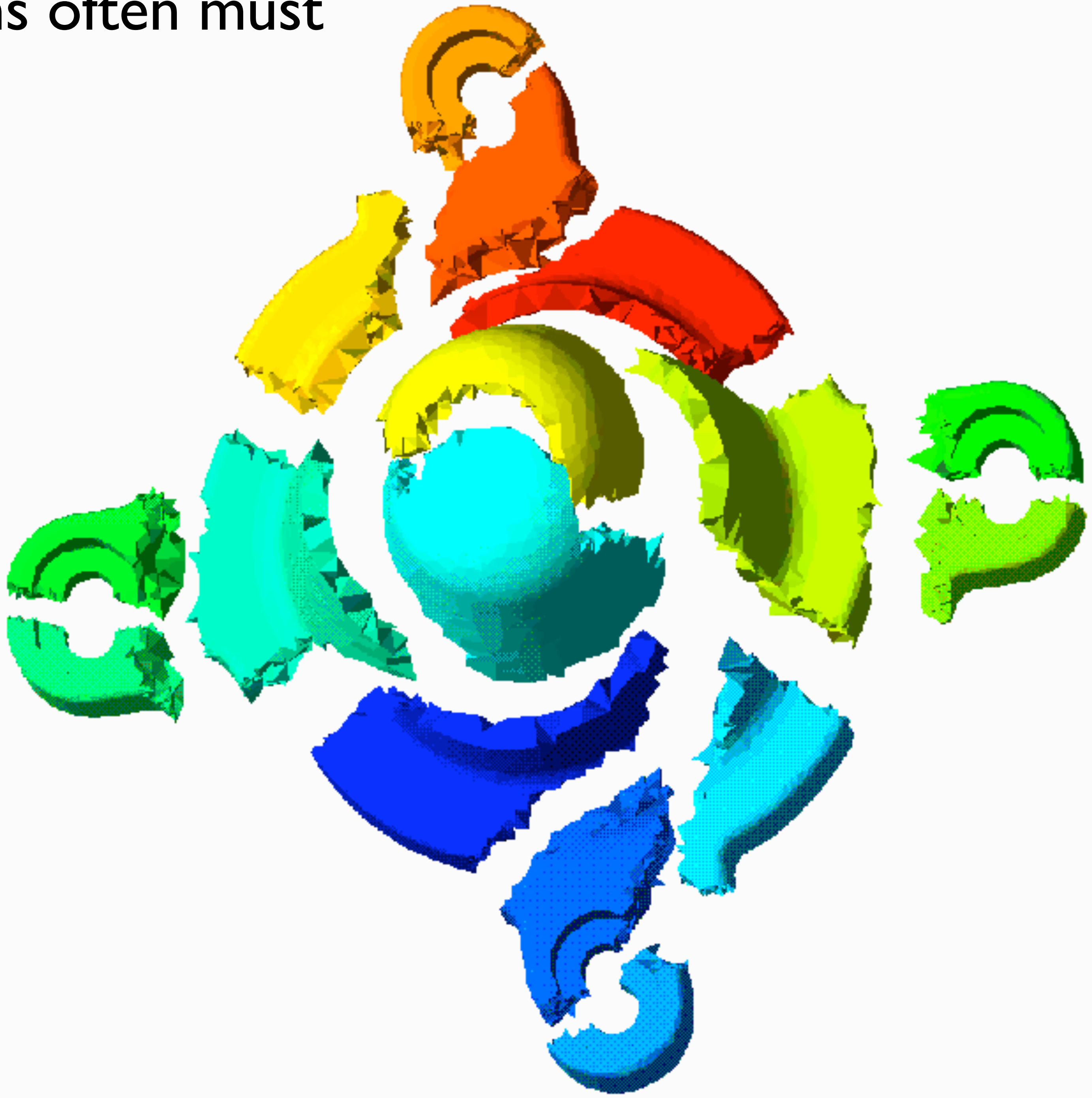
- For many domain decomposition approaches, the agglomeration stage decreases the number of coarse-grained tasks to exactly the number of processors, and the job is done

- In general, however, one wants to map tasks to achieve good load balancing

Load Balancing

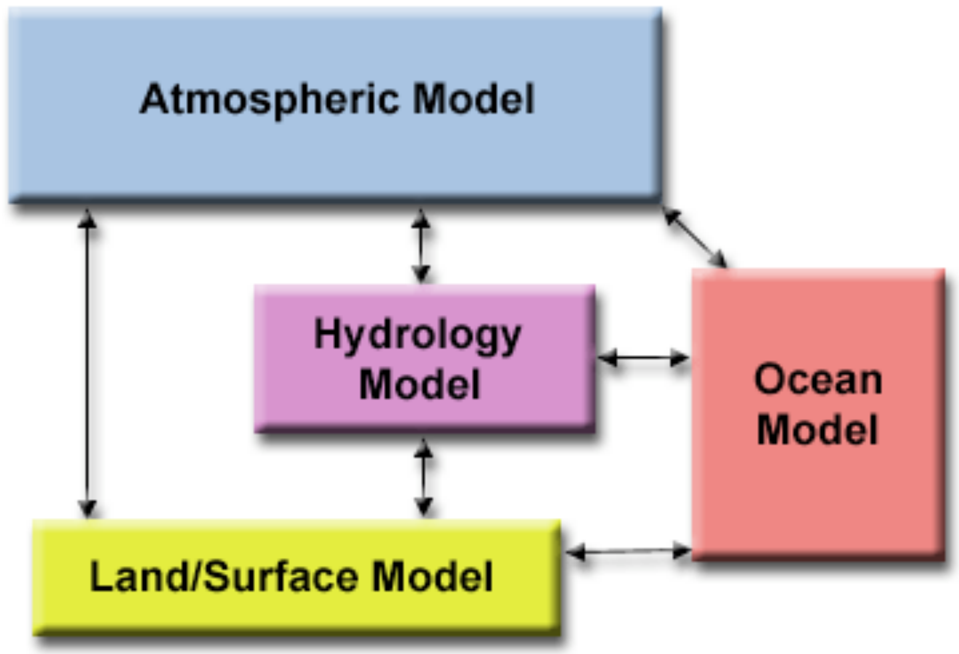
- Good parallel scaling and efficiency requires that all processors have an equal amount of work
- Otherwise, some processors will sit around idle, while others are completing their work, leading to a less efficient computation
- Complicated Load Balancing algorithms often must be employed.

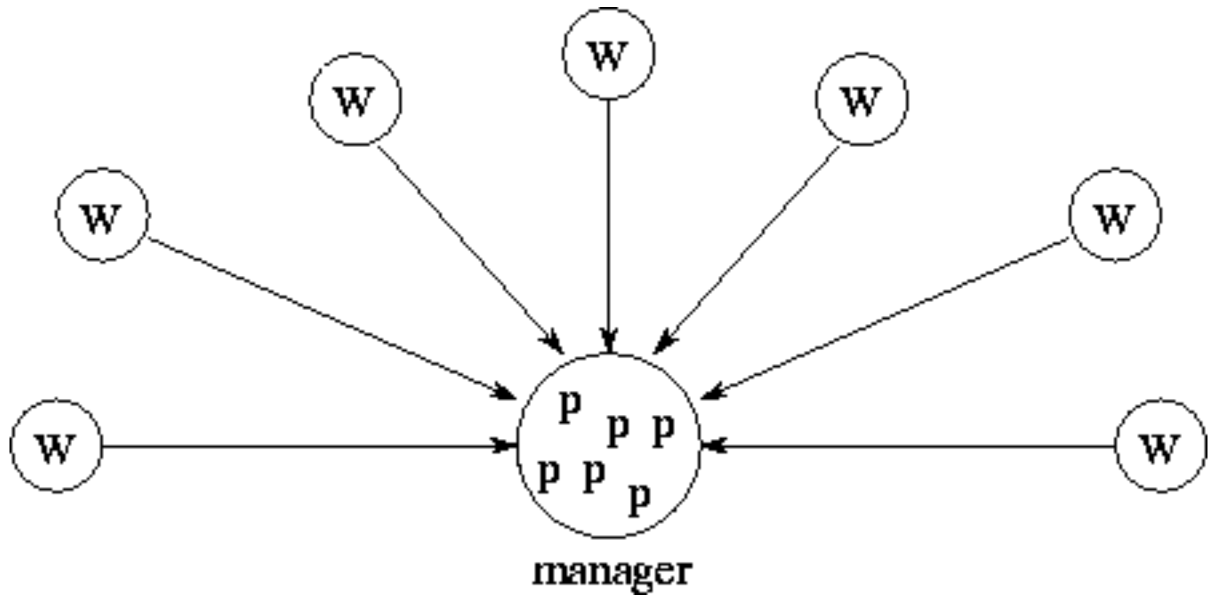




Load Balancing

- For problems involving functional decomposition or a master/slave design, load balancing can be a very significant challange





Parting Thoughts

- Part of the challenge of parallel computing is that the most efficient parallelization strategy for each problem generally requires a unique solution.

- It is generally worthwhile spending significant time considering alternative algorithms to find an optimal one, rather than just implementing the first thing that comes to mind

- But, consider the time required to code a given parallel implementation
  - You can use a less efficient method if the implementation is much easier.
  - You can always improve the parallelization scheme later. Just focus on making the code parallel first.

TIME is the ultimate factor is choosing a parallelization strategy---Your Time!

References

Introductory Information on Parallel Computing

- Designing and Building Parallel Programs, Ian Foster
  http://www.mcs.anl.gov/~itf/dbpp/
  -Somewhat dated (1995), but an excellent online textbook with detailed discussion about many aspects of HPC. This presentation borrowed heavily from this reference
- Introduction to Parallel Computing, Blaise Barney
  https://computing.llnl.gov/tutorials/parallel_comp/
  -Up to date introduction to parallel computing with excellent links to further information
- MPICH2: Message Passage Inteface (MPI) Implementation
  http://www.mcs.anl.gov/research/projects/mpich2/
  -The most widely used Message Passage Interface (MPI) Implementation
- OpenMP
  http://openmp.org/wp/
  -Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran
- Numerical Recipes
  http://www.nr.com/
  -Incredibly useful reference for a wide range of numerical methods, though not focused on parallel algorithms.
- The Top 500 Computers in the World
  http://www.top500.org/
  -Updated semi-annually list of the Top 500 Supercomputers

References

Introductory Information on Parallel Computing

- Message Passing Interface (MPI), Blaise Barney
  https://computing.llnl.gov/tutorials/mpi/
  -Excellent tutorial on the use of MPI, with both Fortran and C example code

- OpenMP, Blaise Barney
  https://computing.llnl.gov/tutorials/openMP/
  -Excellent tutorial on the use of OpenMP, with both Fortran and C example code

- High Performance Computing Training Materials, Lawrence Livermore National Lab
  https://computing.llnl.gov/?set=training&page=index
  -An excellent online set of webpages with detailed tutorials on many aspects of high performance computing.