Overview of High Performance Computing

Timothy H. Kaiser, PH.D.
tkaiser@mines.edu

http://geco.mines.edu/workshop

This tutorial will cover all three time slots.

In the first session we will discuss the importance of parallel computing to high performance computing. We will by example, show the basic concepts of parallel computing. The advantages and disadvantages of parallel computing will be discussed. We will present an overview of current and future trends in HPC hardware.

The second session will provide an introduction to MPI, the most common package used to write parallel programs for HPC platforms. As tradition dictates, we will show how to write "Hello World" in MPI. Attendees will be shown how to and allowed to build and run relatively simple examples on a consortium resource.

The third session will briefly discuss other important HPC topics. This will include a discussion of OpenMP, hybrid programming, combining MPI and OpenMP. Some computational libraries available for HPC will be highlighted. We will briefly mention parallel computing using graphic processing units (GPUs).

Today's Overview

- HPC computing in a nutshell?
- Basic MPI - run an example
- A few additional MPI features
- A "Real" MPI example
- Scripting
- OpenMP
- Libraries and other stuff

Introduction

- What is parallel computing?
- Why go parallel?
- When do you go parallel?
- What are some limits of parallel computing?
- Types of parallel computers
- Some terminology

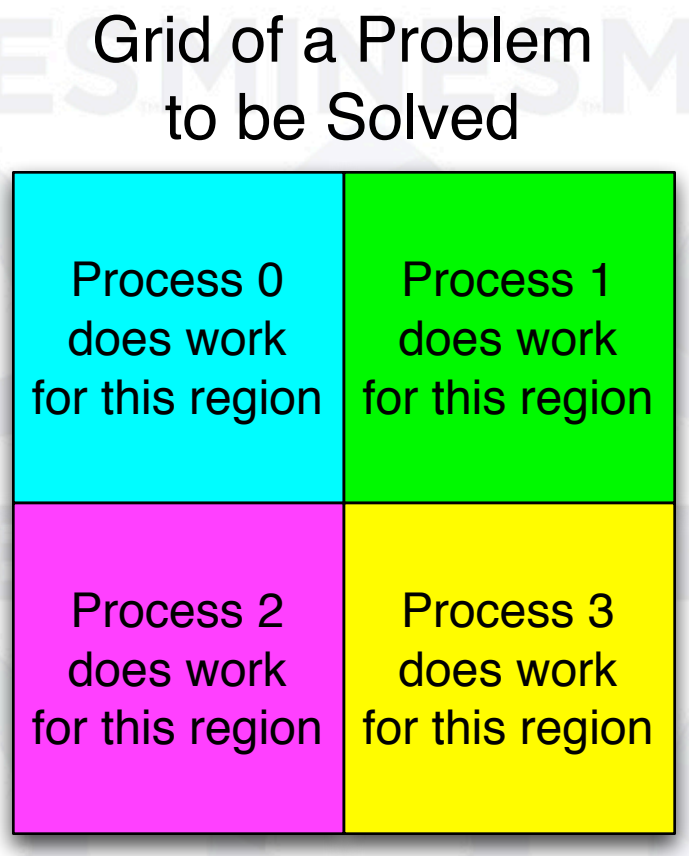
What is Parallelism?

- Consider your favorite computational application
  - One processor can give me results in N hours
  - Why not use N processors
    -- and get the results in just one hour?

The concept is simple:
Parallelism = applying multiple processors
to a single problem

Parallel computing is computing by committee

- Parallel computing: the use of multiple computers or processors working together on a common task.
  - Each processor works on its section of the problem
  - Processors are allowed to exchange information with other processors



Why do parallel computing?

- Limits of single CPU computing
  - Available memory
  - Performance
- Parallel computing allows:
  - Solve problems that don't fit on a single CPU
  - Solve problems that can't be solved in a reasonable time

Why do parallel computing?

- We can run…
  - Larger problems
  - Faster
  - More cases
  - Run simulations at finer resolutions
  - Model physical phenomena more realistically

Weather Forecasting

- Atmosphere is modeled by dividing it into three-dimensional regions or cells
  - 1 mile x 1 mile x 1 mile (10 cells high)
  - about 500×10^6 cells.
- The calculations of each cell are repeated many times to model the passage of time.
- About 200 floating point operations per cell per time step or 10^{11} floating point operations necessary per time step
- 10 day forecast with 10 minute resolution => 1.5×10^{14} flop
  - 100 Mflops would take about 17 days
  - 1.7 Tflops would take 2 minutes
  - 17 Tflops would take 12 seconds

Modeling Motion of Astronomical bodies (brute force)

- Each body is attracted to each other body by gravitational forces.
- Movement of each body can be predicted by calculating the total force experienced by the body.
- For N bodies, N - 1 forces / body yields N^2 calculations each time step
  - A galaxy has, 10^{11} stars => 10^9 years for one iteration
  - Using a N log N efficient approximate algorithm => about a year
  - NOTE: This is closely related to another hot topic: Protein Folding

Types of parallelism two extremes

- Data parallel
  - Each processor performs the same task on different data
  - Example - grid problems
  - Bag of Tasks or Embarrassingly Parallel is a special case
- Task parallel
  - Each processor performs a different task
  - Example - signal processing such as encoding multitrack data
  - Pipeline is a special case

Simple data parallel program

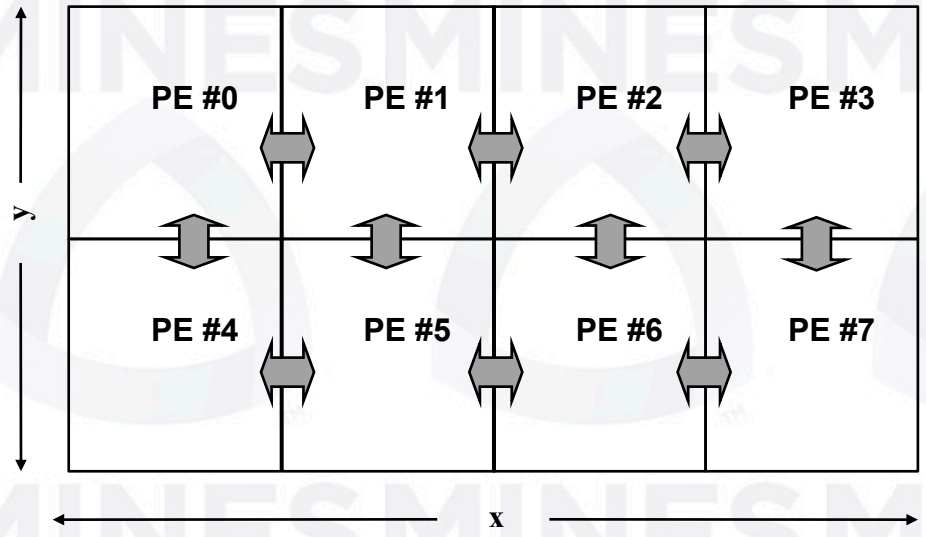
- Example: integrate 2-D propagation problem

Starting partial differential equation:

$$\frac{\partial \Psi}{\partial t} = D \times \frac{\partial^2 \Psi}{\partial x^2} + B \times \frac{\partial^2 \Psi}{\partial y^2}$$

Finite Difference Approximation:

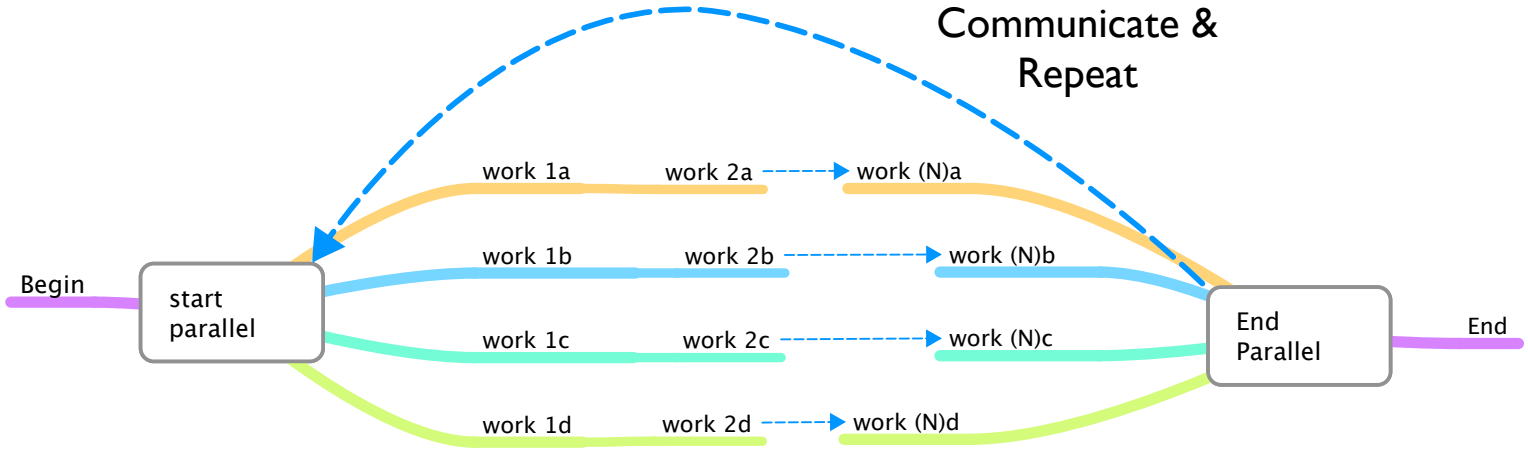
$$\frac{f_i^{n+1} - f_i^n}{\Delta t} = D \times \frac{f_{i+1,j}^n - 2f_{i,j}^n - f_{i-1,j}^n}{\Delta x^2} + B \times \frac{f_{i,j+1}^n - 2f_{i,j}^n - f_{i,j-1}^n}{\Delta y^2}$$



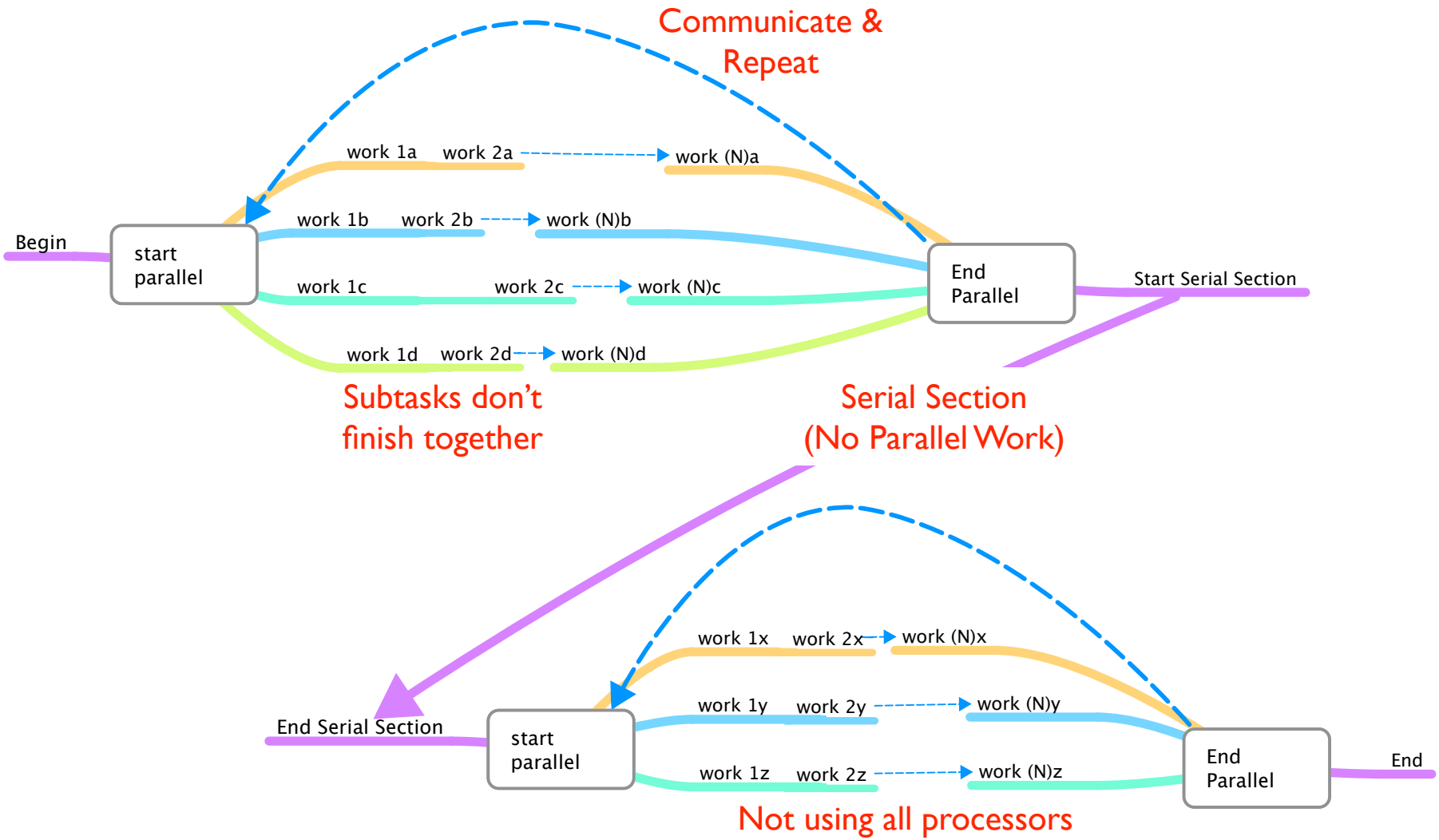
Typical Task Parallel Application

- Signal processing
  - Use one processor for each task
  - Can use more processors if one is overloaded
  - This is a pipeline

Parallel Program Structure



Parallel Problems



A "Real" example

```
#!/usr/bin/env python
from sys import argv
from os.path import isfile
from time import sleep
from math import sin,cos
#
fname="message"
my_id=int(argv[1])
print my_id, "starting program"
#
```

```
if (my_id == 1):
    sleep(2)
    myval=cos(10.0)
    mf=open(fname,"w")
    mf.write(str(myval))
    mf.close()
```

```
else:
    myval=sin(10.0)
    notready=True
    while notready :
        if isfile(fname) :
            notready=False
            sleep(3)
            mf=open(fname,"r")
            message=float(mf.readline())
            mf.close()
            total=myval**2+message**2
        else:
            sleep(5)
```

```
print my_id, "done with program"
```

Theoretical upper limits

- All parallel programs contain:
  - Parallel sections
  - Serial sections
- Serial sections are when work is being duplicated or no useful work is being done, (waiting for others)
- Serial sections limit the parallel effectiveness
  - If you have a lot of serial computation then you will not get good speedup
  - No serial work "allows" perfect speedup
- Amdahl's Law states this formally

Amdahl's Law

- Amdahl's Law places a strict limit on the speedup that can be realized by using multiple processors.
  - Effect of multiple processors on run time

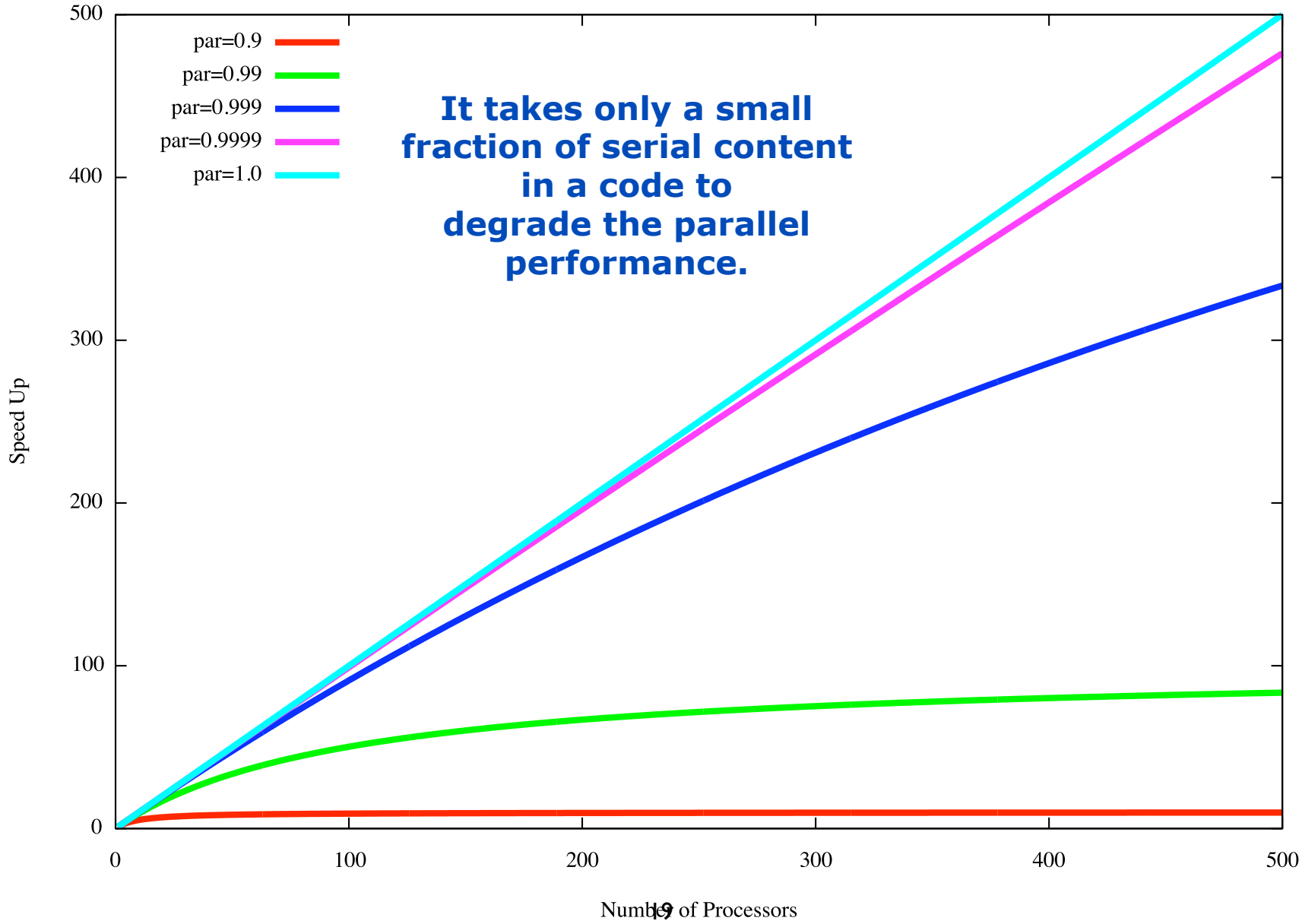
$$t_p = (f_p/N + f_s)\, t_s$$

  - Effect of multiple processors on speed up

$$S = t_s / t_p = \frac{1}{f_p/N + f_s}$$

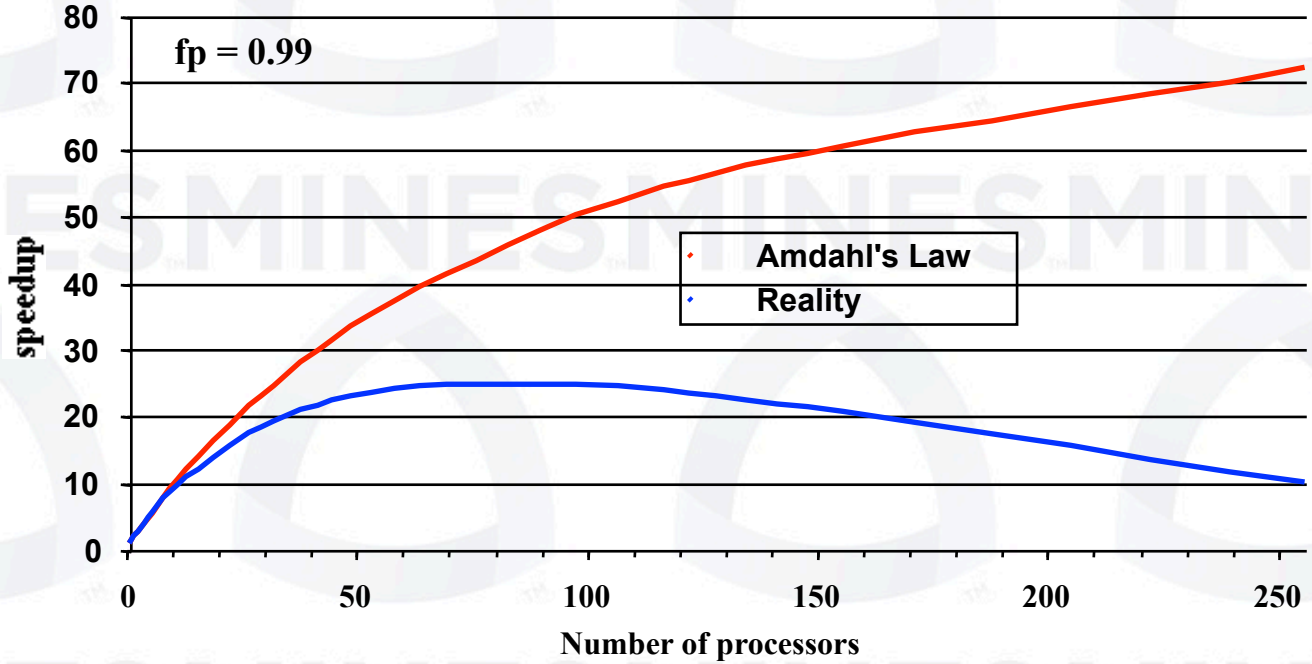
  - Where
    - Fs = serial fraction of code
    - Fp = parallel fraction of code
    - N = number of processors
  - Perfect speedup t=t1/n or S(n)=n

Illustration of Amdahl's Law

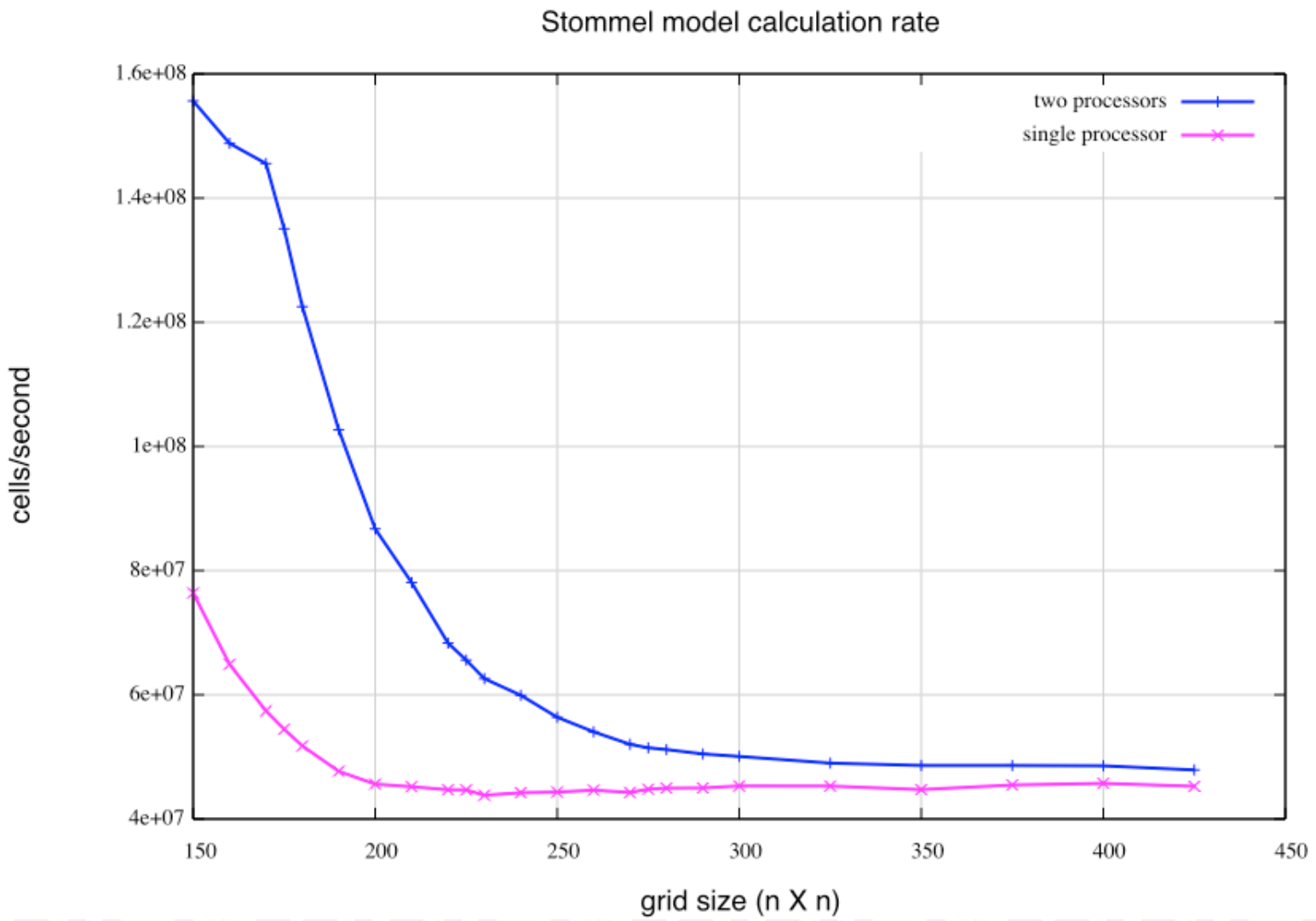


Amdahl's Law Vs. Reality

- Amdahl's Law provides a theoretical upper limit on parallel speedup assuming that there are no costs for communications.
- In reality, communications will result in a further degradation of performance



Sometimes you don't get what you expect!



Some other considerations

- Writing effective parallel application is difficult
  - Communication can limit parallel efficiency
  - Serial time can dominate
  - Load balance is important
- Is it worth your time to rewrite your application
  - Do the CPU requirements justify parallelization?
  - Will the code be used just once?

Parallelism Carries a Price Tag

- Parallel programming
  - Involves a steep learning curve
  - Is effort-intensive
- Parallel computing environments are unstable and unpredictable
  - Don't respond to many serial debugging and tuning techniques
  - May not yield the results you want, even if you invest a lot of time

Will the investment of your time be worth it?

Terms related to algorithms

- Amdahl's Law (talked about this already)
- Superlinear Speedup
- Efficiency
- Cost
- Scalability
- Problem Size
- Gustafson's Law

Superlinear Speedup

S(n) > n, may be seen on occasion, but usually this is due to using a suboptimal sequential algorithm or some unique feature of the architecture that favors the parallel formation.

One common reason for superlinear speedup is the extra cache in the multiprocessor system which can hold more of the problem data at any instant, it leads to less, relatively slow memory traffic.

Efficiency

Efficiency = Execution time using one processor over the Execution time using a number of processors

$$= \frac{t_s}{t_p \times n}$$

Its just the speedup divided by the number of processors

$$E = \frac{S(n)}{n} \times 100\%$$

Cost

The processor-time product or cost (or work) of a computation defined as
Cost = (execution time) x (total number of processors used)

The cost of a sequential computation is simply its execution time, t_s . The cost of a parallel computation is t_p x n. The parallel execution time, t_p , is given by $t_s/S(n)$

Hence, the cost of a parallel computation is given by

$$\text{Cost} = \frac{t_s n}{S(n)} = \frac{t_s}{E}$$

Cost-Optimal Parallel Algorithm
One in which the cost to solve a problem on a multiprocessor is proportional to the cost

Scalability

Used to indicate a hardware design that allows the system to be increased in size and in doing so to obtain increased performance - could be described as architecture or hardware scalability.

Scalability is also used to indicate that a parallel algorithm can accommodate increased data items with a low and bounded increase in computational steps - could be described as algorithmic scalability.

Problem size

Problem size: the number of basic steps in the best sequential algorithm for a given problem and data set size

- Intuitively, we would think of the number of data elements being processed in the algorithm as a measure of size.

- However, doubling the date set size would not necessarily double the number of computational steps. It will depend upon the problem.

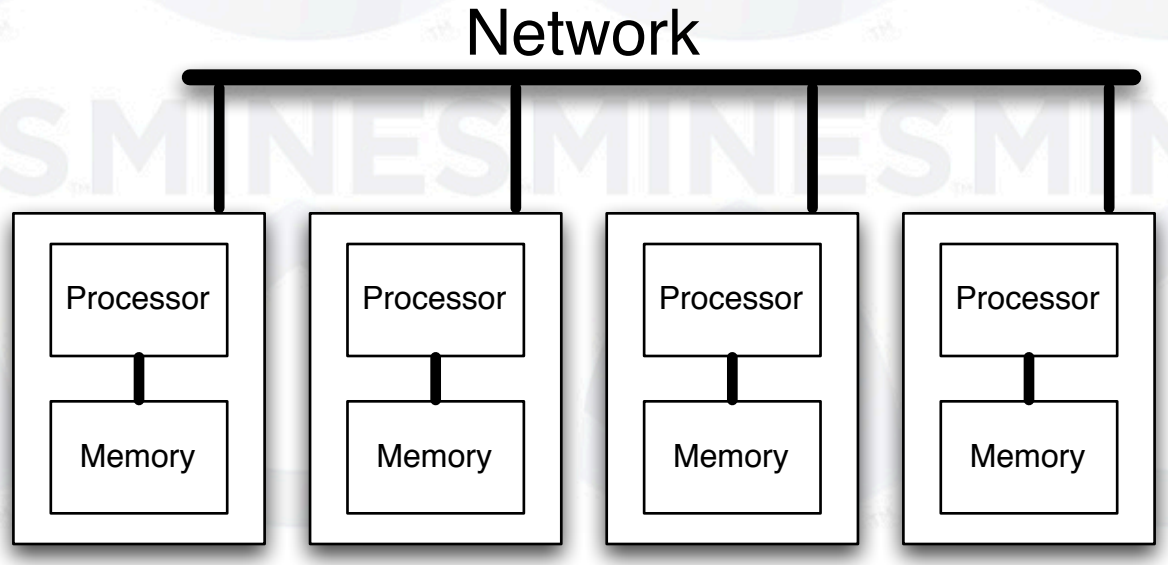
- For example, adding two matrices has this effect, but multiplying matrices quadruples operations.

Note: Bad sequential algorithms tend to scale well

Other names for Scaling

- Strong Scaling (Engineering)
  - For a fixed problem size how does the time to solution vary with the number of processors
- Weak Scaling
  - How the time to solution varies with processor count with a fixed problem size per processor

Some Classes of machines



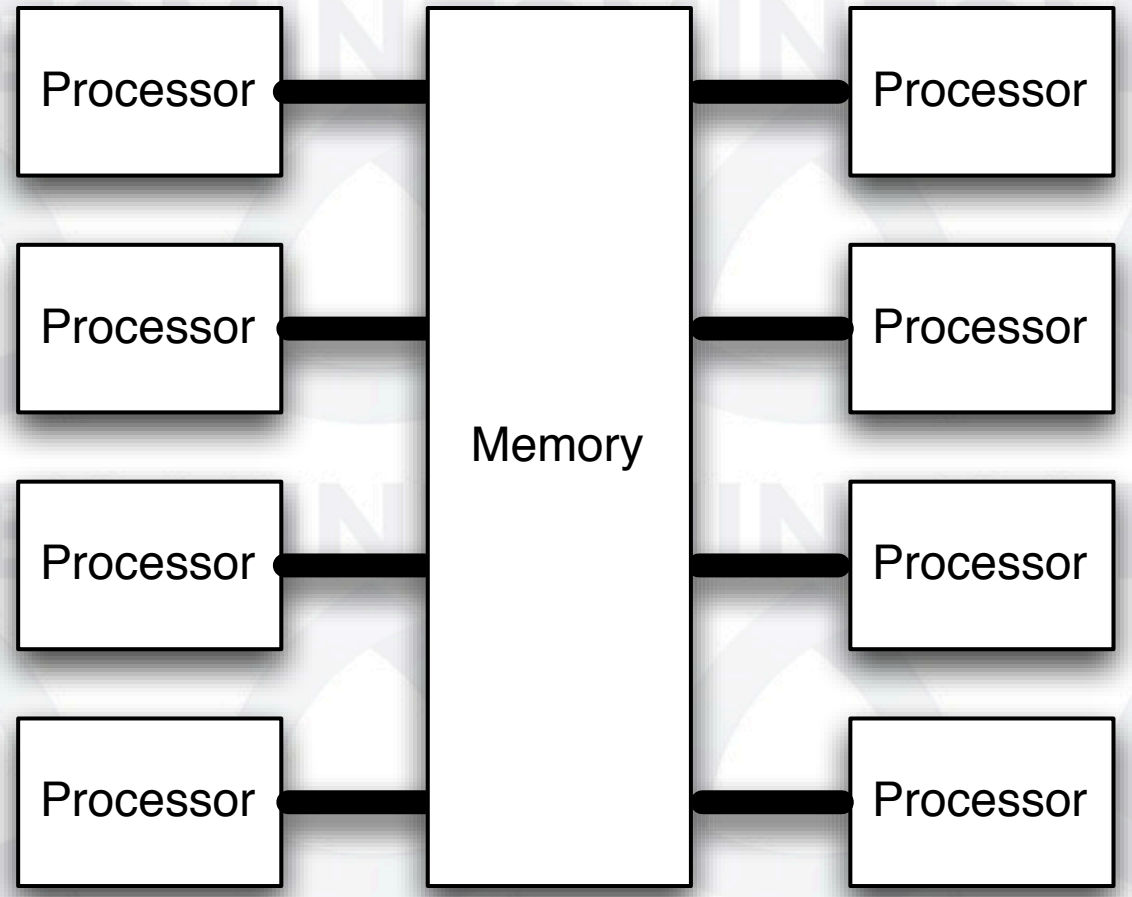
Distributed Memory
Processors only Have access to
their local memory
"talk" to other processors over a network

Some Classes of machines

Uniform Shared Memory (UMA)

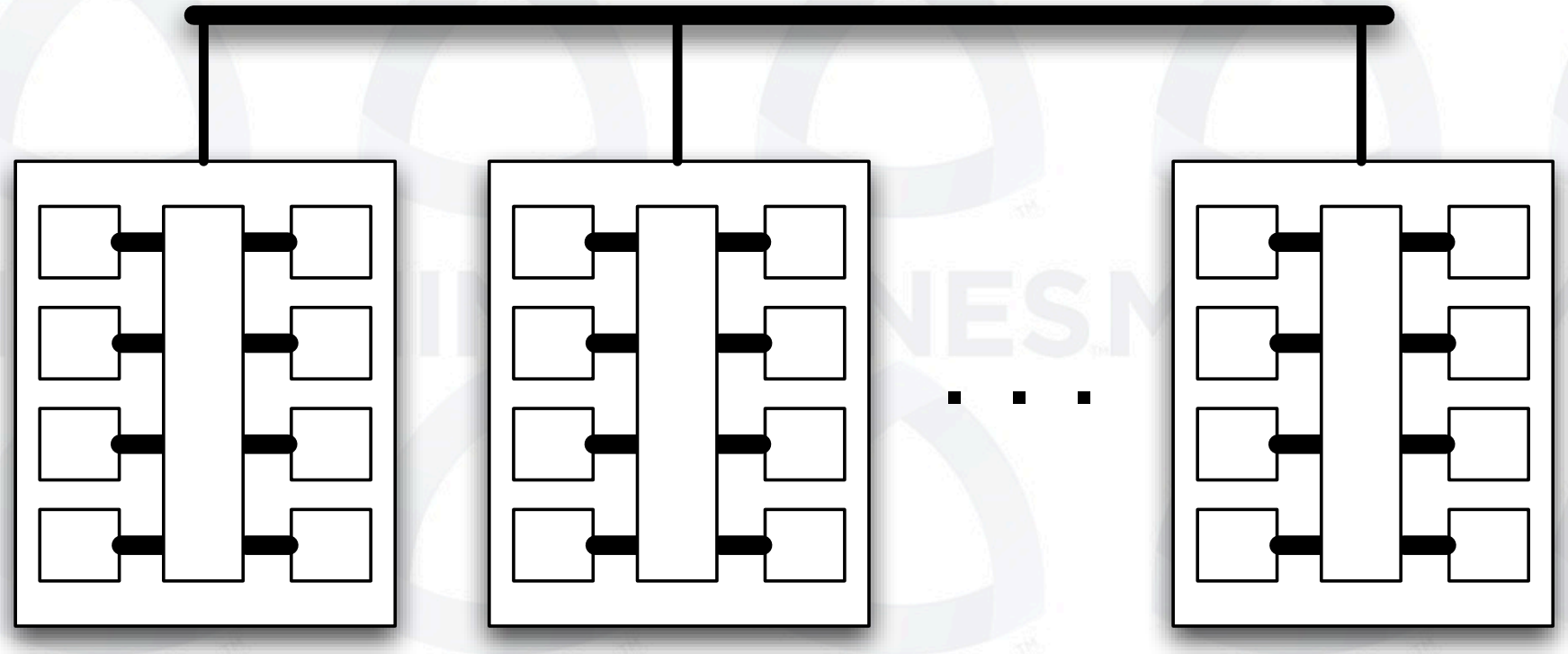
All processors have equal access to Memory

Can "talk" via memory



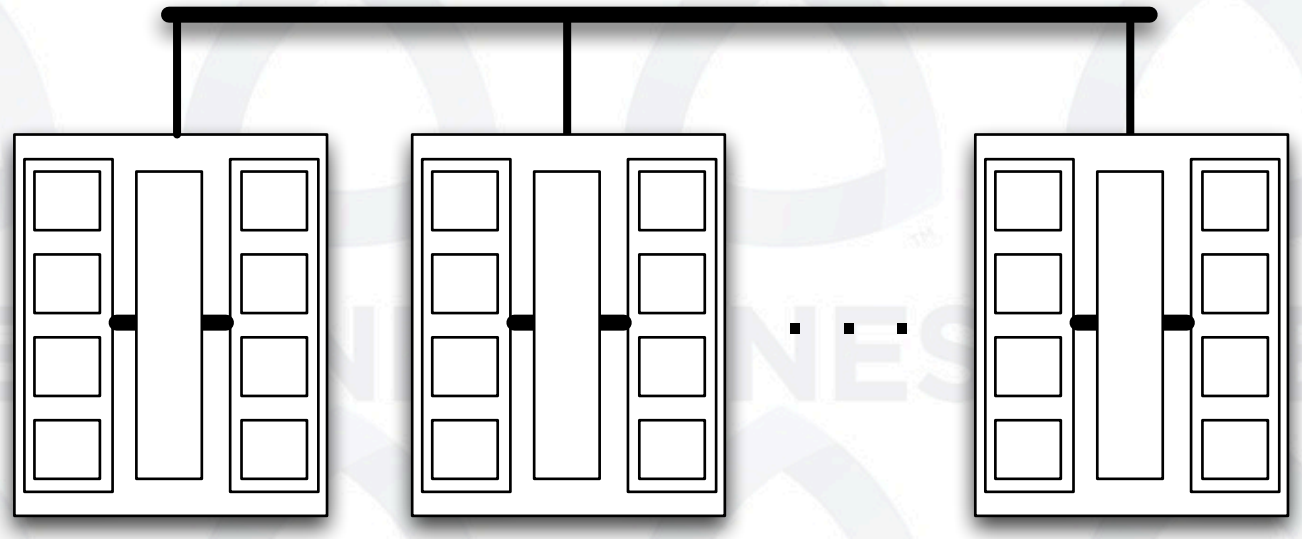
Some Classes of machines

Hybrid
Shared memory nodes
connected by a network



Some Classes of machines

More common today
Each node has a collection
of multicore chips

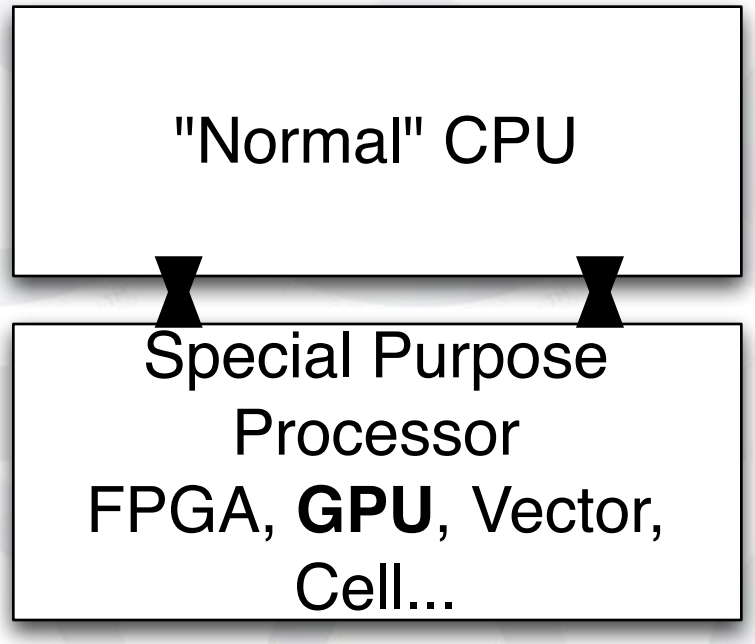


Ra has 268 nodes
256 quad core dual socket
12 dual core quad socket

Some Classes of machines

Hybrid Machines

- Add special purpose processors to normal processors
- Not a new concept but, regaining traction
- Example: our Tesla Nvidia node, cuda I



Network Topology

- For ultimate performance you may be concerned how you nodes are connected.
- Avoid communications between distant node
- For some machines it might be difficult to control or know the placement of applications

Network Terminology

- Latency
  - How long to get between nodes in the network.
- Bandwidth
  - How much data can be moved per unit time.
  - Bandwidth is limited by the number of wires and the rate at which each wire can accept data and choke points

Ring

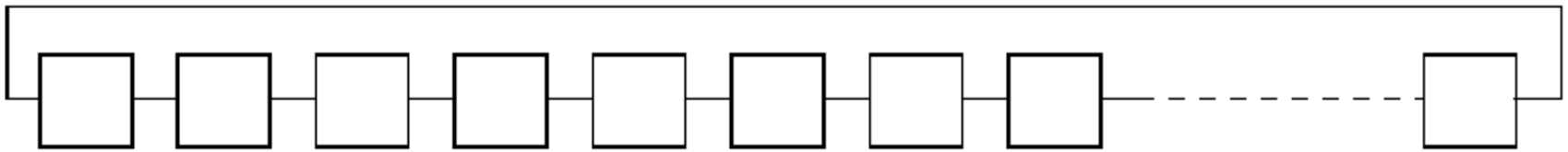


Figure 1.10 Ring.

Grid

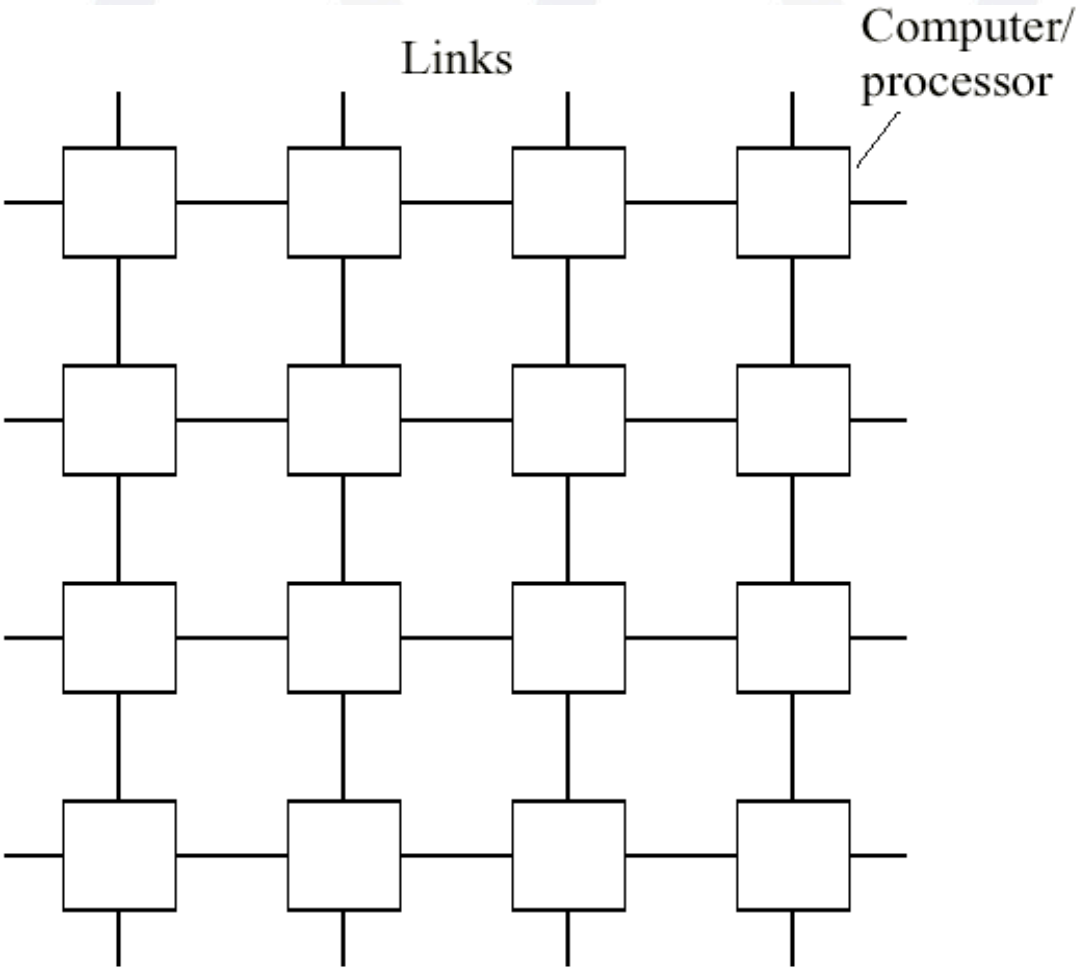
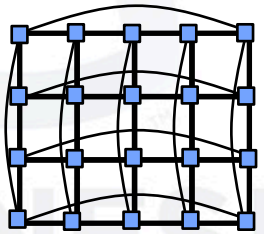


Figure 1.11 Two-dimensional array (mesh).



Wrapping produces torus

Tree

Fat tree
the lines get
wider as you
go up

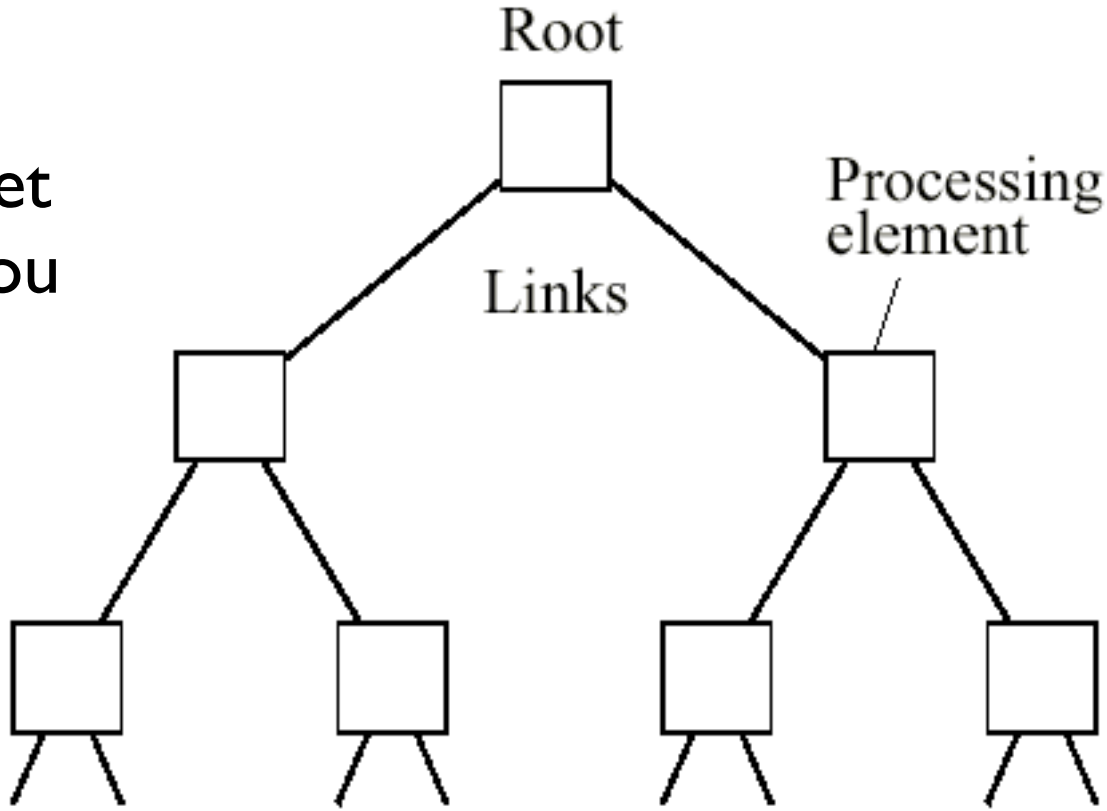
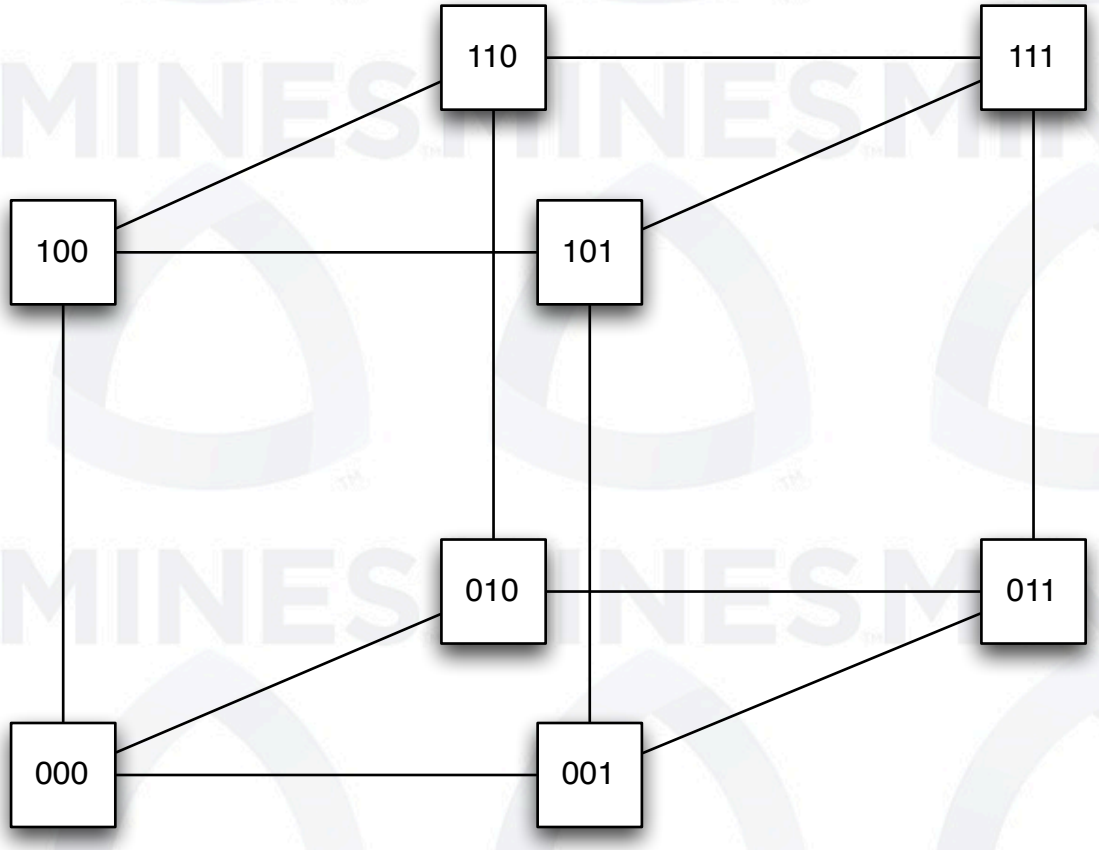


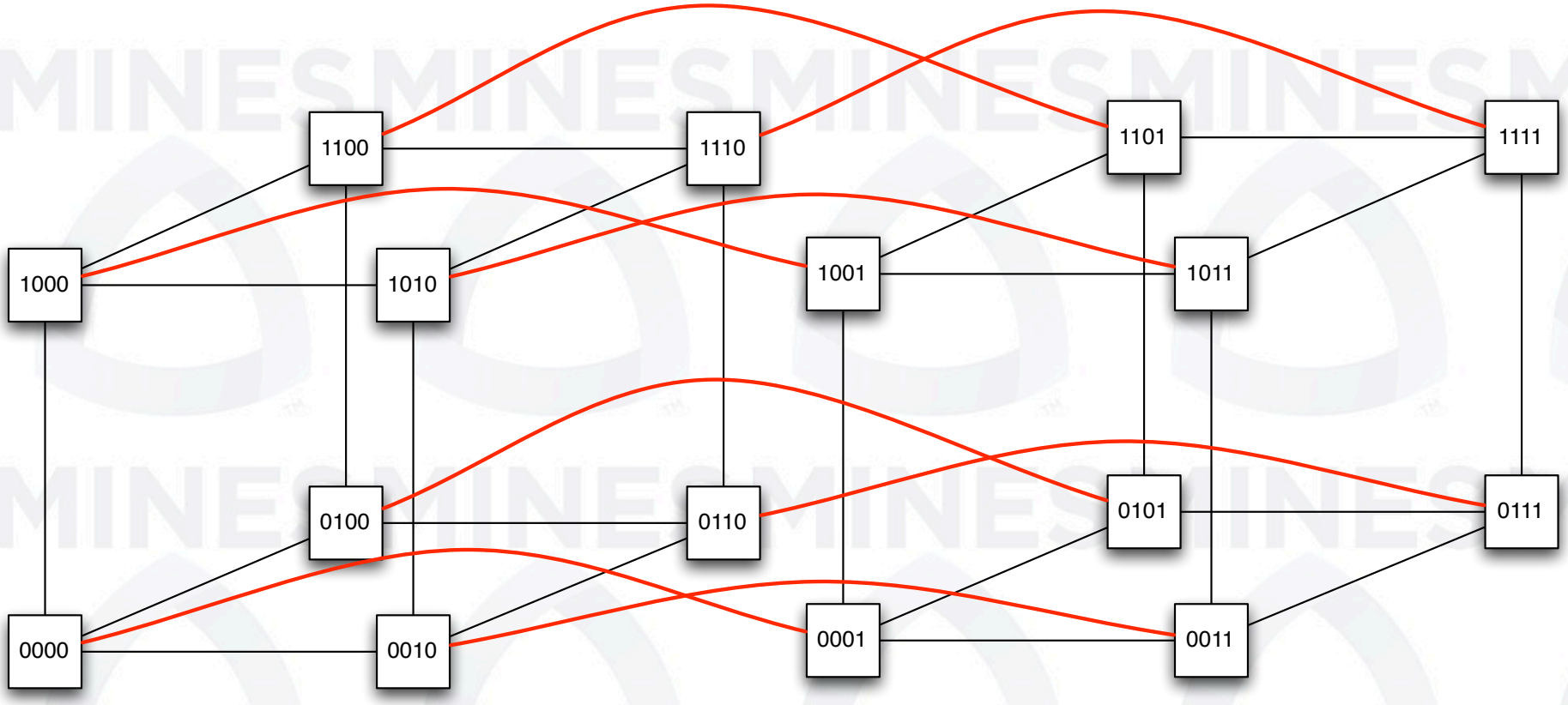
Figure 1.12 Tree structure.

Hypercube



3 dimensional hypercube

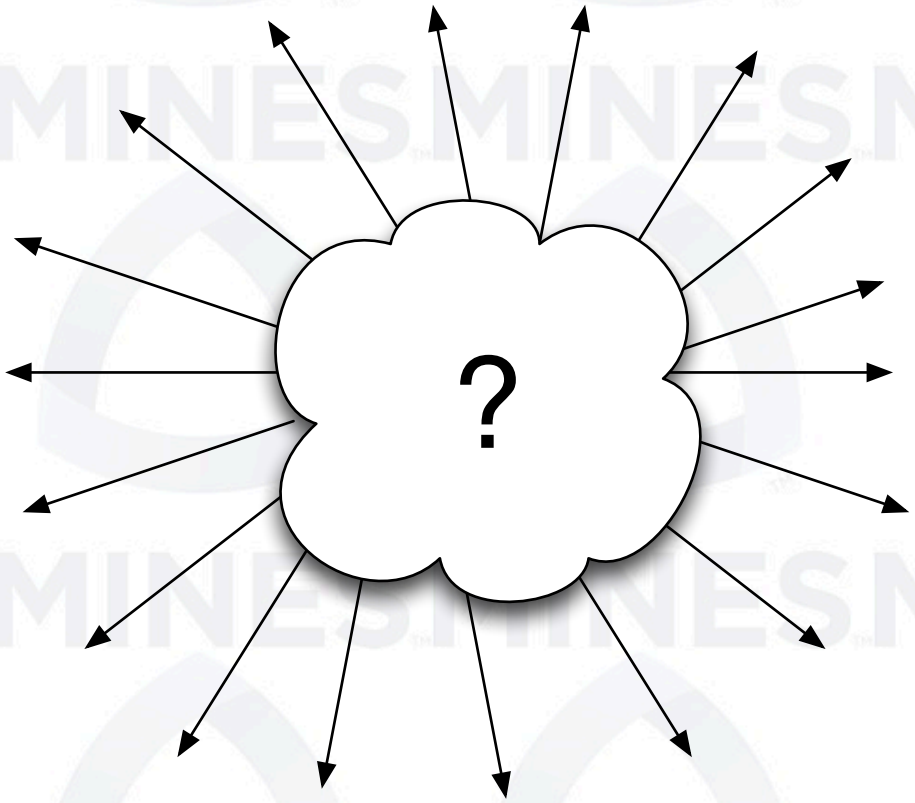
4D Hypercube



Some communications algorithms are hypercube based

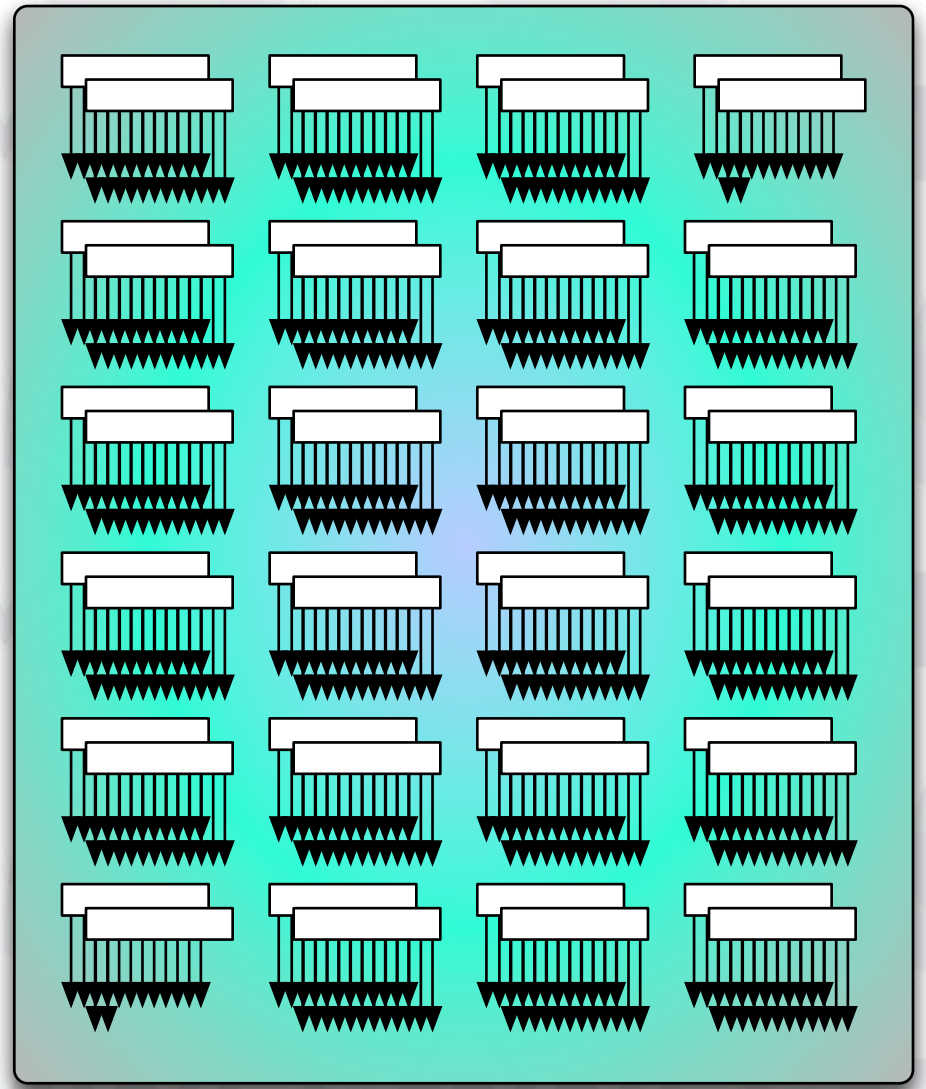
How big would a 7d hypercube be?

Star



Quality depends on what is in the center

Example: An Infiniband Switch



Infiniband, DDR, Cisco 7024 IB Server Switch - 48 Port

Adaptors. Each compute node has one DDR 1-Port HCA

4X DDR=> 16Gbit/sec

140 nanosecond hardware latency

1.26 microsecond at software level

Measured Bandwidth