MAE 5032 High Performance Computing: Methods and Practices

Lecture 14: Parallel Computing Basics

Ju Liu
Department of Mechanics and Aerospace Engineering
liuj36@sustech.edu.cn

Sources of Parallelism: Discrete Events

Parallelism in Simulation

- Parallelism and data locality are both important to performance
- Real world problems have parallelism and locality
  - Many objects operate independently of others
  - Objects often depend much more on nearby than distant objects
  - Dependence on distant objects can often be simplified
  - When a continuous problem is discretized, time dependencies are generally limited to adjacent time steps
  - Far-field effects may be ignored or approximated in many cases
- Many problems exhibit parallelism at multiple levels

Discrete Event Systems

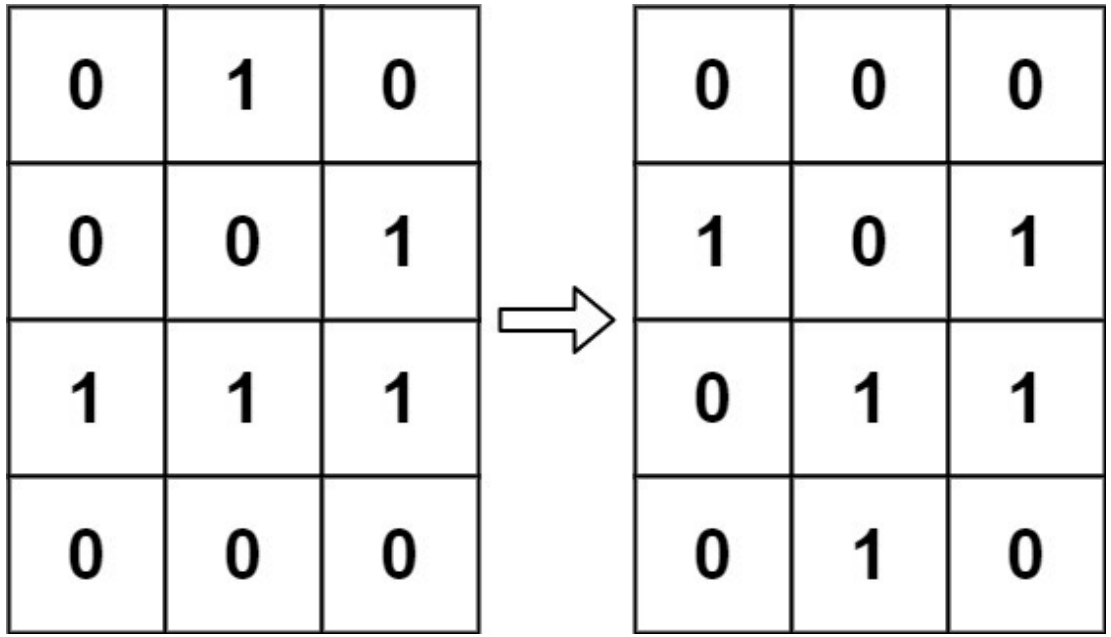
- Systems are represented as:
  - finite set of variables.
  - the set of all variable values at a given time is called the **state**.
  - each variable is updated by computing a **transition** function depending on the other variables.

- System may be:
  - **synchronous**: at each discrete timestep evaluate all transition functions; also called a state machine.
  - **asynchronous**: transition functions are evaluated only if the inputs change, based on an "event" from another part of the system; also called event driven simulation.

- Example: The "game of life:"
  - Also known as Sharks and Fish #3:
  - Space is divided into cells; rules govern cell contents at each step

Sharks and Fish / Game of Life

- The universe of the game of life is an infinite two-dimensional orthogonal grid of square cells, each of which is one of two possible states: live or dead
- At each step in time, the following transition occur:
  - Any live cell with fewer than 2 live neighbors dies, as if by underpopulation
  - Any live cell with 2 or 3 live neighbors lives on to the next generation
  - Any live cell with more than 3 live neighbors dies, as if by overpopulation
  - Any dead cell with exactly 3 live neighbors. becomes a live cell, as if by reproduction



Sharks and Fish / Game of Life

- **S&F 1.** Fish alone move continuously subject to an external current and Newton's laws.
- **S&F 2.** Fish alone move continuously subject to gravitational attraction and Newton's laws.
- **S&F 3.** Fish alone play the "Game of Life" on a square grid.
- **S&F 4.** Fish alone move randomly on a square grid, with at most one fish per grid point.
- **S&F 5.** Sharks and Fish both move randomly on a square grid, with at most one fish or shark per grid point, including rules for fish attracting sharks, eating, breeding and dying.
- **S&F 6.** Like Sharks and Fish 5, but continuous, subject to Newton's laws.

Sharks and Fish / Game of Life

- The simulation is synchronous
  - use two copies of the grid (old and new), “ping-pong” between them
  - the value of each new grid cell depends only on 9 cells (itself plus 8 neighbors) in old grid.
  - simulation proceeds in timesteps-- each cell is updated at every step.

- Easy to parallelize by dividing physical domain: **Domain Decomposition**

P1	P2	P3
P4	P5	P6
P7	P8	P9

```
Repeat
        compute locally to update local system
        barrier()
        exchange state info with neighbors
        finish updates
until done simulating
```

- Locality is achieved by using large patches of the ocean
  - Only boundary values from neighboring patches are needed.

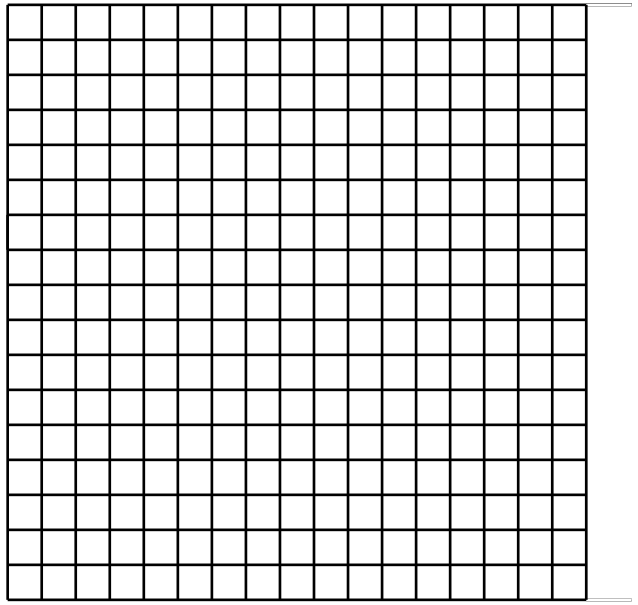
- How to pick shapes of domains?

Regular meshes

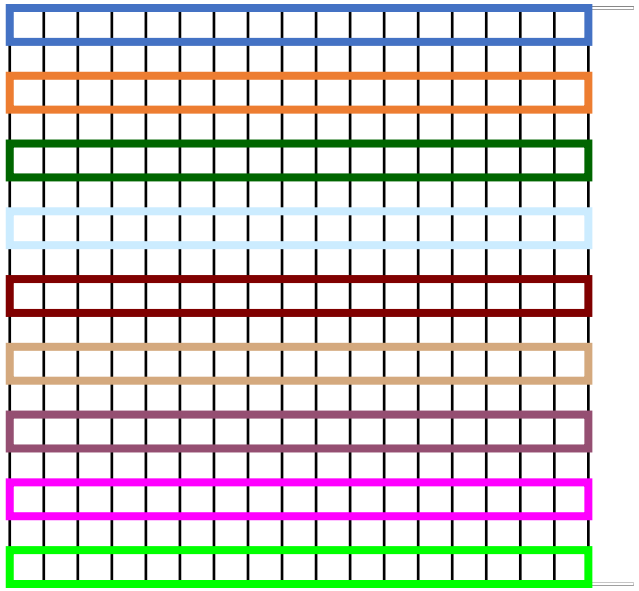
- Suppose graph is $n \times n$ with connection to NSEW neighbors
- minimize communications on mesh : minimizing surface to volume ratio of partition.
- Graph partitioning assigns subgraphs to processors:
  - determine parallelism and locality
  - Goal 1: evenly distribute subgraphs to processors (load balancing)
  - Goal 2: minimize edge crossings (minimizing communication)
  - NP-hard in general.

Regular meshes

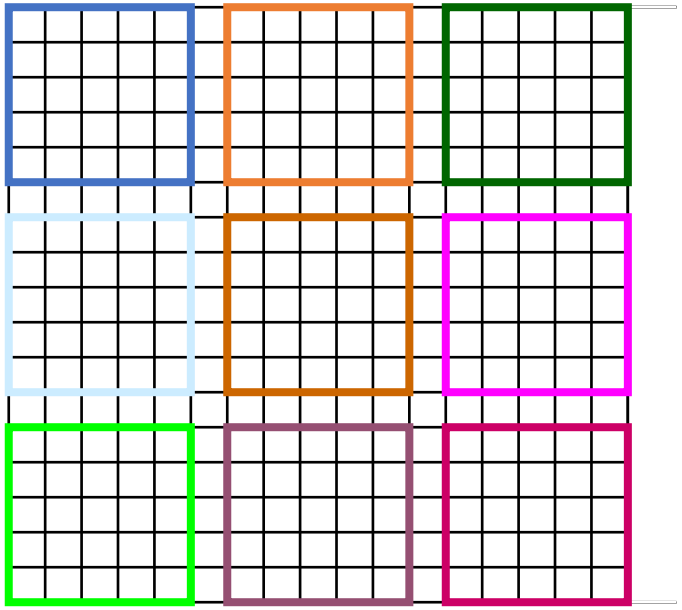
- Suppose graph is $n \times n$ with connection to NSEW neighbors
- minimize communications on mesh: minimizing surface to volume ratio of partition.



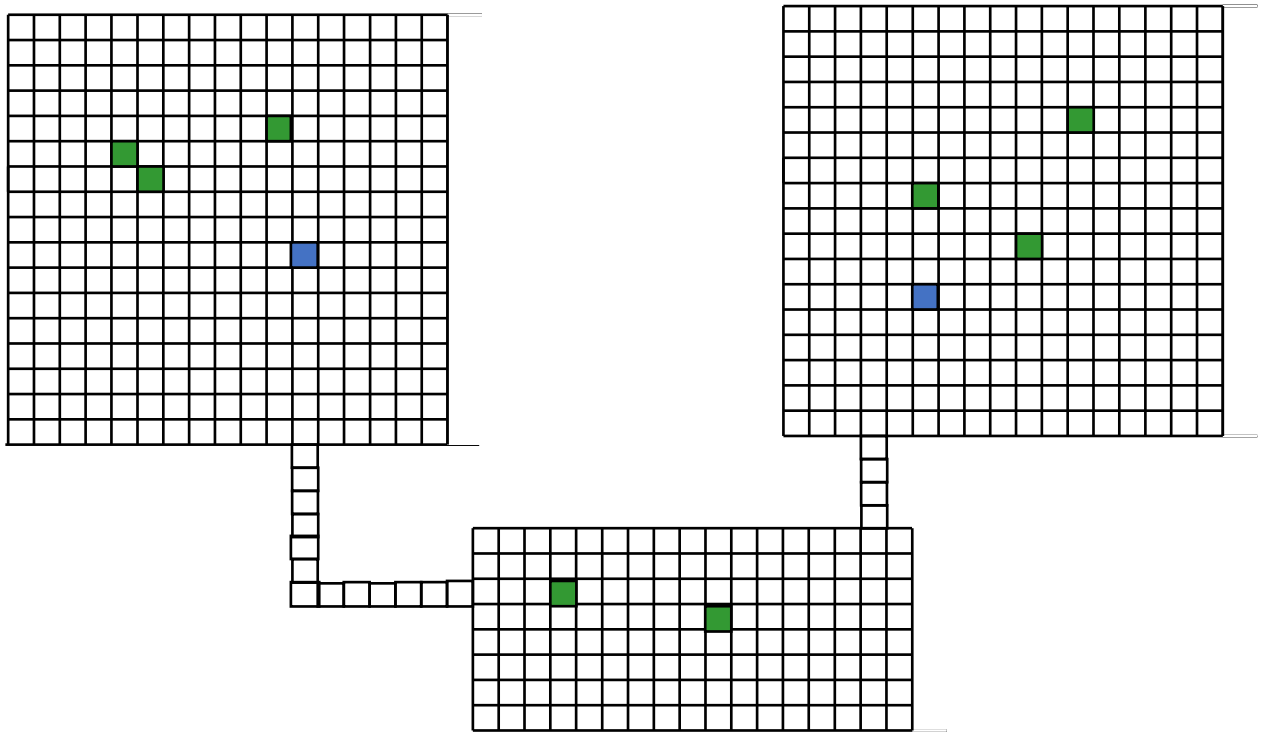
n*(p-1)
edge crossings



$2*n*(p^{1/2}-1)$
edge crossings



Loosely connected ponds



- Parallelization: each processor gets a set of ponds with roughly equal total area.
- One pond can affect another through streams but infrequently
- Asynchronous (event-driven) simulations update only when an event arrives from another component.
- Asynchronous is more efficient, but harder to parallelize (hard to predict when to perform a communication.)

Summary

- Parallelism and Locality arise naturally in simulation

  So far: Discrete Event Simulation (time and space discrete)

  Next: Particle Systems, Lumped variables (ODEs), Continuous variables (PDEs), …

- Discrete Event Simulation
  - Game of Life, Digital Circuits, Pacman, …
  - Finite set of variables, values at a given time called **state**
  - Each variable updated by **transition function** depending on others
  - Assign work to processors (**domain decomposition***)*
    - Goals*:* balance load and minimize communication
    - Represent problem by graph

      Nodes are states, edges are dependencies

      Partition graph (NP hard, so approximate) → **METIS**
  - **Synchronous**: update all values at each discrete timestep
  - **Asynchronous**: update a value only if inputs change, when an “event” occurs; also called event driven simulation

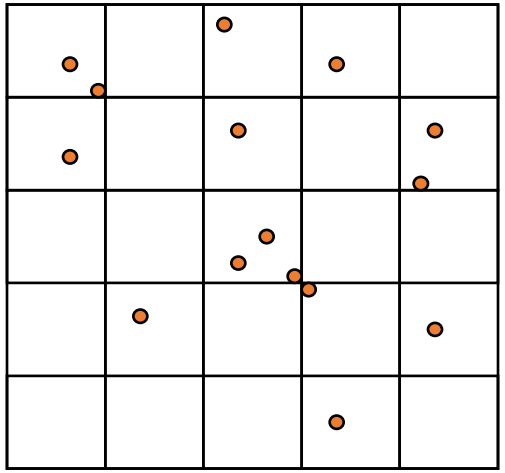
Sources of Parallelism: Particle Systems

Particle systems

- A particle system has
  - a finite number of particles
  - moving in space according to Newton's Laws (i.e. F = ma)
  - Time and positions are continuous
- Examples
  - stars in space with laws of gravity
  - electron beam in semiconductor manufacturing
  - atoms in a molecule with electrostatic forces
  - neutrons in a fission reactor
  - cars on a freeway with Newton's laws plus model of driver and engine
  - balls in a pinball game
- Reminder: many simulations combine techniques such as particle simulations with some discrete events (e.g. Game of Life)

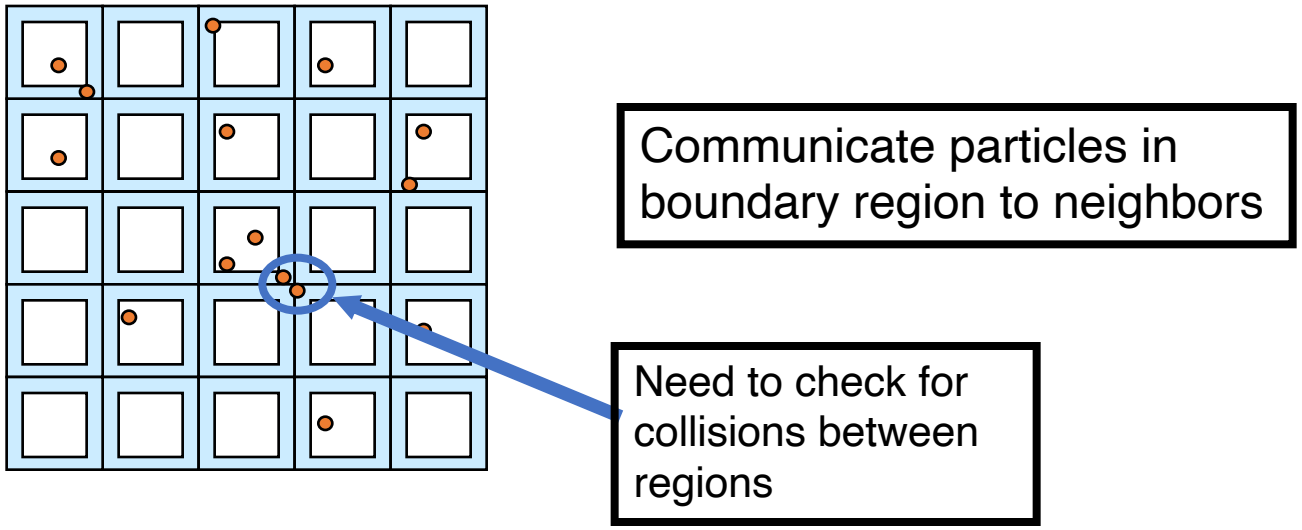
Parallelism in nearby forces

- Nearby forces require interaction and therefore communication.
- Force may depend on other nearby particles:
  - Example: collisions.
  - simplest algorithm is $O(n^2)$: look at all pairs to see if they collide.
- Usual parallel model is domain decomposition of physical region in which particles are located
  - O(n/p) particles per processor if evenly distributed.



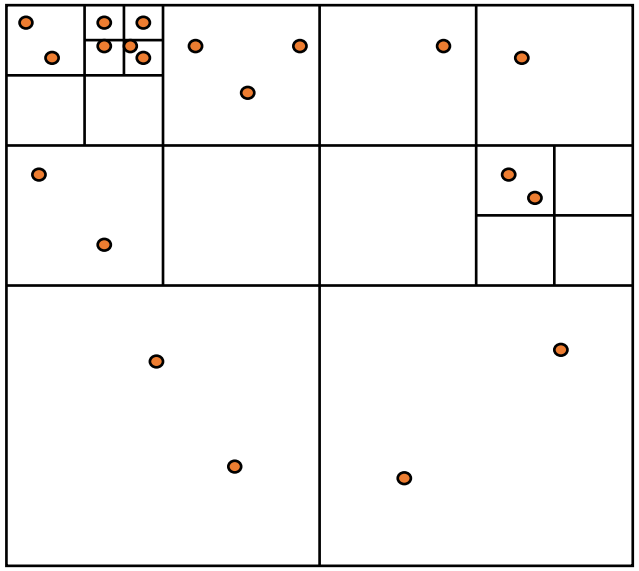
Parallelism in nearby forces

- Challenge 1: interactions of particles near processor boundary:
  - ➢ need to communicate particles near boundary to neighboring processors.
    - Region near boundary called “ghost zone” or “halo”
  - ➢ Low surface to volume ratio means low communication.
    - Use squares, not slabs, to minimize ghost zone sizes



Parallelism in nearby forces

- Challenge 2: load imbalance, if particles cluster:
  - galaxies, electrons hitting a device wall.
- To reduce load imbalance, divide space unevenly.
  - Each region contains roughly equal number of particles.
  - Quad-tree in 2D, oct-tree in 3D.



Example: each square contains at most 3 particles

- May need to rebalance as particles move, hopefully seldom

Parallelism in far-field forces

- Far-field forces involve all-to-all interaction and therefore communication.
- Force depends on all other particles:
  - Examples: van der Waals force
  - Simplest algorithm is $O(n^2)$ as in S&F 2, 4, 5.
  - Just decomposing space does not help since every particle needs to "visit" every other particle.
- Use more clever algorithms to beat $O(n^2)$.

Parallelism in far-field forces

- Based on approximation:

  Superimpose a regular mesh.

  "Move" particles to nearest grid point.

- Exploit fact that the far-field force satisfies a PDE that is easy to solve on a regular mesh:

  FFT, multigrid (described in future maybe)

  Cost drops to O(n log n) or O(n) instead of $O(n^2)$

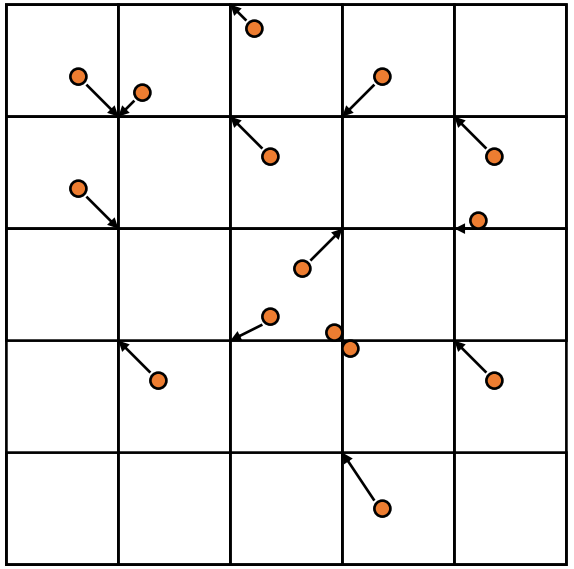
- Accuracy depends on the fineness of the grid is and the uniformity of the particle distribution.

  1) Particles are moved to nearby mesh points (scatter)

  2) Solve mesh problem

  3) Forces are interpolated at particles from mesh points (gather)

  scatter and gather are common operations in many libraries.



Summary

- Model contains discrete entities, namely, particles
- Time is continuous – must be discretized to solve
- Simulation follows particles through timesteps
  - All-pairs algorithm is simple, but inefficient, $O(n^2)$
  - Particle-mesh methods approximates by moving particles to a regular mesh, where it is easier to compute forces
- May think of this as a special case of a “lumped” system

Sources of Parallelism: ODEs

Solving ODEs

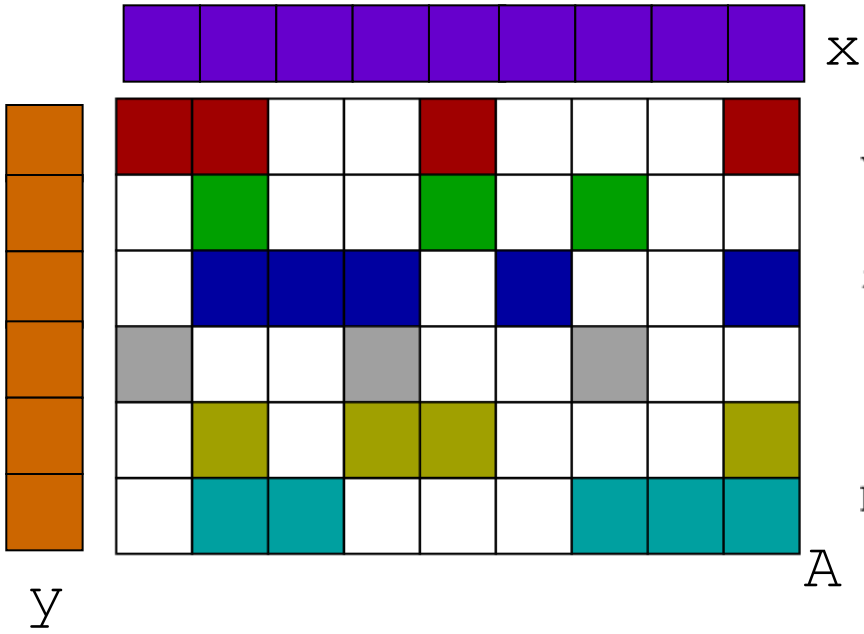
- In the discretized ODEs, the matrices are sparse:
  i.e., most array elements are 0.
  neither store nor compute on these 0's.
  Sparse because each component only depends on a few others

- Given a set of ODEs, two kinds of questions are:
  - Compute the values of the variables at some time t
    - Explicit methods
    - Implicit methods
  - Compute modes of vibration
    - Eigenvalue problems

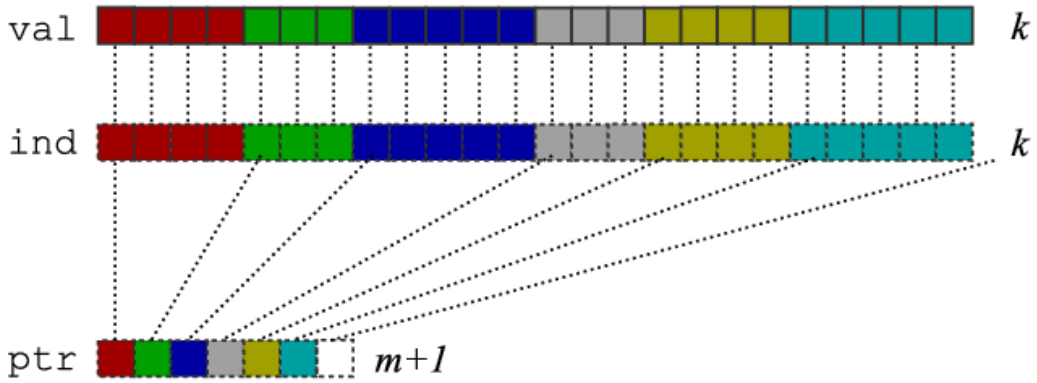
ODE methods

- Explicit methods for ODEs need sparse-matrix-vector mult.
- Implicit methods for ODEs need to solve linear systems
- Direct methods (Gaussian elimination)
  - Called LU Decomposition, because we factor A = L*U.
  - More complicated than sparse-matrix vector multiplication.
- Iterative solvers
  - Jacobi, Successive over-relaxation (SOR) , Conjugate Gradient (CG), Multigrid,...
  - Most have sparse-matrix-vector multiplication in kernel.
- Eigenproblems (HW6)
  - Also depend on sparse-matrix-vector multiplication.

Sparse matrix-vector product



Representation of A

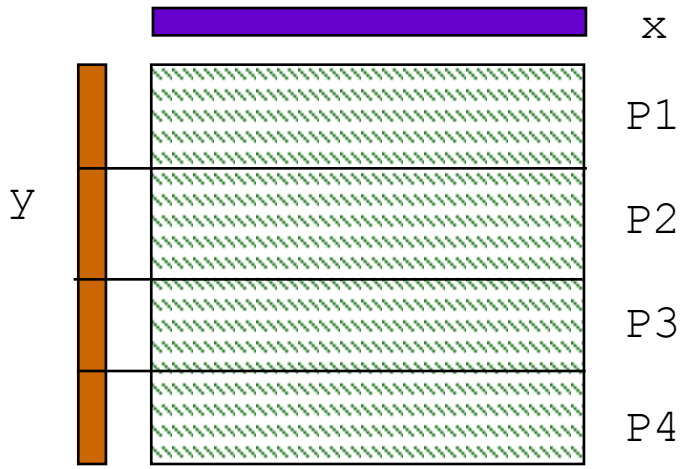


Matrix-vector multiply kernel: y(i) ← y(i) + A(i,j) × x(j)

```
for each row i
  for k=ptr[i] to ptr[i+1]-1 do
       y[i] = y[i] + val[k]*x[ind[k]]
```

Parallel sparse matrix-vector product

- y = A*x, where A is a sparse n x n matrix
- Questions
  - which processors store
    - y[i], x[i], and A[i,j]
  - which processors compute
    - y[i] = sum (from 1 to n) A[i,j] * x[j]
      - = (row i of A) * x … a sparse dot product

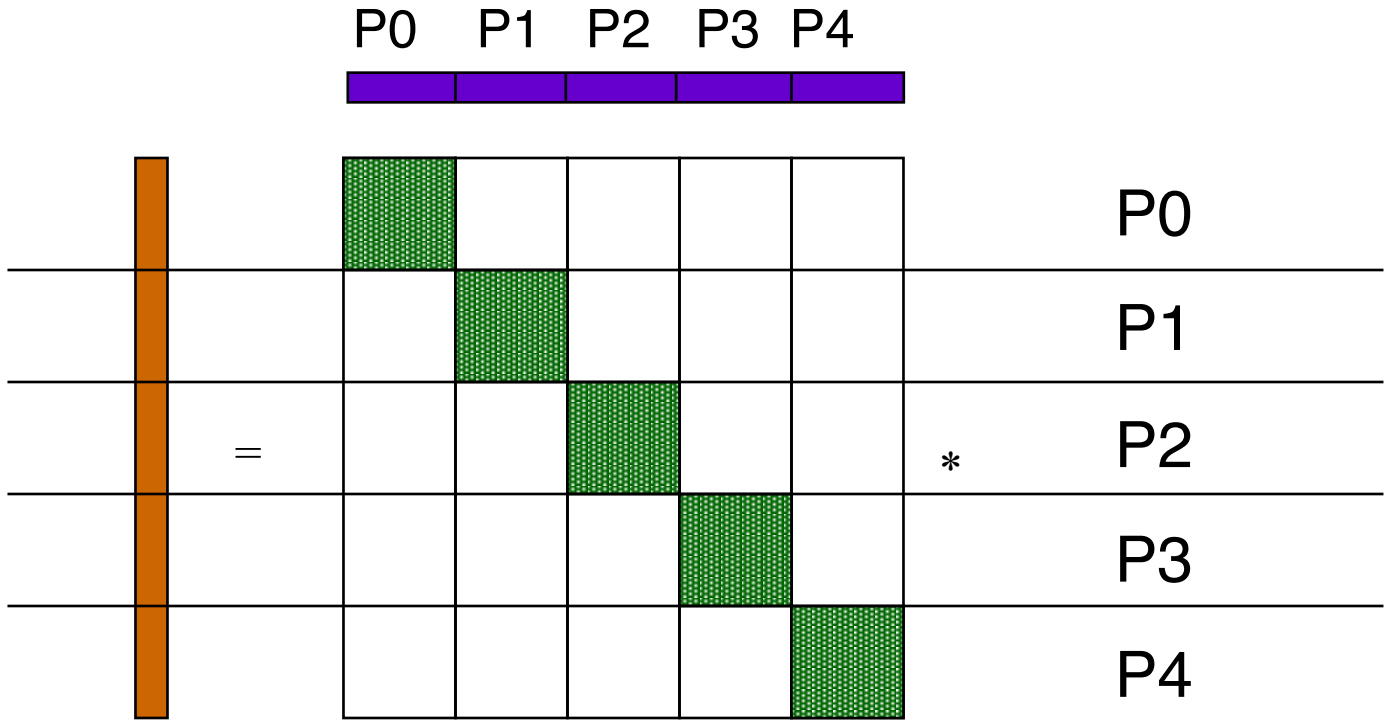


- Partitioning
  - Partition index set {1,…,n} = N1 ∪ N2 ∪ … ∪ Np.
  - For all i in Nk, Processor k stores y[i], x[i], and row i of A
  - For all i in Nk, Processor k computes y[i] = (row i of A) * x
    - "owner computes" rule: Processor k computes the y[i]s it owns.

May require communication

Matrix reordering via graph partitioning

- "Ideal" matrix structure for parallelism: block diagonal
  p (number of processors) blocks, can all be computed locally.
  If no non-zeros outside these blocks, no communication needed
- Can we reorder the rows/columns to get close to this?
  Most nonzeros in diagonal blocks, few outside



Matrix reordering via graph partitioning

- Performance goals
  - balance load (how is load measured?).
    - Approximately equal number of nonzeros (not necessarily rows)
  - balance storage (how much does each processor store?).
    - Approximately equal number of nonzeros
  - minimize communication (how much is communicated?).
    - Minimize nonzeros outside diagonal blocks
    - Related optimization criterion is to move nonzeros near diagonal
  - improve register and cache re-use
    - Group nonzeros in small vertical blocks so source (x) elements loaded into cache or registers may be reused (temporal locality)
    - Group nonzeros in small horizontal blocks so nearby source (x) elements in the cache may be used (spatial locality)
- Other algorithms reorder rows/columns for other reasons
  - Reduce # nonzeros in matrix after Gaussian elimination
  - Improve numerical stability

Summary

- Load Balancing
  - Statically - Graph partitioning
    Discrete event simulation
    Sparse matrix vector multiplication
  - Dynamically – if load changes significantly during job
- Linear algebra
  - Solving linear systems (sparse and dense)
  - Eigenvalue problems will use similar techniques

Designing Parallel Programs

Types of parallelism

- Data parallelism
  - each processor performs the same task on different data (like SIMD)
- Task parallelism
  - each processor performs a different task (like pipelining)
- Most applications fall somewhere on the continuum between these two extremes

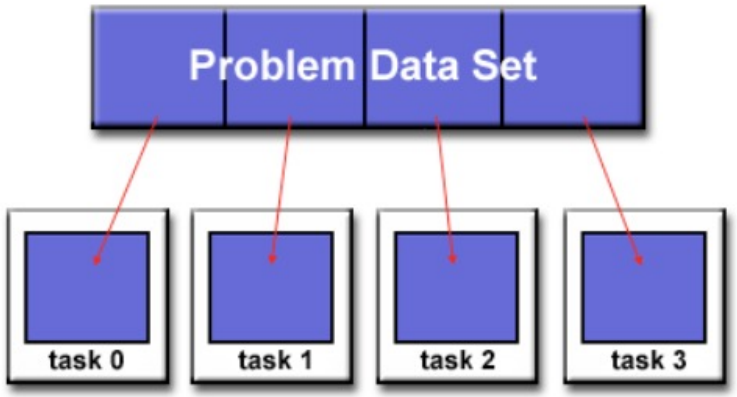
```
program:
…
if      CPU=a then
  start=1
  end  =50
elseif CPU=b then
  start=51
  end  =100
end if
do I = start, end
   work on A(I)
end do
...
end program
```

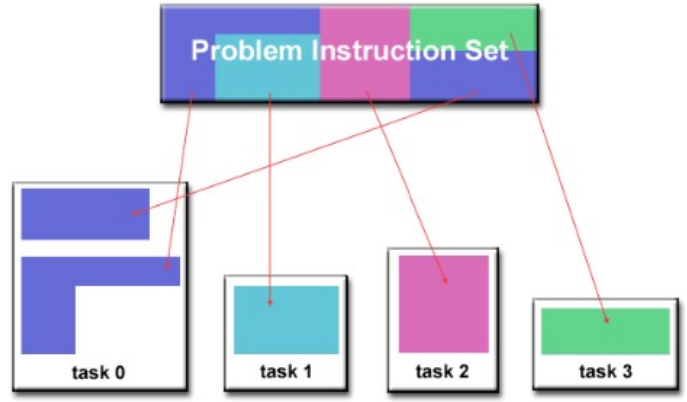
```
program.f:
…
initialize
…
if CPU=a then
   do task a
elseif CPU=b then
   do task b
end if
…
end program
```

Parallel program design

- Understand the problem
  - ➢ Determine if the problem is one that can be parallelized
    - Explicit scheme for time marching
    - Calculate the Fibonacci series by F(n) = F(n-1) + F(n-2)

- Decomposition/partitioning
  - ➢ Domain decomposition
  - ➢ Functional decomposition
  - ➢ or both





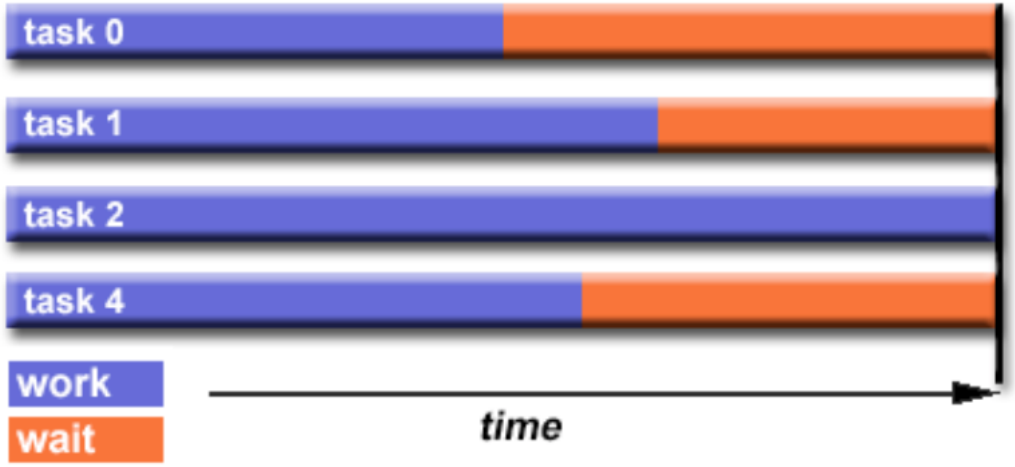
- Decide communication needs
  - ➢ Some tasks can be decomposed

  and executed with virtually no need for tasks to share data. These jobs are called embarrassingly parallel.
  - ➢ Most tasks are not quite so simple and do require sharing data with each other.

Parallel program design

- Data dependence exists between program statements when the order of execution affects the results of the program
  - Primary inhibitors to parallelism

- Load balancing: distribute the load among processes to keep all of them busy



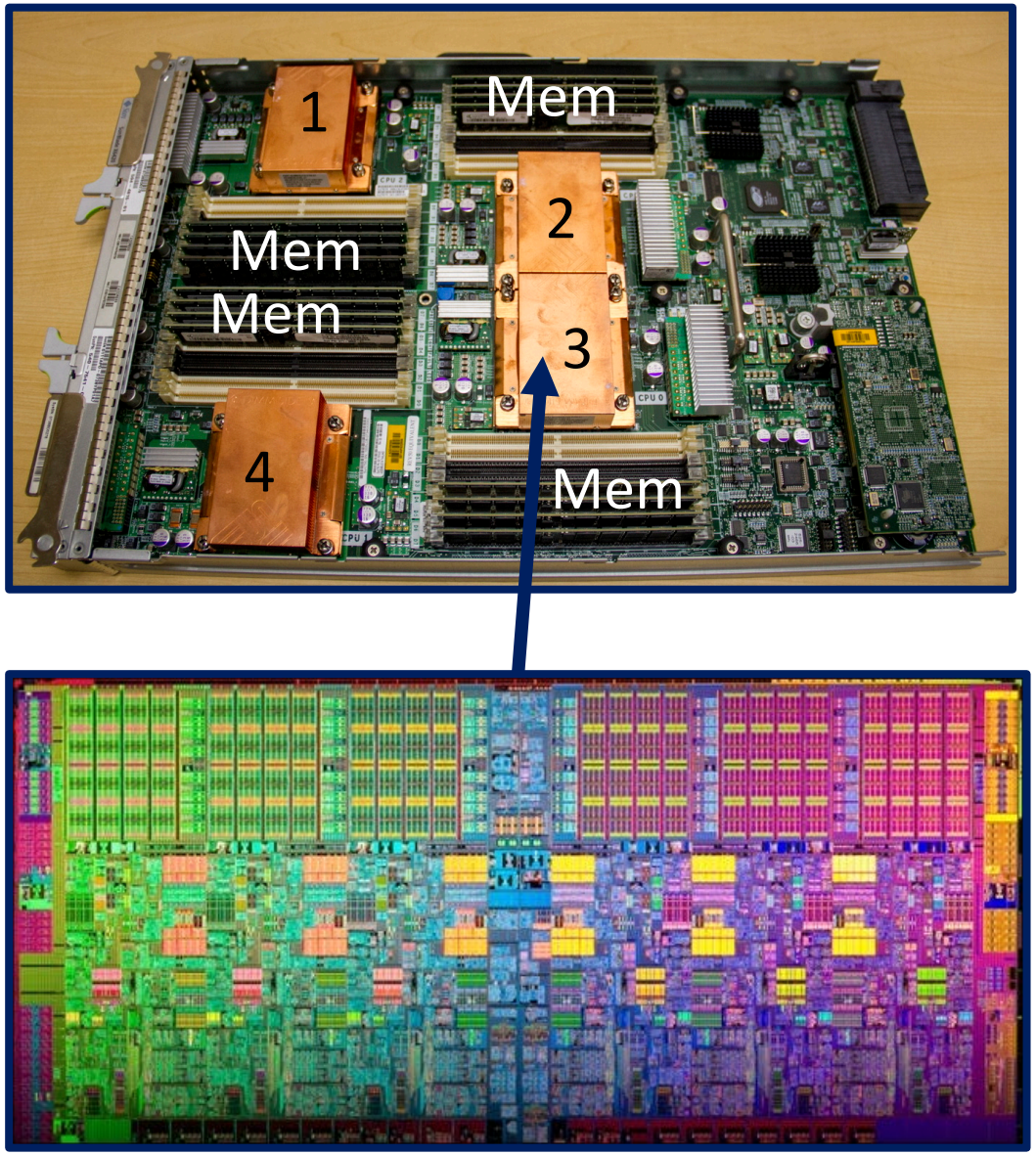
- I/O
  - bad news: I/O operations are generally considered as inhibitors to parallelism
  - good news: parallel file system and parallel I/O programming interface are available

Hardware

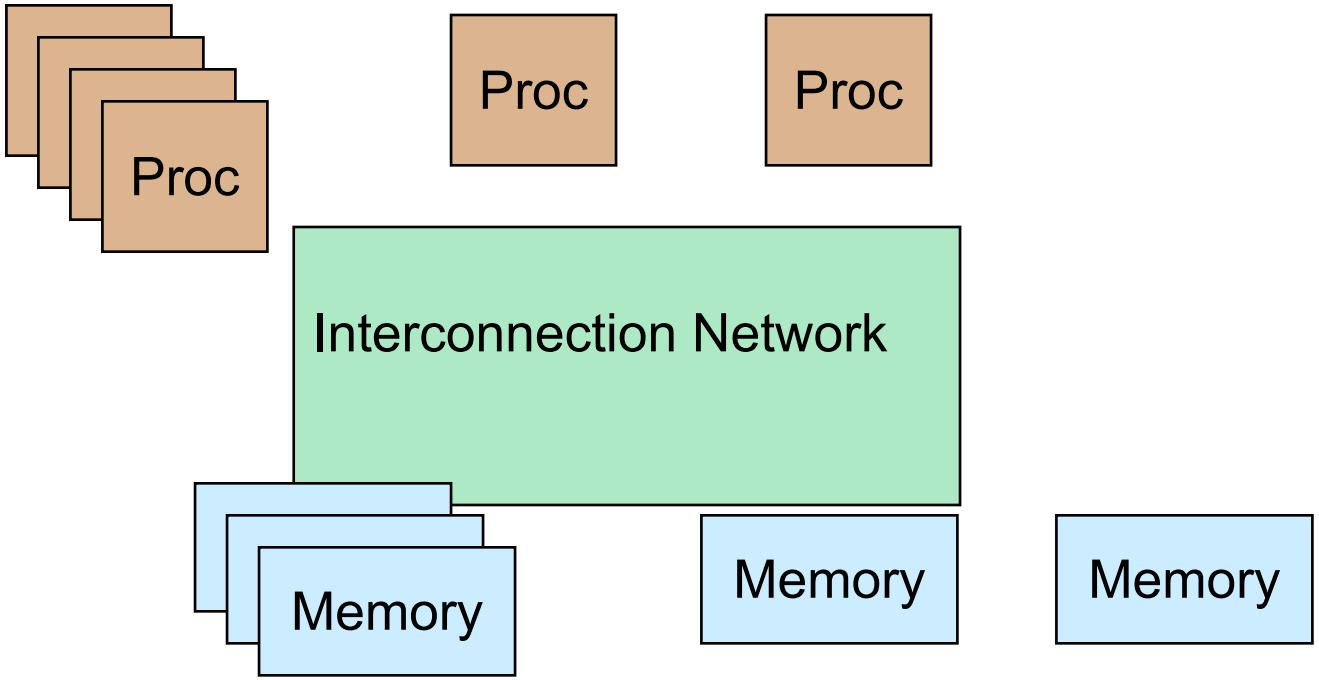
Terminology

Node: A standalone “computer in a box”. Usually comprised of multiple sockets/CPUs.

CPU/Socket/Processor/Core: This varies, depending on whom you talk to. A **CPU** traditionally was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then individual CPUs were subdivided into multiple **cores**, each being a unique execution unit. CPUs with multiple cores are sometimes called **sockets**. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confusing at times.

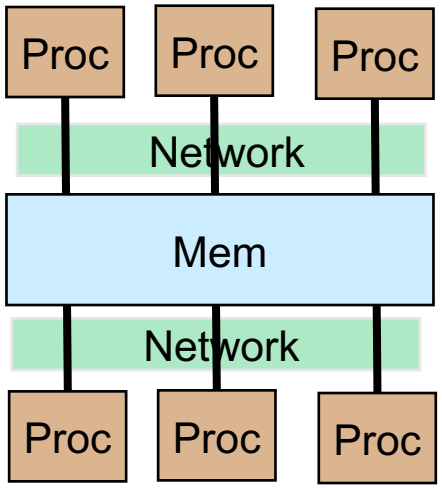


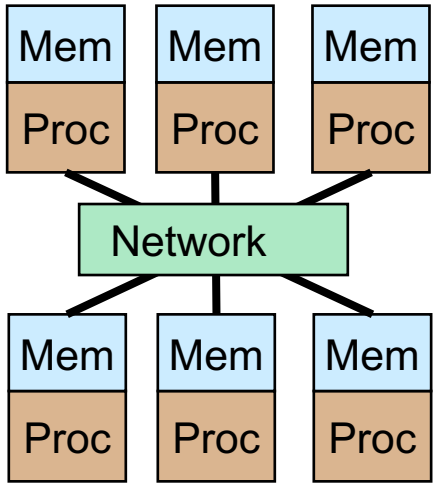
Parallel machines and programming

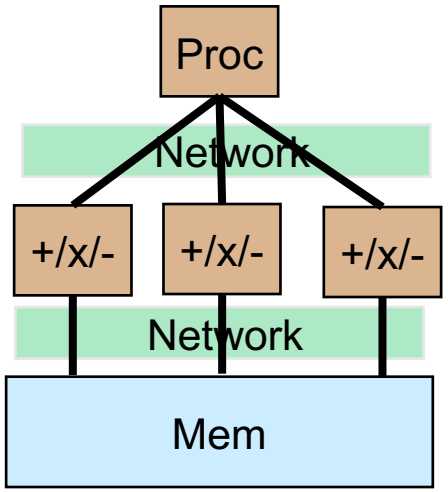


- Where is the memory physically located?
- Is it connected directly to processors?
- What is the connectivity of the network?
- How are the processors controlled?

Parallel machines and programming







Shared Memory	Distributed Memory	Single Instruction Multiple Data (SIMD)
Processors execute own instruction stream	Processors execute own instruction stream	One instruction stream (all run same instruction)
Communicate by reading/writing memory	Communicate by sending messages	Communicate through memory
Cost of a read/write is constant	Message time depends on size, but not location	Assume unbounded # of arithmetic units

These are the natural “abstract” machine models

Terminology

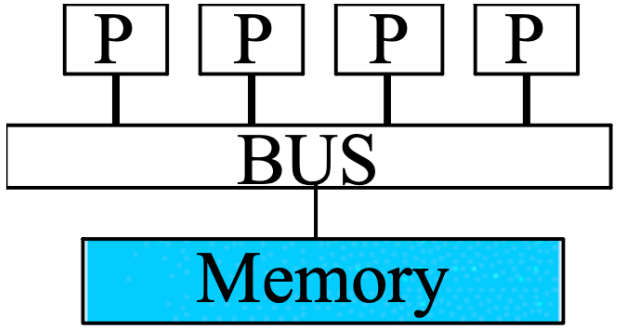
Shared memory: From a hardware point of view, it describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same picture of memory and can directly address and access the same logical memory locations regardless of where the physical memory exists.

Distributed memory: In hardware, it refers to network-based memory access for physical memory that is not common. As a programming model, tasks can only logically see local machine memory and must use communications to access memory on other machines where other taks are executing.

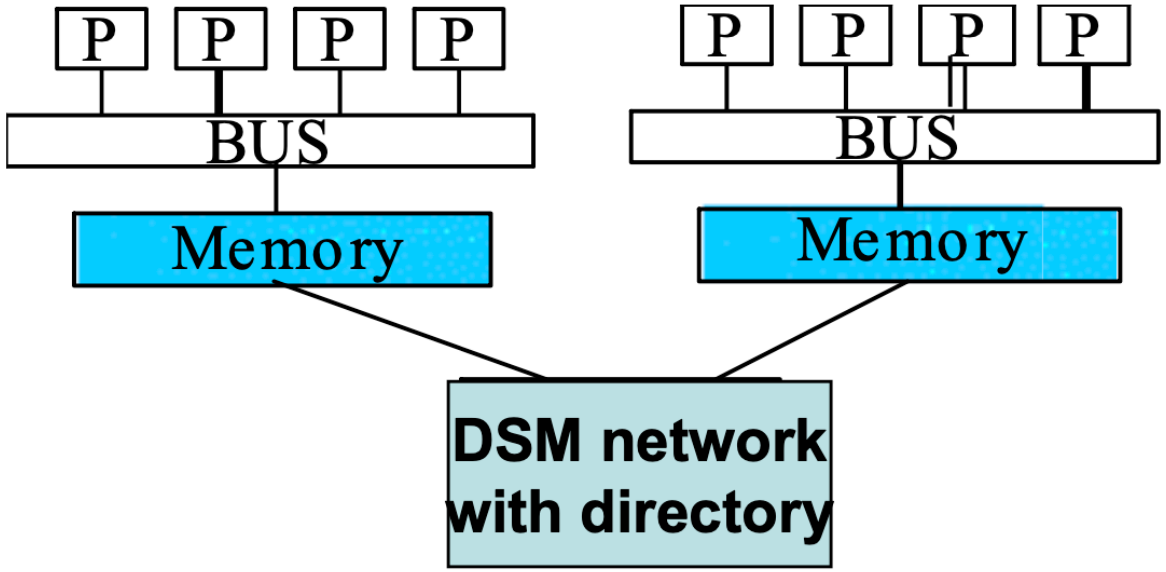
Communications: Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network.

Parallel overhead: The amount of time required to coordinate parallel tasks, as opposed to doing useful work. It includes task start-up time, data communications, task termination time, etc.

Shared memory: UMA and NUMA



Uniform memory access (UMA): each processor has uniform access time to memory – more commonly known as Symmetric multiprocessors, or SMPs.



Non-uniform memory access (NUMA): time for memory access depends on location of data; also known as distributed shared memory systems. Easier to scale than SMPs, but more expensive due to extra directory hardware in network.

Different systems

- SMPs (Symmetric MultiProcessors): easy to program, good price-performance for small number of CPUs; Uniform memory access gives predictable performance.

- cc-NUMAs (cache coherent Non-UMA, distributed shared memory machines): enables larger number of processors and shared memory address space than SMPs. Easy to program, but harder and more expensive to build.
  - Applications can be developed in which vector loop iterations without dependencies are executed by different processors
  - Shared memory codes are mostly data parallel SIMD kinds of codes
  - OpenMP is the most used standard for shared memory programming

- Clusters (Collection of externally interconnected stand-alone computers): maximum scalability. Stand-alone computers are nodes. Programming on it is the most difficult due to multiple address spaces and need to access remote data.
  - local memory accessed faster than remote memory
  - data must be manually decomposed
  - MPI is the standard for distributed memory programming

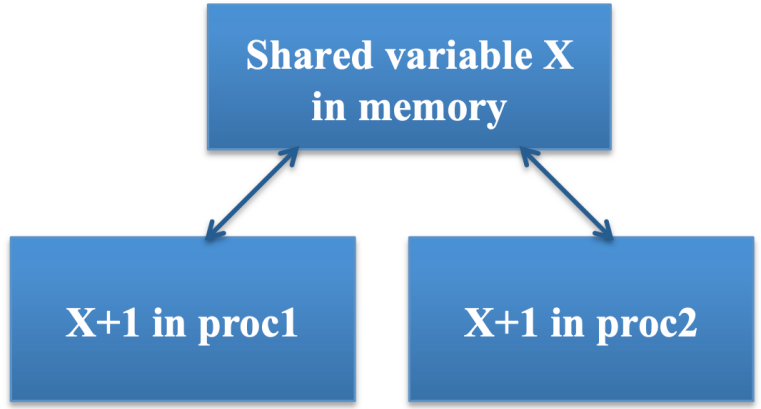
OpenMP

```
!$OMP PARALLEL DO
for(int ii=0; ii<128; ++ii)
{…}
!$OMP END PARALLEL DO
```

- The first directive specifies that the loop immediately following should be executed in parallel
- The second directive specifies the end of the parallel section
- Parallel do can result in significant parallel performance
- It is easy to write an openMP code

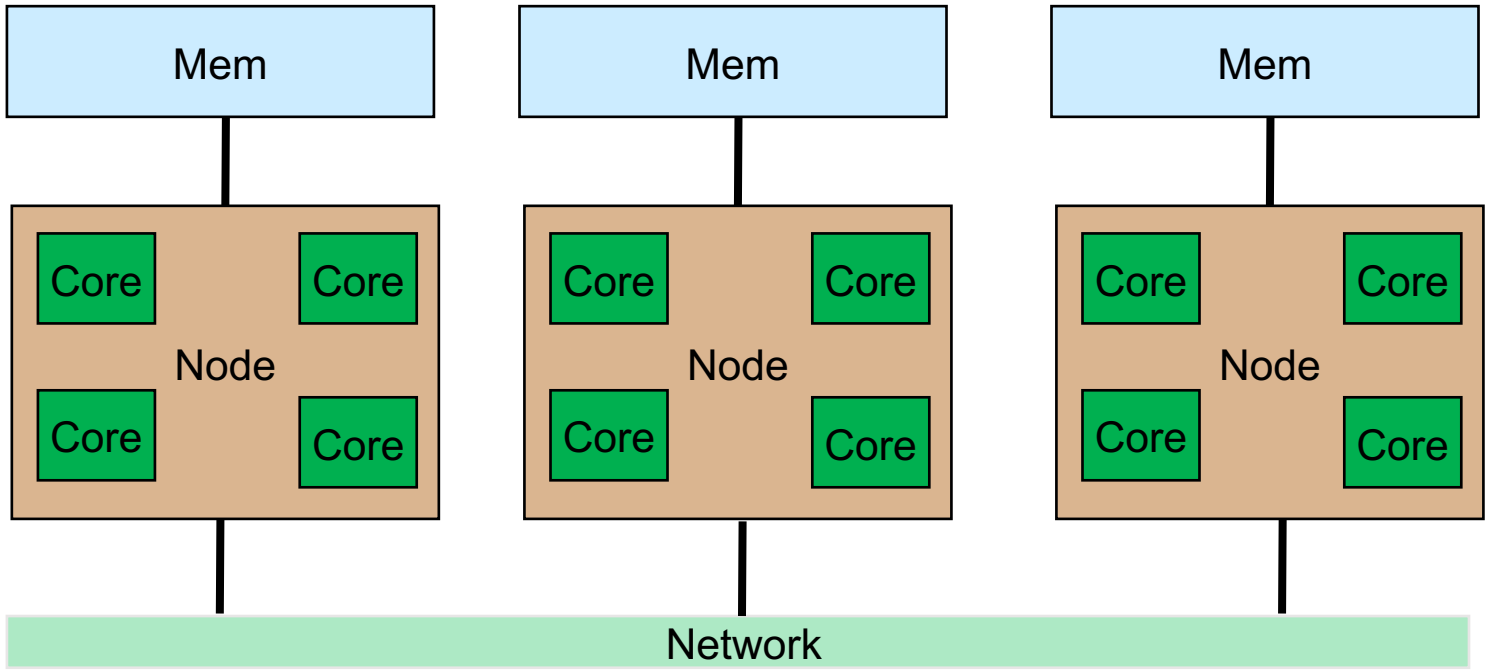
SMPs: momory access problems

- Cache coherence: when a processor modifies a shared variable in local cache, different processors have different value of the variable. Several mechanism have been developed for handling the cache coherence problem
- Cache coherence problem
  - copies of a variable can be present in multiple caches
  - a write by one processor may not become visible to others
  - they will keep accessing state value in their caches
  - need to take actions to ensure visibility or cache coherence



- Implementation
  - when a processor writes a value, a signal is sent over the bus
  - signal is either
    - write invalidate: tell others cached value is invalid and then update
    - write broadcast/update: tell others the new value continuously as it is updated
- System do not scale well: bus-based systems can become saturated; fast crossbars are expensive

Terminology

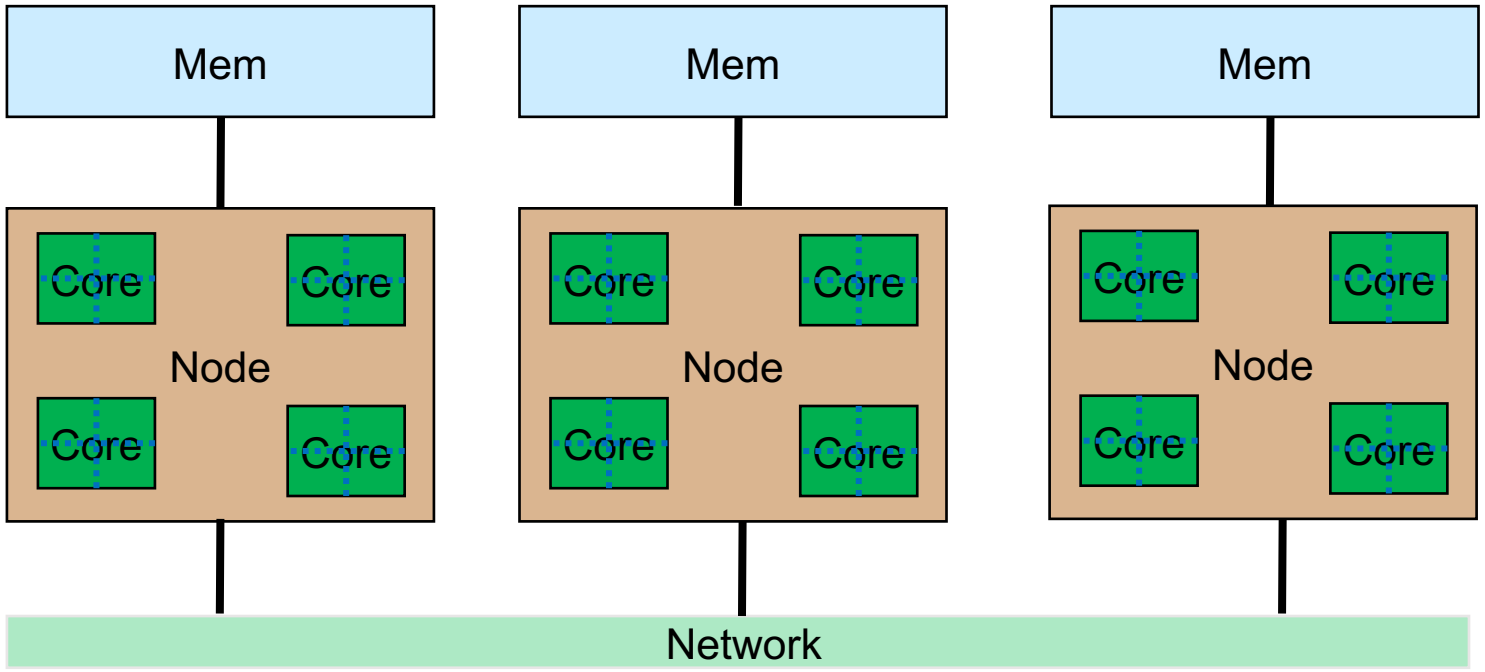


Hybird system: a limited number, say N, of processors have access to a common pool of shared memory

Using more than N processros requires data exchange over a network

Example: cluster with multi-socket blades

Terminology

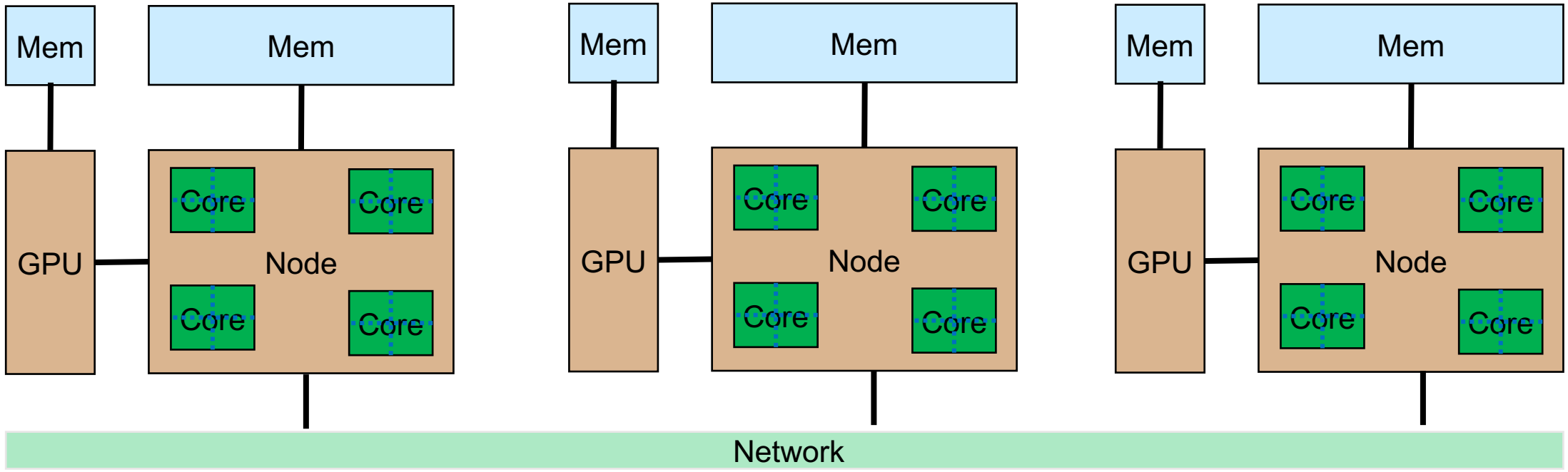


Hybird multicore system: extension of hybird model

Communication over multiple levels

- cache
- main memory
- socket connections
- node-to-node network

Terminology



Heterogeneous system: calculations made in both CPU and other units (GPU, co-processor, etc.)

Load balancing is critical for performance

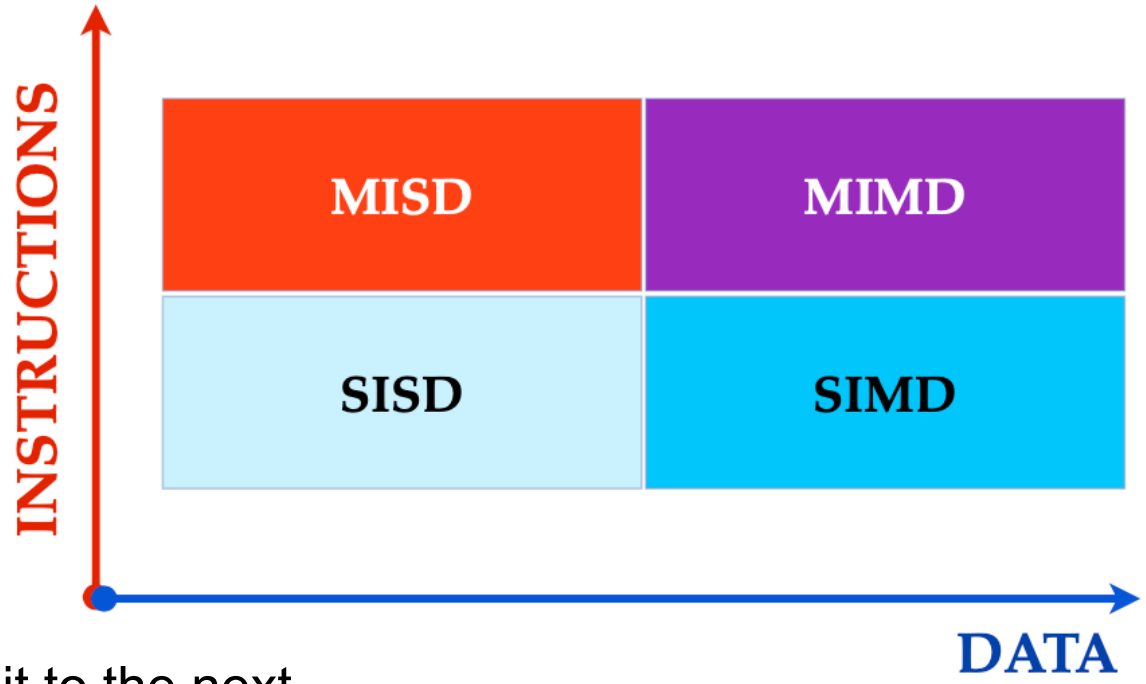
Requires special libraries and compilers (CUDA, OpenCL)

PETSc has enabled CUDA support for its data structures; Kokkos is developed by SNL with the goal of supporting all kinds of heterogeneous systems.

Flynn's taxonomy

- Single Instruction Single Data (SISD):

  rare nowadays
- Single Instruction Multiple Data (SIMD):
  - modern CPUs have SIMD units
  - vector machines
- Multiple Instruction Single Data (MISD)
  - a pipeline of multiple independently executing functional unis operating on a single stream of data, forwarding results from one functional unit to the next
- Multiple Instruction Multiple Data (MIMD)
  - multiple instruction streams and multiple data streams
  - can be further categorized into
    - shared memory
    - distributed memory



SISD

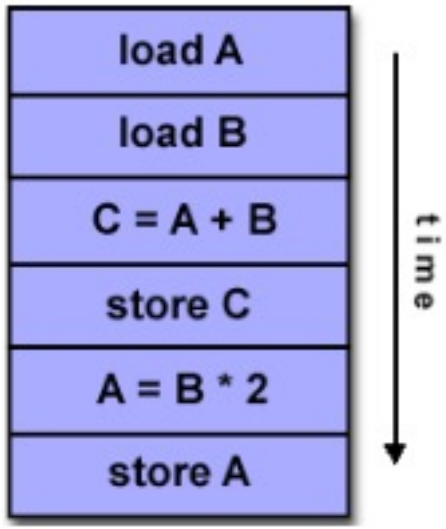
- Single Instruction Single Data (SISD): a serial computer
  - single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
  - single data: only one data stream is being used as input during any one clock cycle

UNIVAC1

IBM360

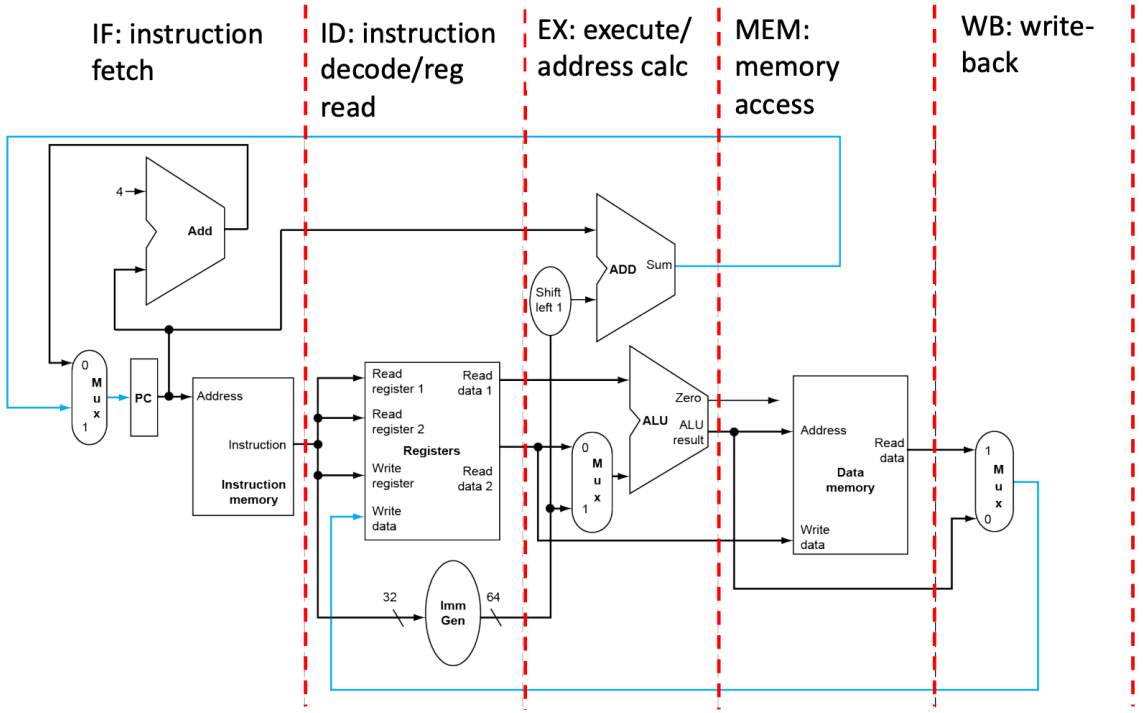
CDC7600

PDP1



MISD

- Multiple Instruction Since Data (MISD):
  - multiple instruction operate independently on a single stream of data, forwarding results from one functional unit to the next
  - data is different after processing by each stage in the pipeline
  - used most often inside processing units
  - rare to see in modern clusters

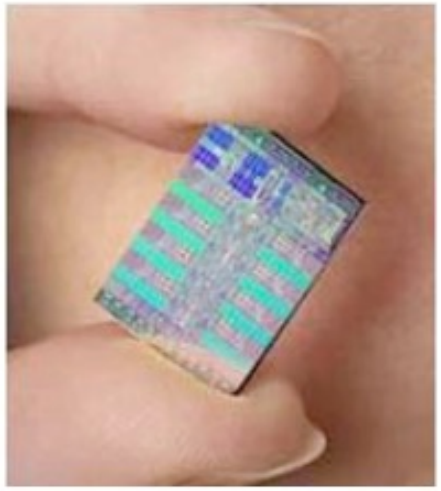


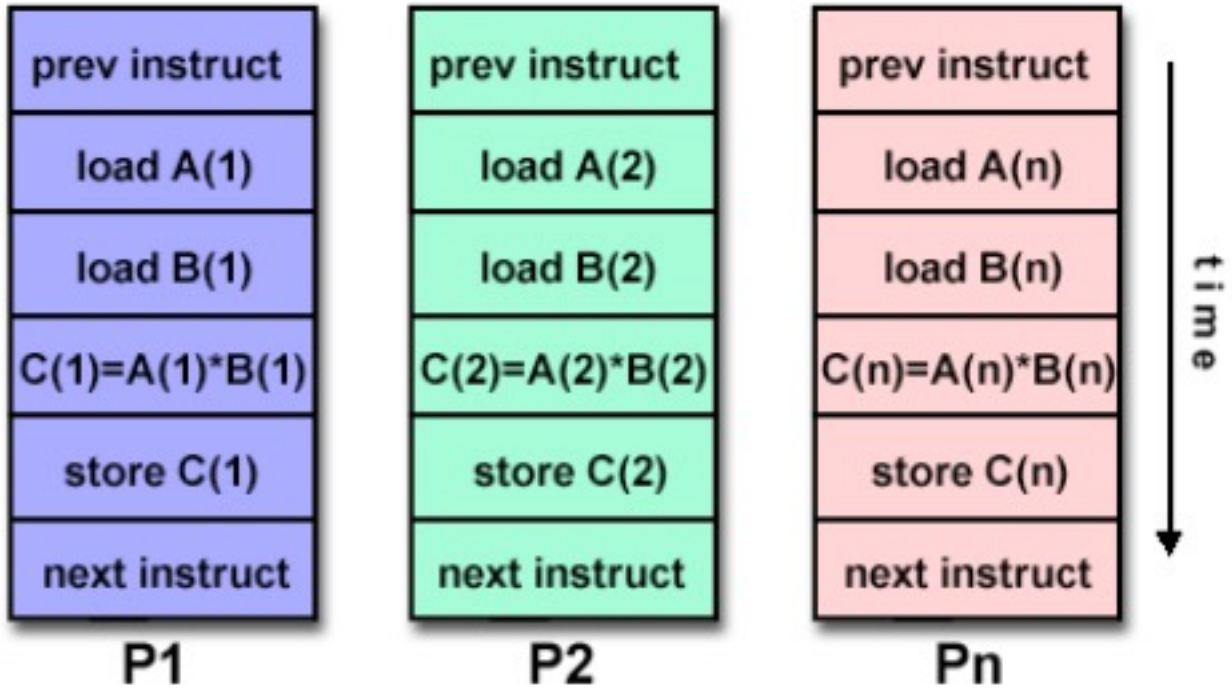
SIMD

- Single Instruction Multiple Data (SIMD):
  - single instruction: all processing units execute the same instruction at any given clock cycle
  - multiple data: processing units operate on different data input
  - best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing
  - all CPUs today are SIMD inherently

Cray Y-MP

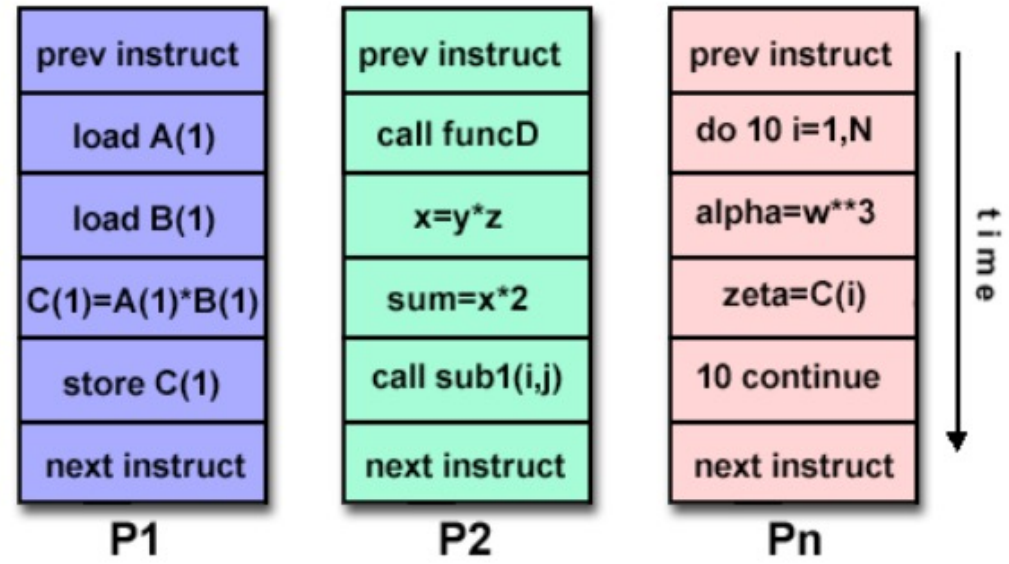
Cell Processor





MIMD

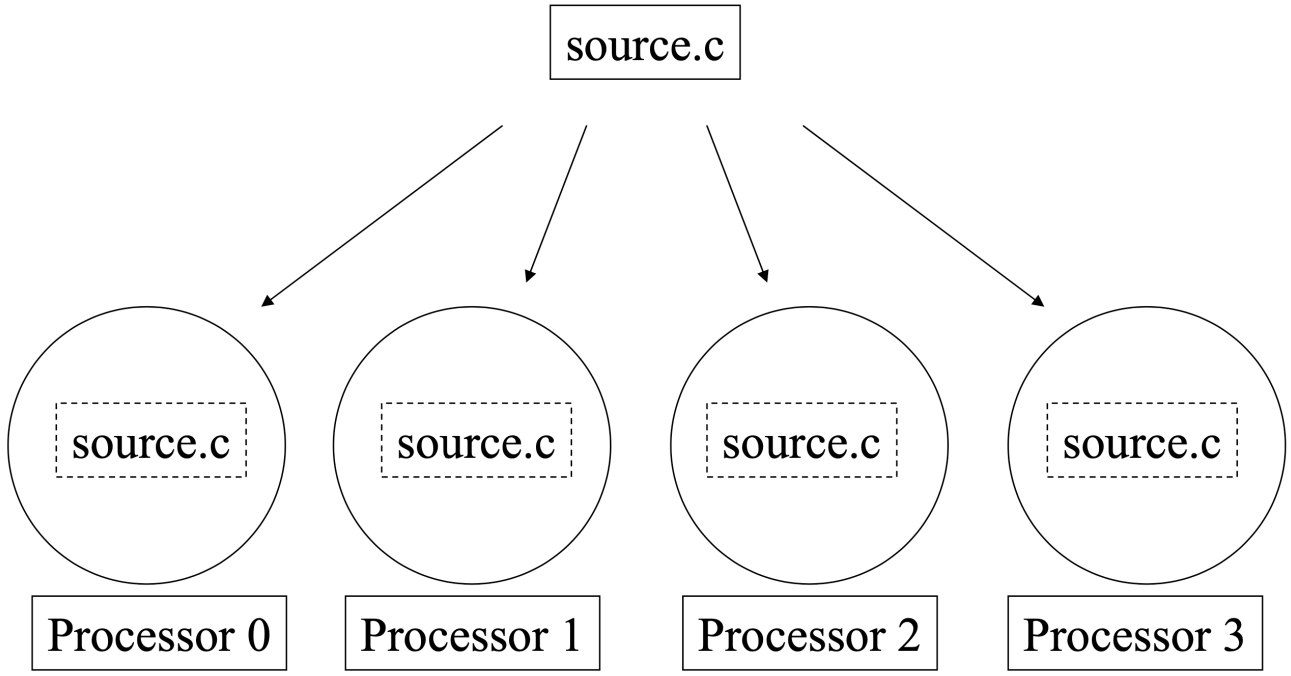
- Multiple Instruction Multiple Data (MIMD):
  - multiple instruction: each processor may be executing a different instruction stream
  - multiple data: every processor may be working with a different data stream
- Currently the most common type of parallel computer
- Many MIMD architecture also include SIMD execution sub-units.



Programming models

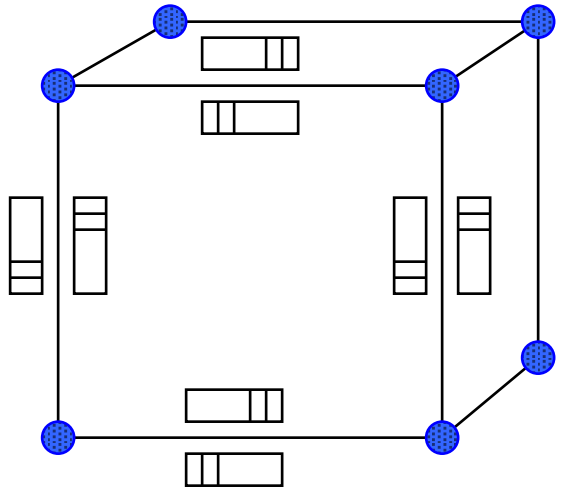
- SPMD: dominant programming model for shared and distributed memory machines
  - One source code is written
  - Code can have conditional execution based on which processor is executing the copy
  - All copies of code are started simultaneously and communicate and synch with each other periodically

- MPMD: more general
  - Multiple program: tasks may execute different program simultaneously.
  - Multiple data: all tasks may use different data
  - Not as common as SPMD model, but may be suited for certain types of problems.



Network

- Early distributed memory machines were:
  - a collection of microprocessors
  - communication was performed using bi-directional queues between nearest neighbors.
- There was a strong emphasis on topology in algorithm in order to minimize the traffic time.
- Networks are like streets:
  - link = street
  - Switch = intersection
  - Distances = number of blocks travelled
  - Routing algorithm = travel plan
- Properties:
  - Latency: how long to get between nodes in the network
    - time for one car: distance / speed
  - Bandwidth: how much data can be moved per unit time
    - number of lanes and car density per lane
    - limited by the bit rate per wire and number of wires



Design characteristics of a network

- **Topology** (how things are connected)
  - Crossbar; ring; 2D, 3D, higher-D mesh; hypercube; tree; butterfly; perfect shuffle; *dragon fly*, …
- **Switching strategy**
  - Circuit switching: full path reserved for entire message, like the telephone
  - Packet switching: message broken into separately routed packets, like the post office
- **Flow control** (what if there is a congestion)
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node temporarily halt.

Performance of a network: latency

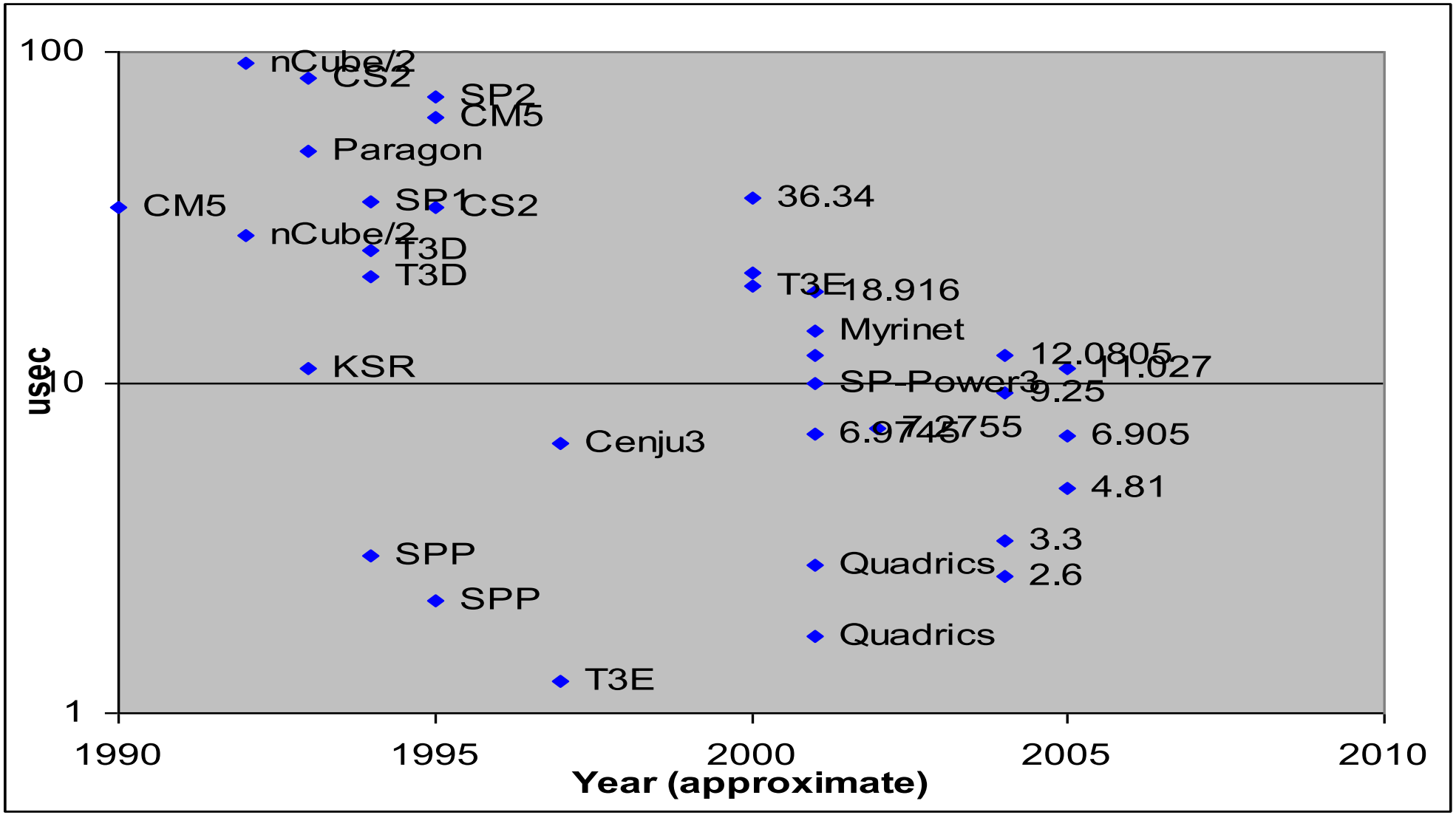
- **Diameter**: the maximum of the shortest path between a given pair of nodes

- Latency: delay between send and receive times
  - Latency tends to vary widely across architectures
  - Vendors often report hardware latencies
  - Application programmers care about software latencies (packing messages, copying, unpacking, etc.)

- Observations:
  - Latencies differ by 1-2 orders across network designs
  - Software/hardware overhead at source/destination dominates cost (usec)
  - Hardware latency varies with distance, but is small compared to overheads (10 to 100 nsec)

- Latency is critical for programs with many small messages.

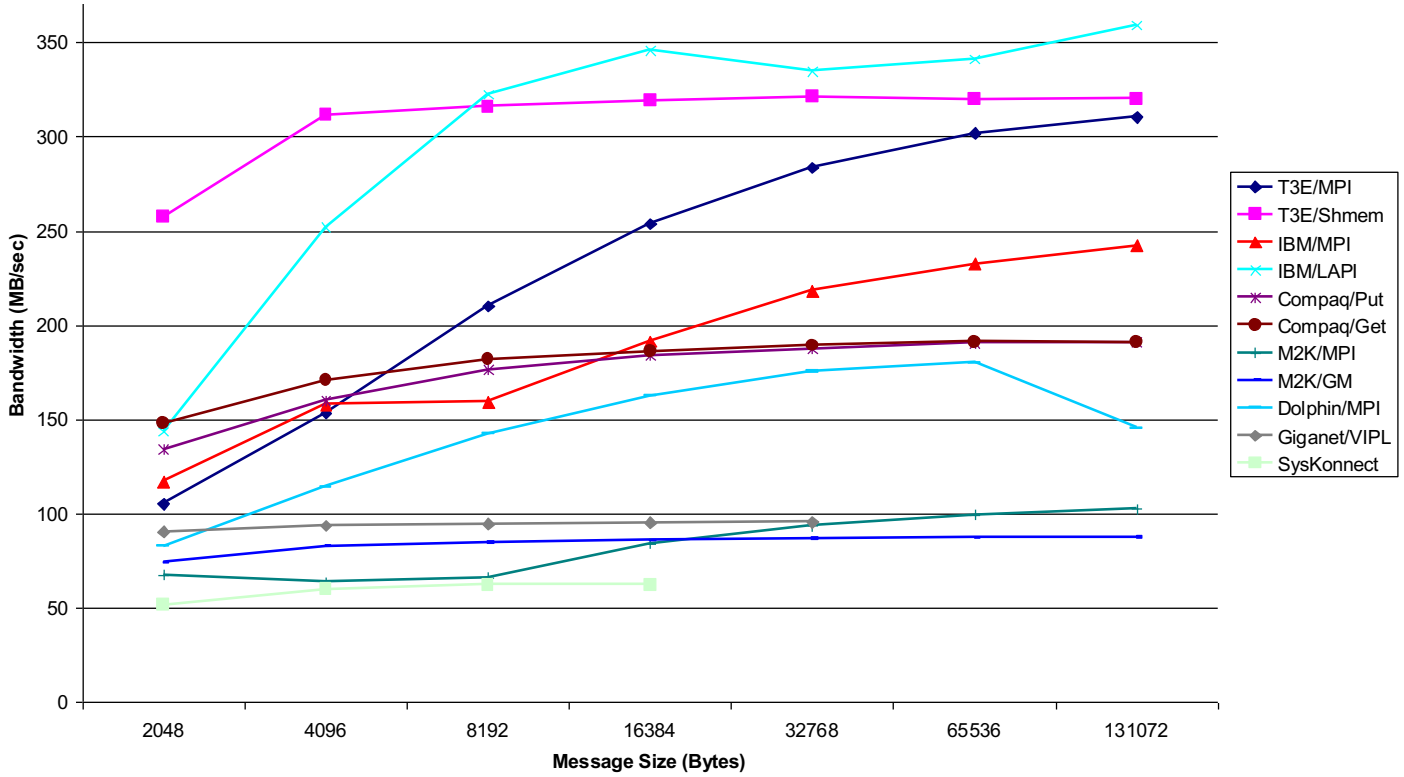
End-to-end latency over time

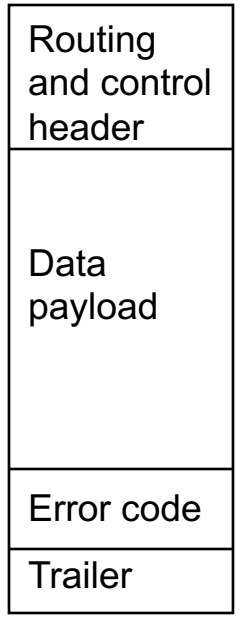


- Latency has not improved significantly

Performance of a network: bandwidth

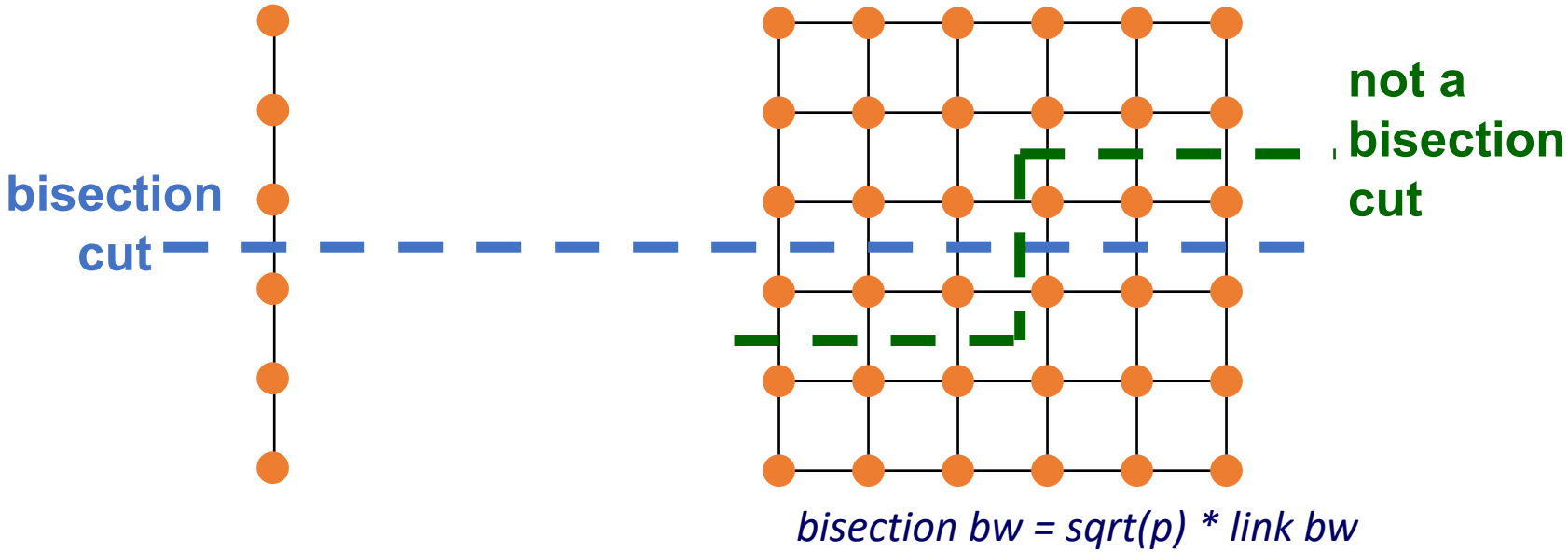
- Bandwidth unit is typically in Gigabytes per sec (GB/s)
- Effective bandwidth is usually lower than physical link bandwidth due to packet overhead
- Bandwidth is critical for applications with mostly large messages.





Performance of a network: bandwidth

- Bisection bandwidth: bandwidth across smallest cut that divides network into two equal halves
- Bandwidth across "narrowest" part of the network



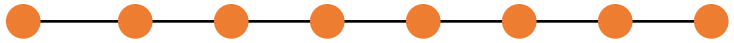
- Bisection bandwidth is important for algorithms in which all processors need to communicate with all others

Network topology

- Network topology is key for minimize cost in node-to-node communication

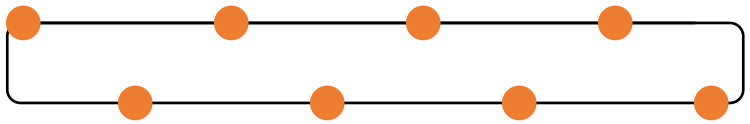
- Example: on IBM SP system, hardware latency varies from 0.5 us to 1.5 us, but user-level message passing latency is roughly 36 us.

- Linear array



  - Diameter n-1; average distance n/3
  - Bisection bandwidth = 1 in units of link bandwidth

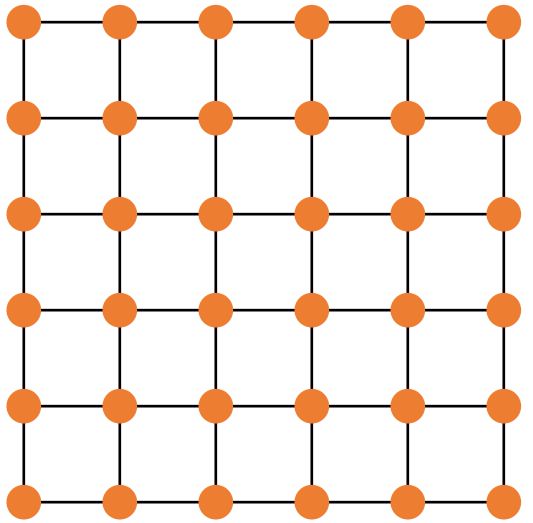
- Torus or Ring



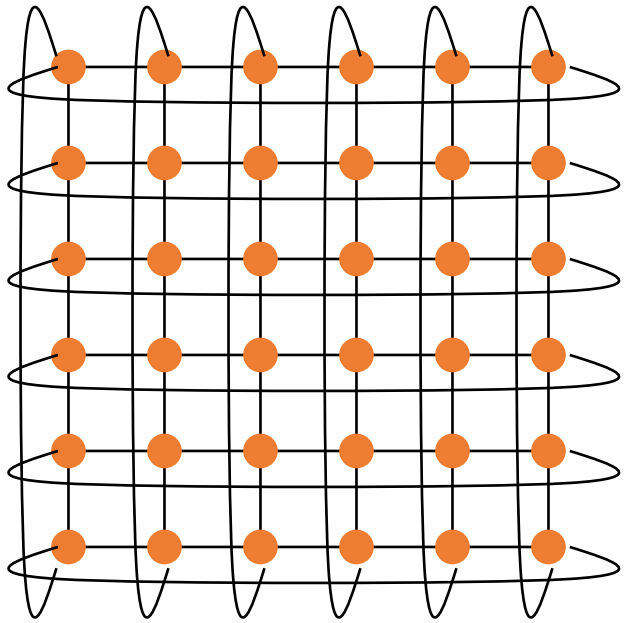
  - Diameter n/2; average distance n/4
  - Bisection bandwidth = 2
  - First improvement over a linear array or bus

Network topology

- Two-dimensional mesh
  - Diameter $2\sqrt{n} - 2$
  - Bisection bandwidth = $\sqrt{n}$

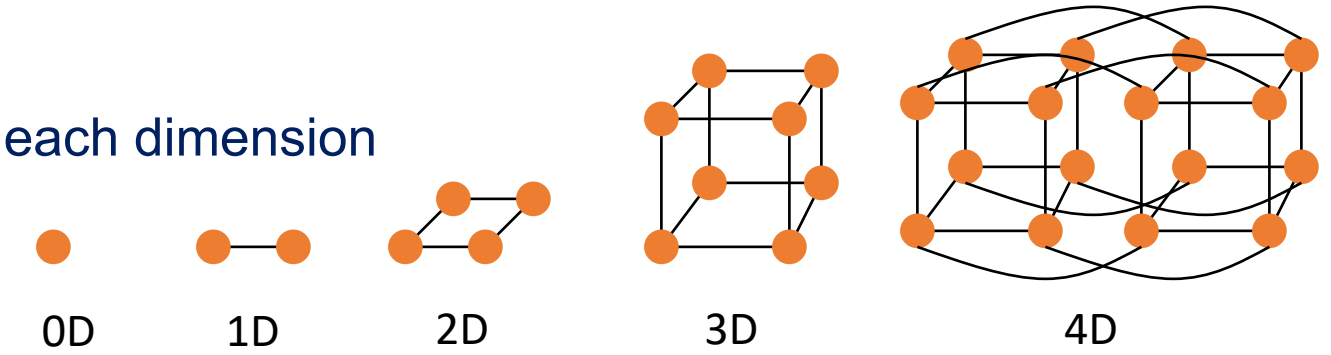


- Two-dimensional torus
  - Diameter $\sqrt{n}$
  - Bisection bandwidth = $2\sqrt{n}$

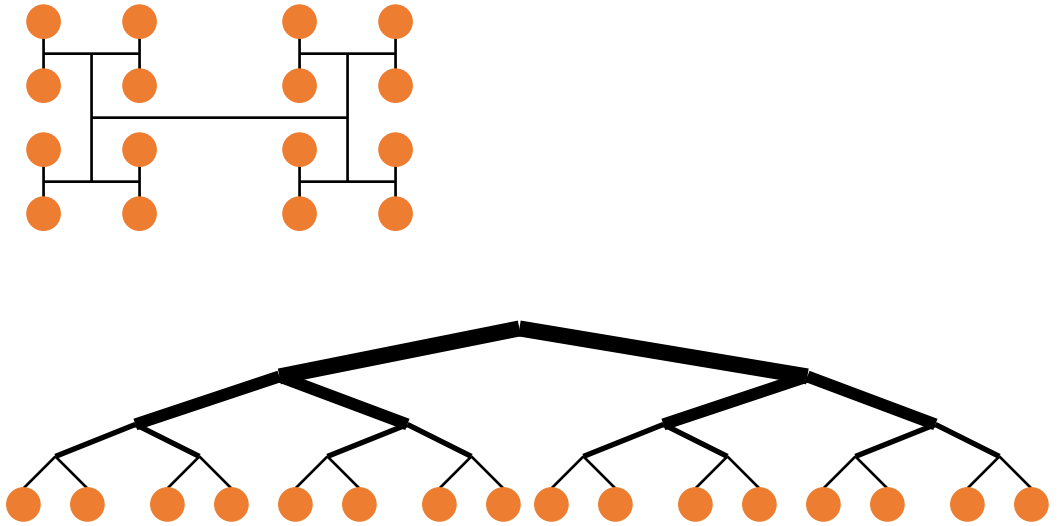


Network topology

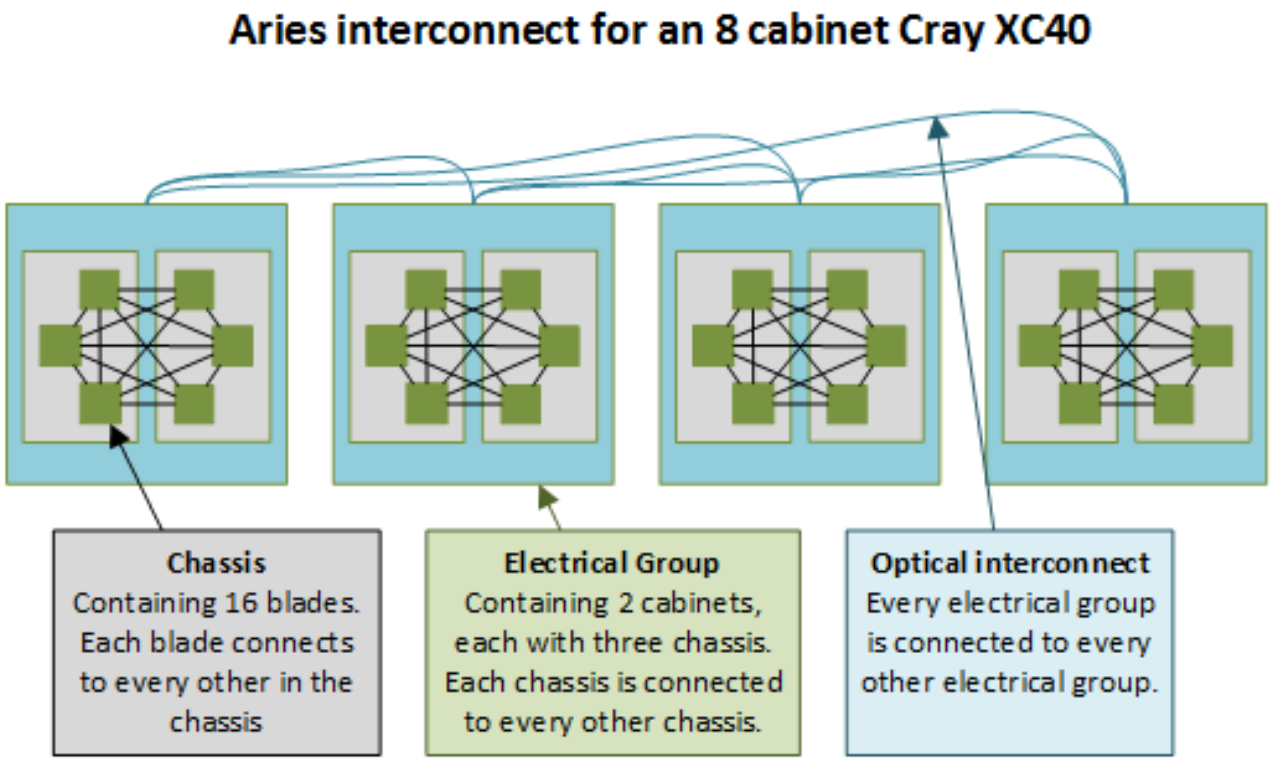
- Hypercubes:
  - multidimensional with 2 proc in each dimension
  - Number of nodes 2^d
  - Diameter = d
  - Bisection bandwidth 2^{d-1}
  - Popular in early machines

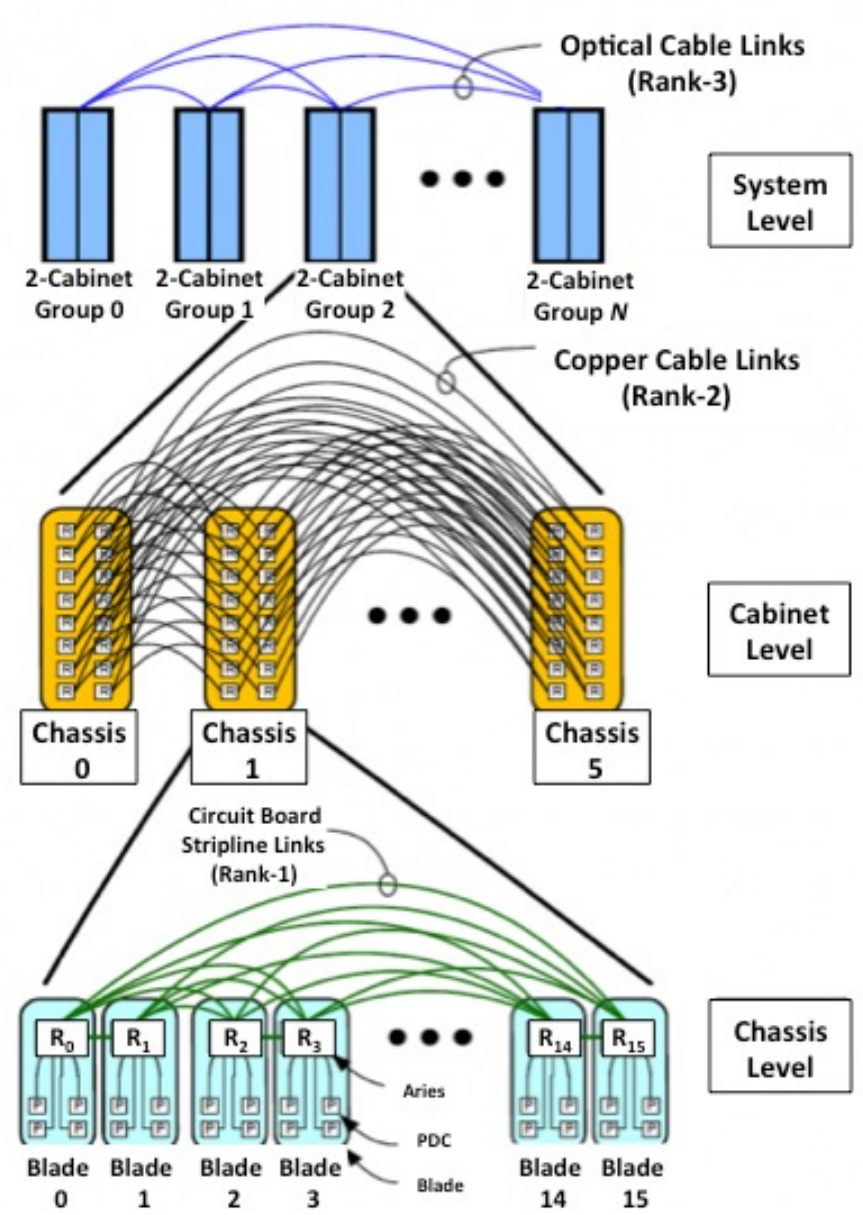


- Trees
  - Diameter $\log(n)$
  - Bisection bandwidth = 1
  - Fat tress avoid bisection bandwidth issue
    - More or wider links near top



Dragonflies





Performance models

Outline

- Amdahl's Law – strong scaling
  - fixed problem size
  - how much does parallelism reduce the execution time of a problem
- Gustafson's Law – weak scaling
  - Fixed execution time
  - How much longer does it take for the problem without parallelism

Amdahl's Law

Example :

Suppose you want to perform simultaneously two sums: sum of 10 scalar variables and matrix sum with dimension 10-by-10.

Assume the time for an addition is t

Time for 1 processor is 100 t + 10 t = 110 t .

Time for 10 processor is 100 t /10 + 10 t = 20 t. Speedup is 110/20 = 5.5 (out of 10).

Time for 100 processor is 100 t/100 + 10 t = 11 t. Speedup is 110/11 = 10 (out of 100).

Amdahl's Law

Example :

Suppose you want to perform simultaneously two sums: sum of 10 scalar variables and matrix sum with dimension 100-by-100.

Assume the time for an addition is t

Time for 1 processor is 10000 t + 10 t = 10010 t .

Time for 10 processor is 10000 t /10 + 10 t = 1010 t. Speedup is 10010/1010 = 9.9 (out of 10).

Time for 100 processor is 10000 $t/100$ + 10 t = 110 t. Speedup is 10010/110 = 91 (out of 100).

Amdahl's Law

Example :

Suppose you want to perform simultaneously two sums: sum of 10 scalar variables and matrix sum with dimension 100-by-100.

Time for 100 processor is 10000 $t/100$ + 10 t = 110 t. Speedup is 10010/110 = 91 (out of 100).

If one has 2% of the load, it must do 200 additions.

The rest 99 processor shares 98% of the job.

Execution time is max(9800 t /99, 200 t) + 10 t = 210 t.

Speedup drops to 10010/210 = 48 (out of 100).

Parallel performance

- Speedup = sequential execution time / parallel execution time

- The operations in a parallel code include
  - Sequential computations
  - Parallel computations
  - Parallel overhead (e.g. communications)

Parallel performance

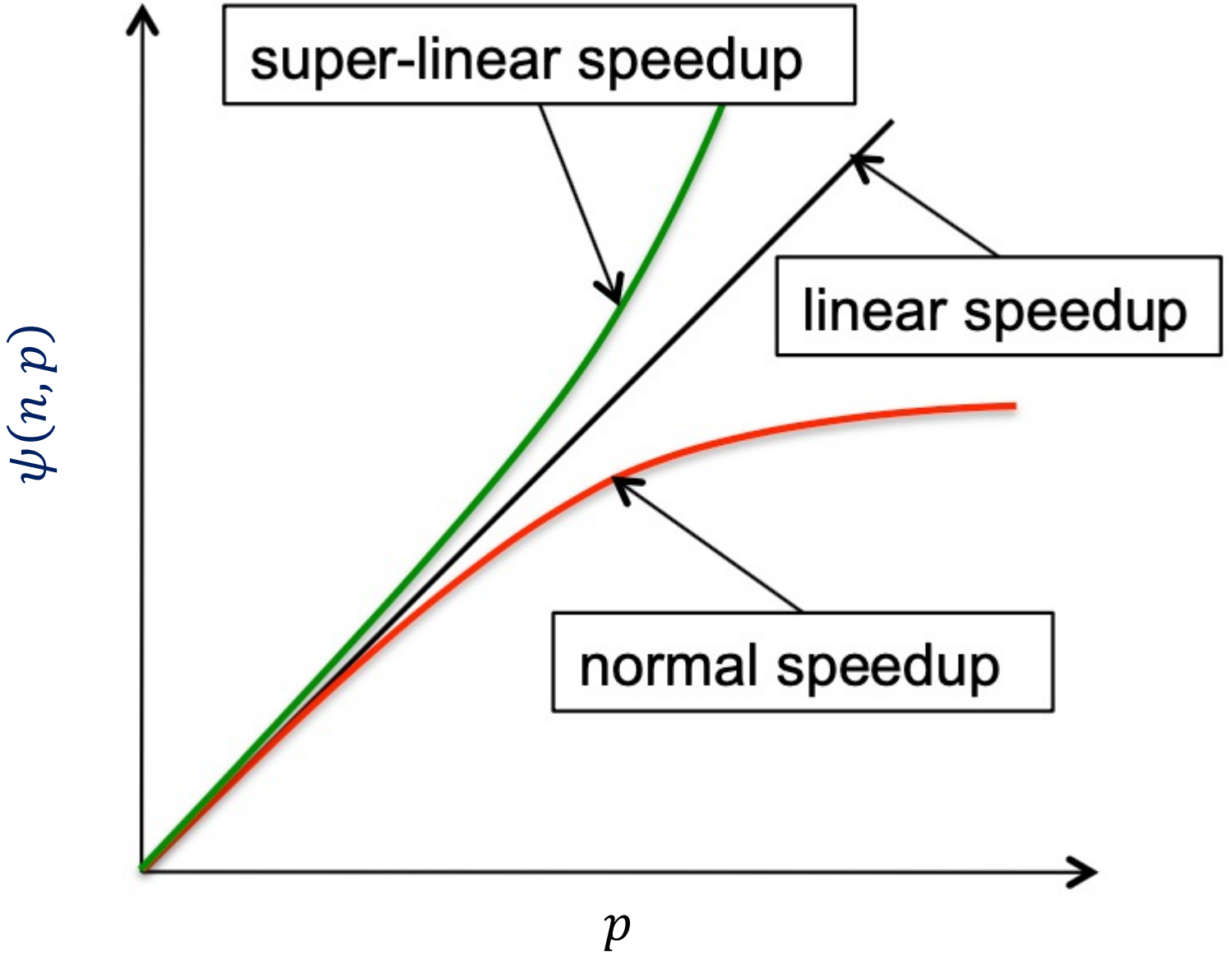
- Let
  - $\psi(n,p)$ be the speedup of solving problem of size n on p processors
  - $\sigma(n)$ be the inherently serial portion of the computation
  - $\varphi(n)$ be portion of computation that can be executed in parallel
  - $\kappa(n,p)$ be parallel overhead time

Then

- $\sigma(n) + \varphi(n)$ is the time required for all inherently sequential code
- $\varphi(n)/p$ is the best possible execution time of a parallel code
- the speedup is now

$$\psi(n,p) = \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n,p)}$$

Parallel performance



Efficiency

- Efficiency $\varepsilon = \dfrac{\text{sequential execution time}}{\text{parallel execution time} \times \text{processors used}}$

- Then the efficiency on a problem of size n on p processors is

$$\varepsilon(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\left(\sigma(n)+\dfrac{\varphi(n)}{p}+\kappa(n,p)\right)p}$$

$$= \frac{\sigma(n)+\varphi(n)}{p\sigma(n)+\varphi(n)+p\kappa(n,p)}$$

Amdahl's Law

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)} \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}}$$

- Now let f denote the inherently serial portion of the computation:

$$f = \frac{\sigma(n)}{\sigma(n)+\varphi(n)}$$

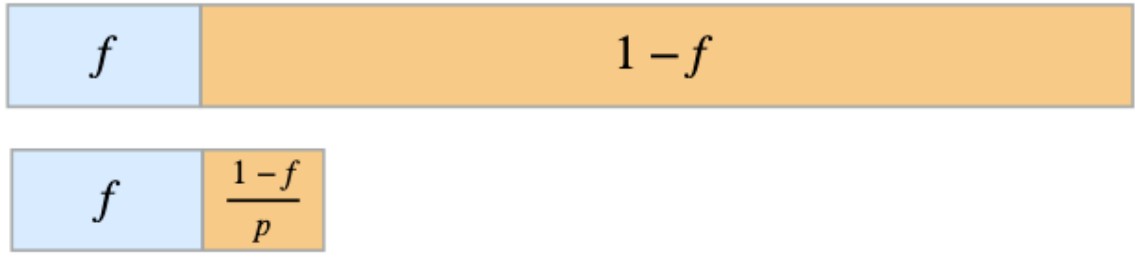
$$\psi(n,p) \le \frac{\frac{\sigma(n)}{f}}{\sigma(n)+\frac{\sigma(n)\left(\frac{1}{f}-1\right)}{p}} = \frac{1}{f+(1-f)/p}$$

- The speedup limit is bounded by f.

Amdahl's Law: practical limits

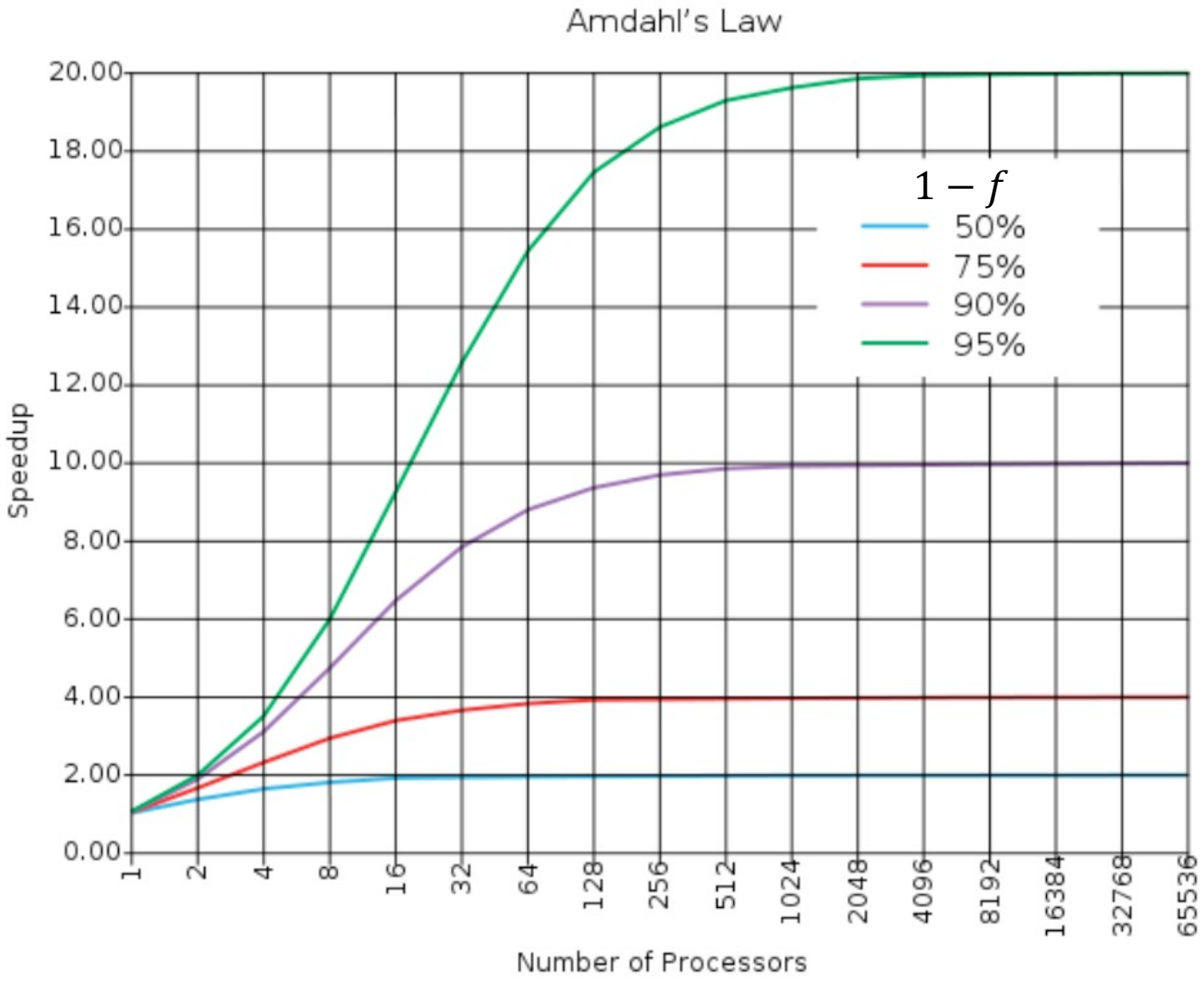
Implicit assumptions in Amdahl's law:

- fixed problem size
  - sometimes we want to keep the execution time constant and increase the problem size
- negligible communication cost
  - the number of processors should be small
- All-or-None parallelism
  - A more realistic model will be $\frac{1}{f_1+\sum_{i=2}^{p}\frac{f_i}{i}}$

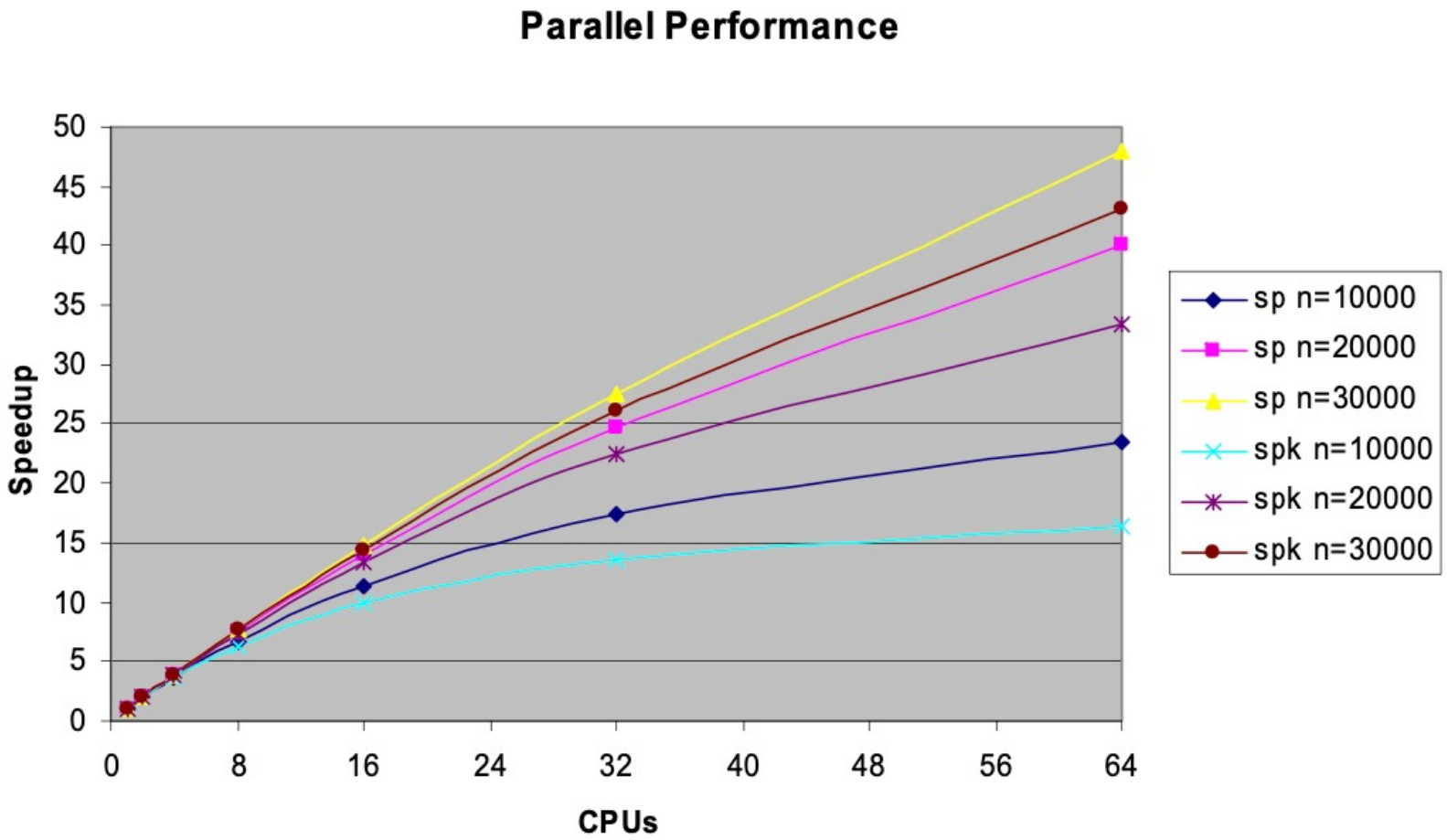


Problems for which those assumptions are reasonable can use this model for performance analysis. Such analysis is called Strong or Fixed-size Scaling: $\psi(n,p) \leq \frac{1}{f+(1-f)/p}$

Amdahl's Law


Amdahl's Law
Speedup
Number of Processors
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
$1 - f$
50%
75%
90%
95%

Amdahl's Law



$$\sigma = n + 18000$$

$$\varphi = 0.01n^2$$

$$\kappa(n,p) = \frac{n}{10} + 10000\log(p)$$

sp uses σ and φ
spk uses σ, φ, and κ

As the problem size increases, the fraction $f = \frac{\sigma(n)}{\sigma(n)+\varphi(n)}$ decreases.

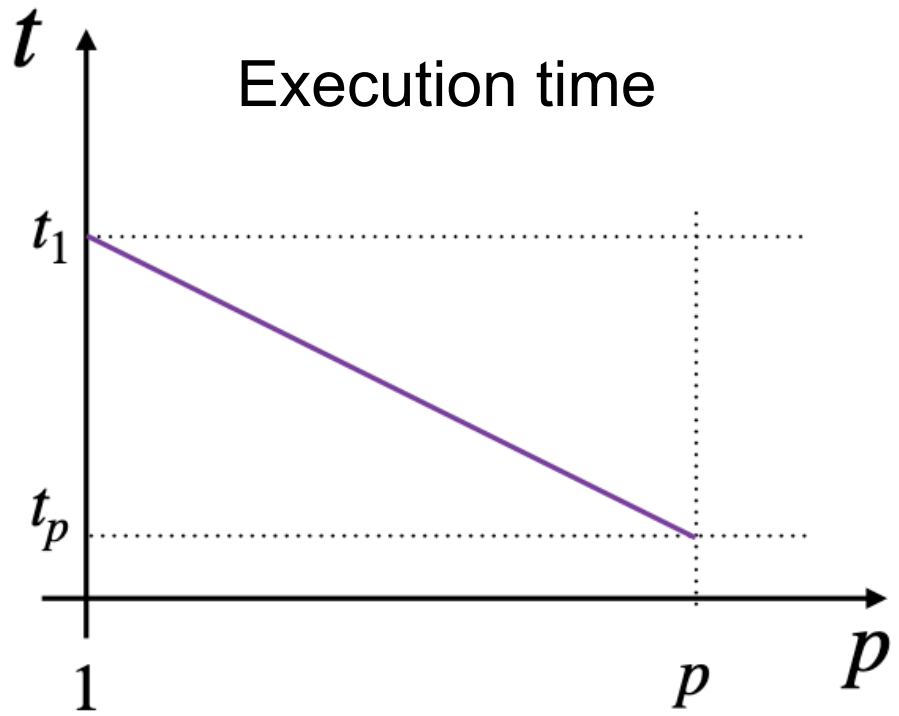
Amdahl effect: increasing problem size increases efficiency or speedup.

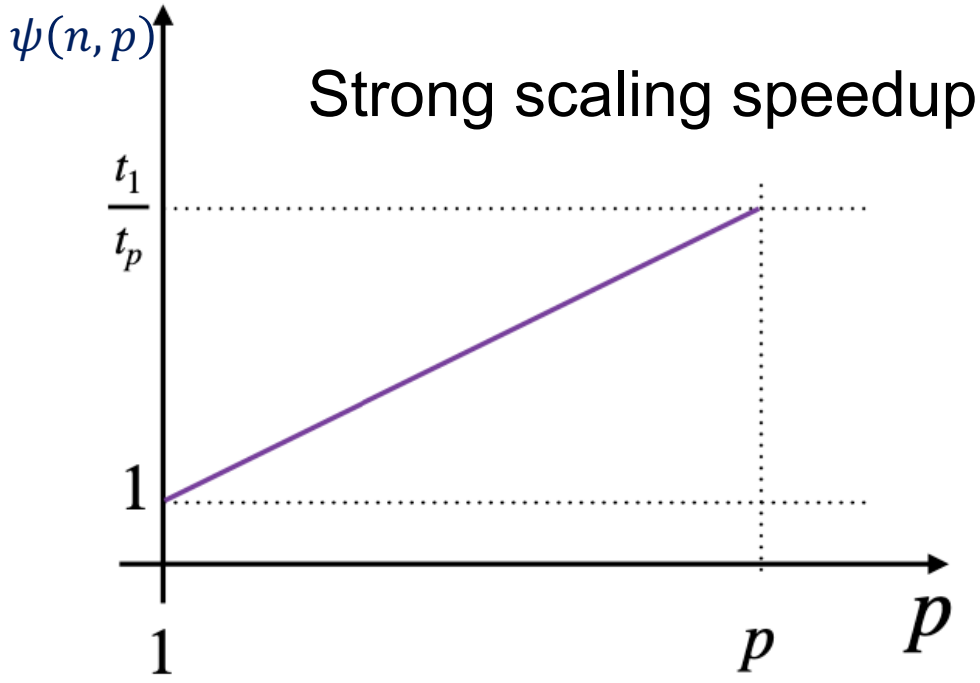
Amdahl's Law: strong scaling analysis

Problem size n is fixed

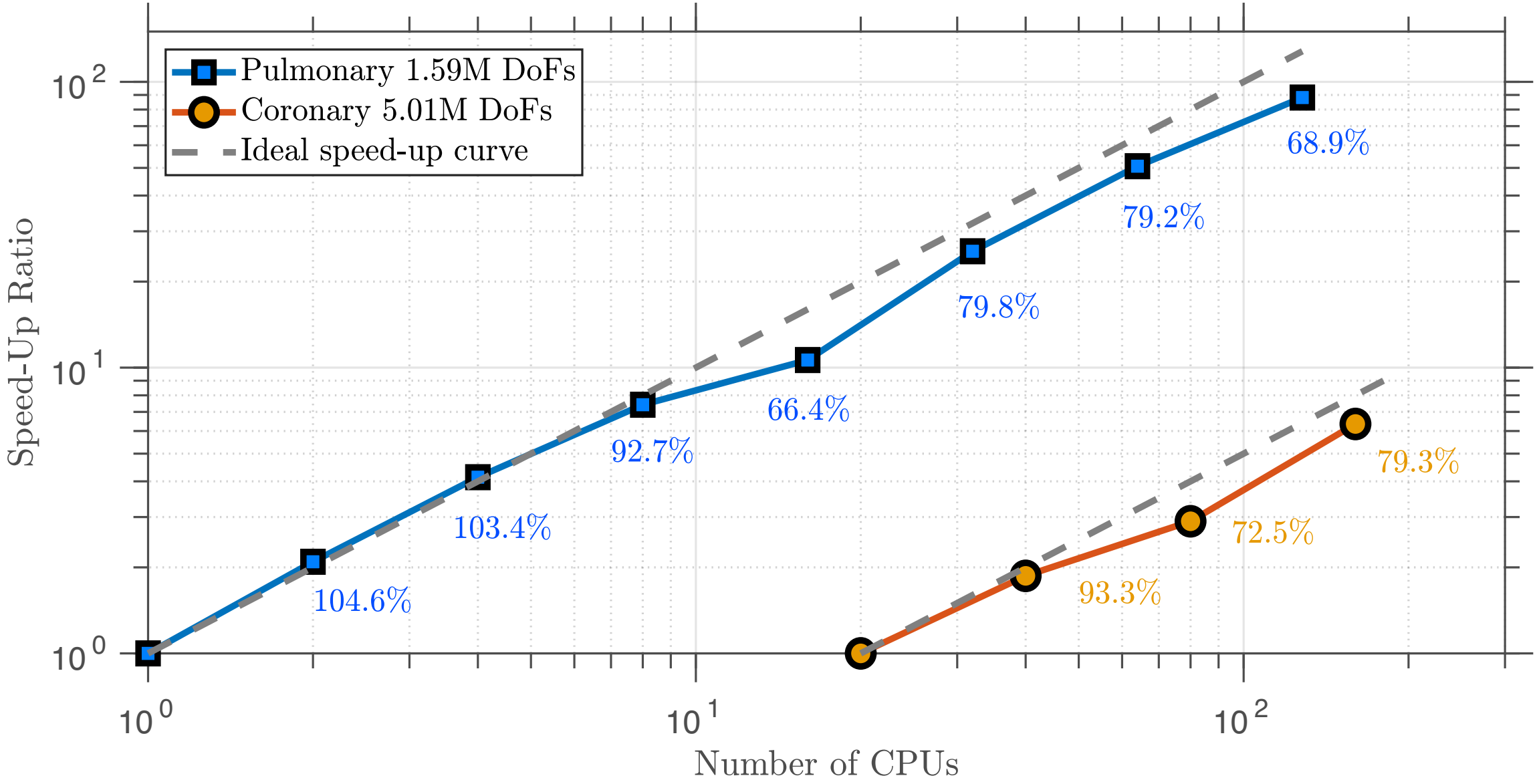
Strong scaling speedup $\psi(n,p) = \frac{t_1}{t_p}$

Presenting strong scaling data:





Parallel performance



Amdahl's Law

Example 1:

Suppose we are trying to determine if it is worthwhile to develop a parallel version of a program. Benchmarking reveals that 90% of the execution time is spent inside functions that can be run in parallel, and 10% of the execution time is spent on functions that can only be run on a single processor. What is the maximum speedup that we may expect from a parallel version using 8 processors?

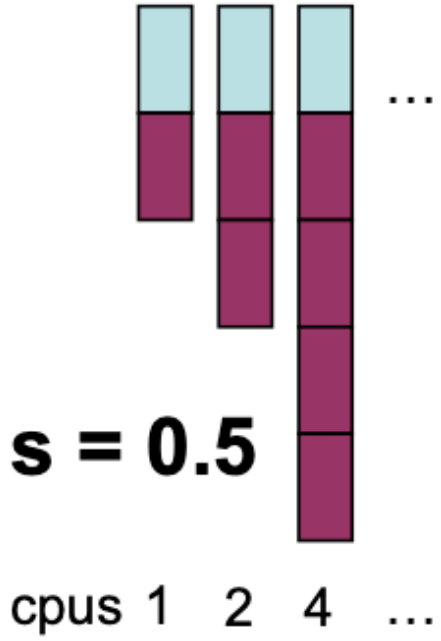
Example 2:

If 25% of the operations in a program must be run on a single processor, what is the maximum speedup achievable?

Gustafson's Law

- Let s denote the fraction of time spent in the parallel computation performing inherently sequential operations

Then



and

$$s = \frac{\sigma(n)}{\sigma(n) + \frac{\varphi(n)}{p}} = \frac{\sigma}{\sigma + \frac{\varphi(n)}{p}}$$

$$\sigma = \left(\sigma + \frac{\varphi(n)}{p}\right) s$$

$$\varphi\,(n) = \left(\sigma + \frac{\varphi(n)}{p}\right)(1 - s)p$$

Implicit assumptions:

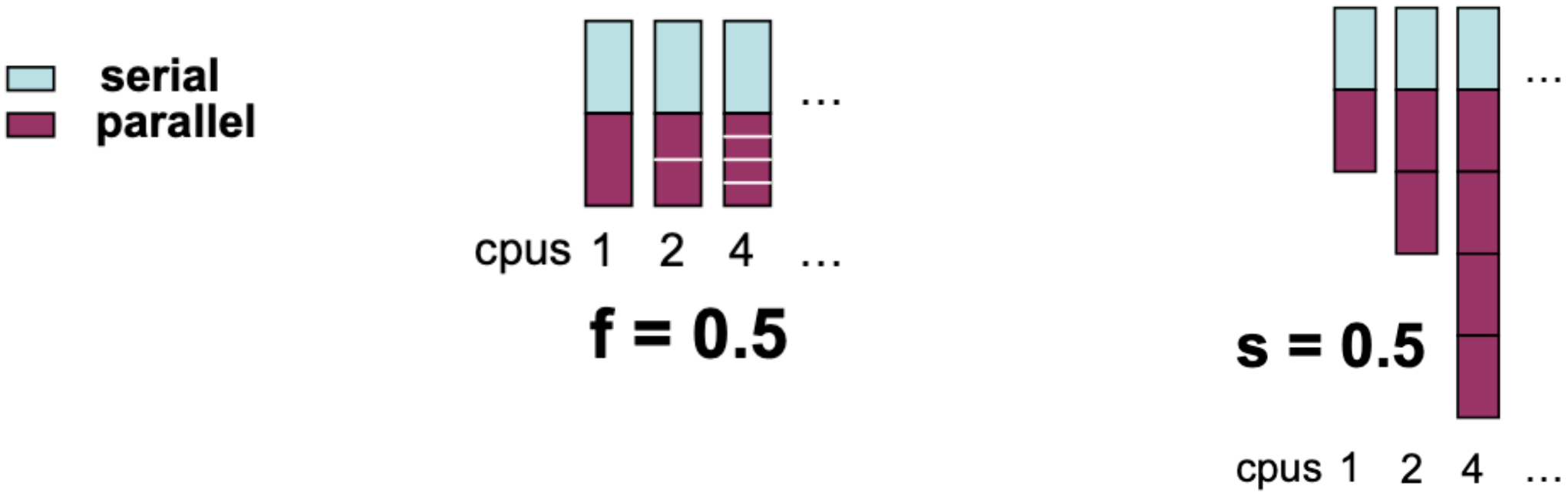
Gustafson observes that σ typically does not vary with problem size n

If $\frac{\varphi(n)}{p}$ is constant, we have s as a constant.

$$\psi(n,p) = \frac{\sigma(n) + \varphi(n)}{\sigma(n) + \frac{\varphi(n)}{p} + \kappa(n,p)} \leq p + (1-p)s$$

$$\varepsilon(n,p) \leq 1 - \left(1 - \frac{1}{p}\right) s$$

Amdahl's and Gustafson's Laws



Amdahl: fixed work size

$$\psi(n,p) = \frac{1}{f + (1-f)/p}$$

$$\psi(n,2) = \frac{1}{0.5 + 0.5/2} = 1.3$$

$$\psi(n,3) = \frac{1}{0.5 + 0.5/3} = 1.6$$

Amdahl's effect

Gustafson: fixed work per processor

$$\psi(n,p) = p + (1-p)s$$

$$\psi(n,2) = 2 + (1-2)0.5 = 1.5$$

$$\psi(n,3) = 4 + (1-4)0.5 = 2.5$$

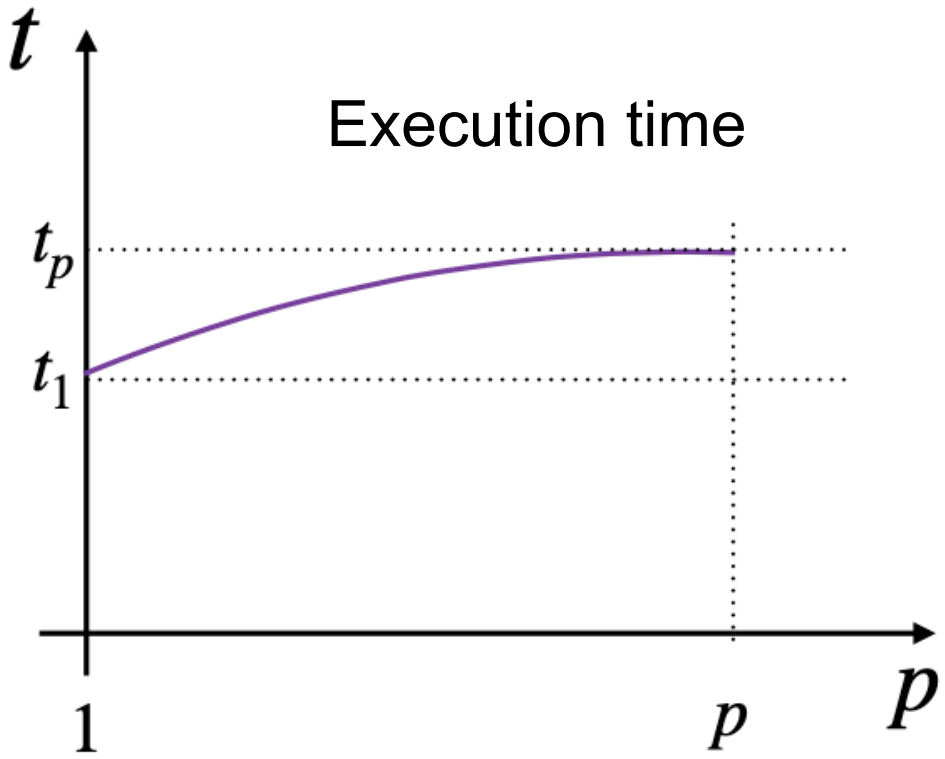
Gustafson's Law: weak scaling analysis

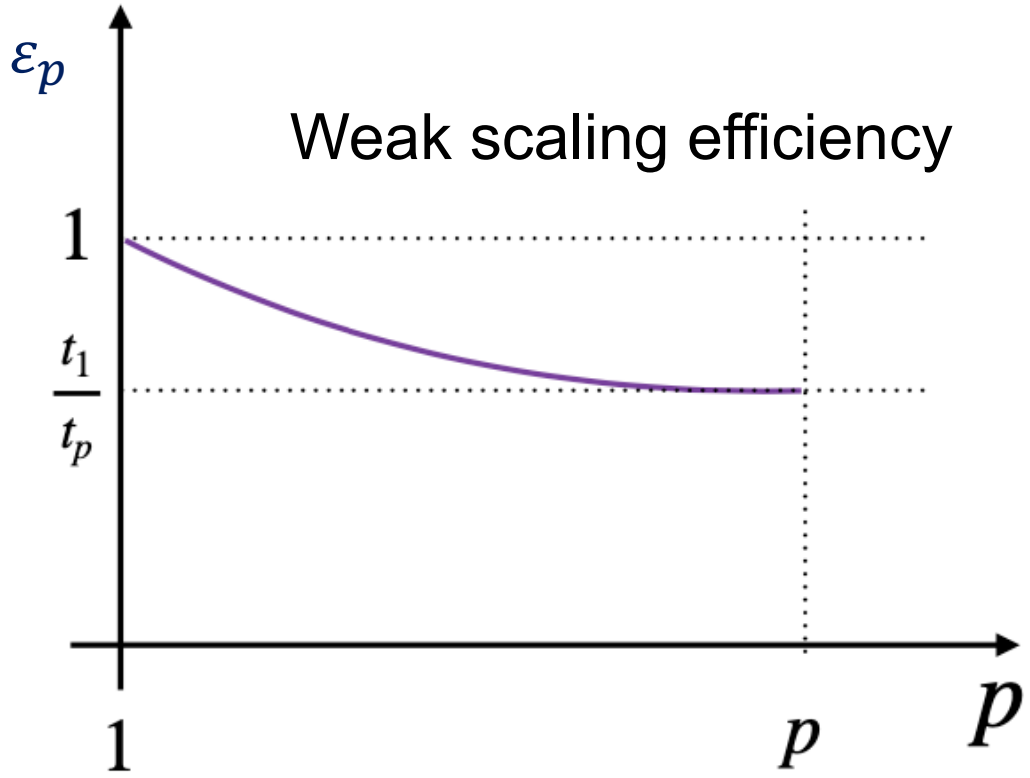
Problem size w per process is fixed

Weak scaling speedup $\psi_p = \frac{{}^{pw}/_{t_p}}{{}^{w}/_{t_1}} = p\,\frac{t_1}{t_p}$

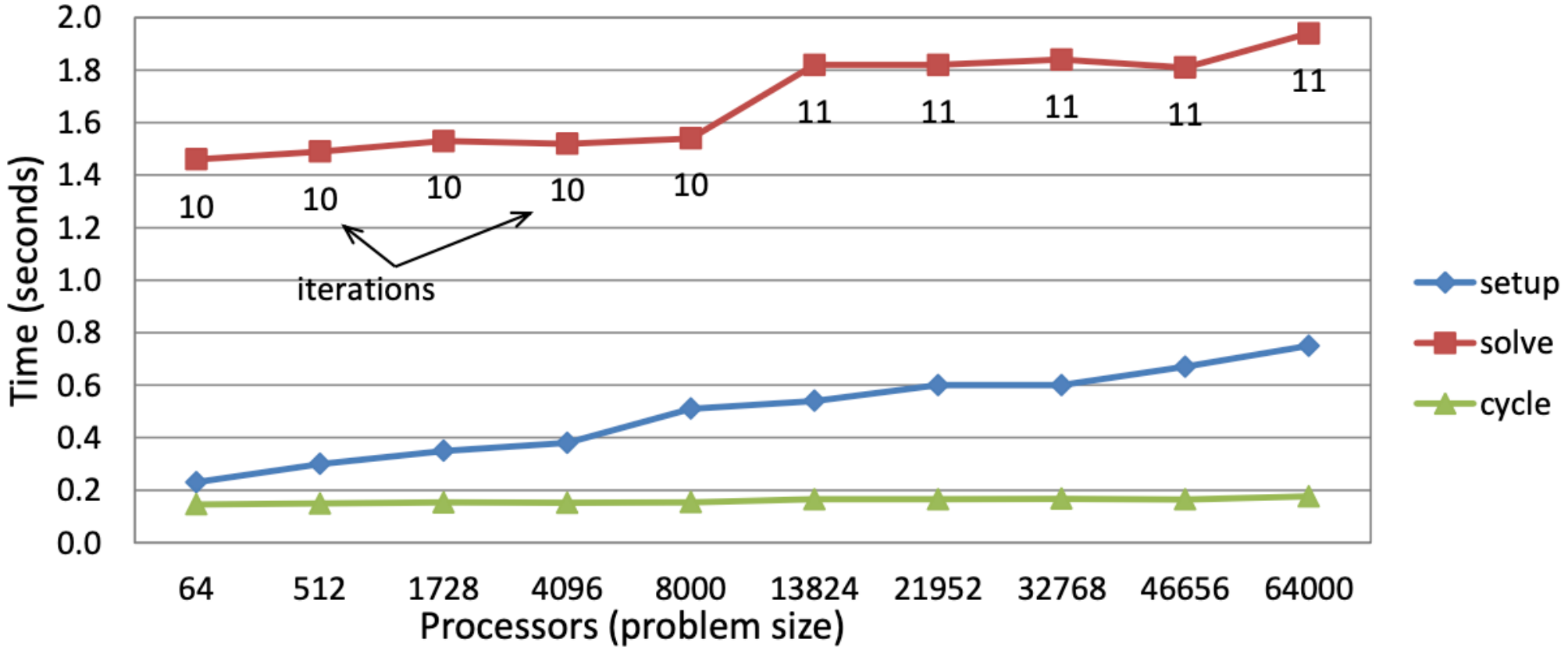
Weak scaling efficiency $\varepsilon_p = \frac{\psi_p}{p} = \frac{t_1}{t_p}$

Presenting weak scaling data:





Gustafson's Law: weak scaling analysis



Laplacian on a cube;

40 x 40 x 40 grid points per processor

Hypre solver

Terminology: Strong and weak scaling

- Scalability: a measure of the system's ability to increase performance as the number of processors increases

- Strong scaling or fixed-size scaling: speedup on multiprocessors **without** increase on problem size
  - Amdahal, G.M. "Validity of single-processor approach to achieving large-scale computing capability", Proceedings of AFIPS conference, Reston, VA. 1967.

- Weak scaling or isogranular scaling: speedup on multiprocessors, while increasing the size of the problem proportionally to the increase in the number of processors
  - Gustafson, J.L. "Reevaluating Amdahl's Law", CACM, 31, 1988.

References

- Parallel programming in C with MPI and OpenMP, Chapters 2 & 7