Applications of Parallel Computing

Logistics

Lectures: Mondays 15:00-17:00 PM

Last Lecture: 15:00-18:00 PM, we might have a Dec 9th lecture

Instructor: Maryam Mehri Dehnavi, mmehride@cs.toronto.edu

Office hours: Mondays 14:00-15:00 PM

See the course syllabus for links to meetings.

Course Overview

Required Prerequisites:
CSC 367 Parallel Computing
CSC336/CSC436 Numerical Methods

Discussions:
Piazza and *Quercus* for all course material along with important announcements.

The course syllabus is in Quercus, please read it carefully**!**

Evaluation

Presentation: We will provide a list of papers with presentation dates. Each group (composed of two students) will present 1-2 paper and possibly one tool (a debugger, profiler, etc.) depending on the final class enrolment count.

Project: An essential component of this course is the course project. You are expected to work on a cutting-edge research problem (in groups of two). The project should have an experimental component and you are expected to deliver a novel contribution that is a scaled-down version of conference paper. The final project report has to be in a 10-page research paper format and should have a reproducibility appendix for artifact evaluation.

Reviews and discussions: For sessions that you are not presenting, you should prepare a short summary of the paper being presented. Also prepare a list of questions that you would ask the presenter during his or her talk. Contributing in the open discussions of research presentations plays an important role in your final grade.

Evaluation

Item	Grade breakdown
Project	50%
Paper/tool presentation	20%
Reviews	15%
Participation	15% (you should also participate in discussions for papers you did not write a review for)

Important Dates

Item	Due date
One-page project proposal	Tuesday, October 12
Four-page mid-term project report	Friday, November 5
Final project presentation	Monday, December 6
Final project report	Wednesday, December 8
Paper/tool presentation and review deadlines	Will be released after the last day of enrolment. Check announcements in Quercus and Piazza.

Parallel Computing Resources

You can use any of the below for your project

- SciNet: Canada's largest supercomputing center, we will use their teach cluster.
- Amazon Web Services: Have to use your free education credits.
- Any compute resource provided by your research group.

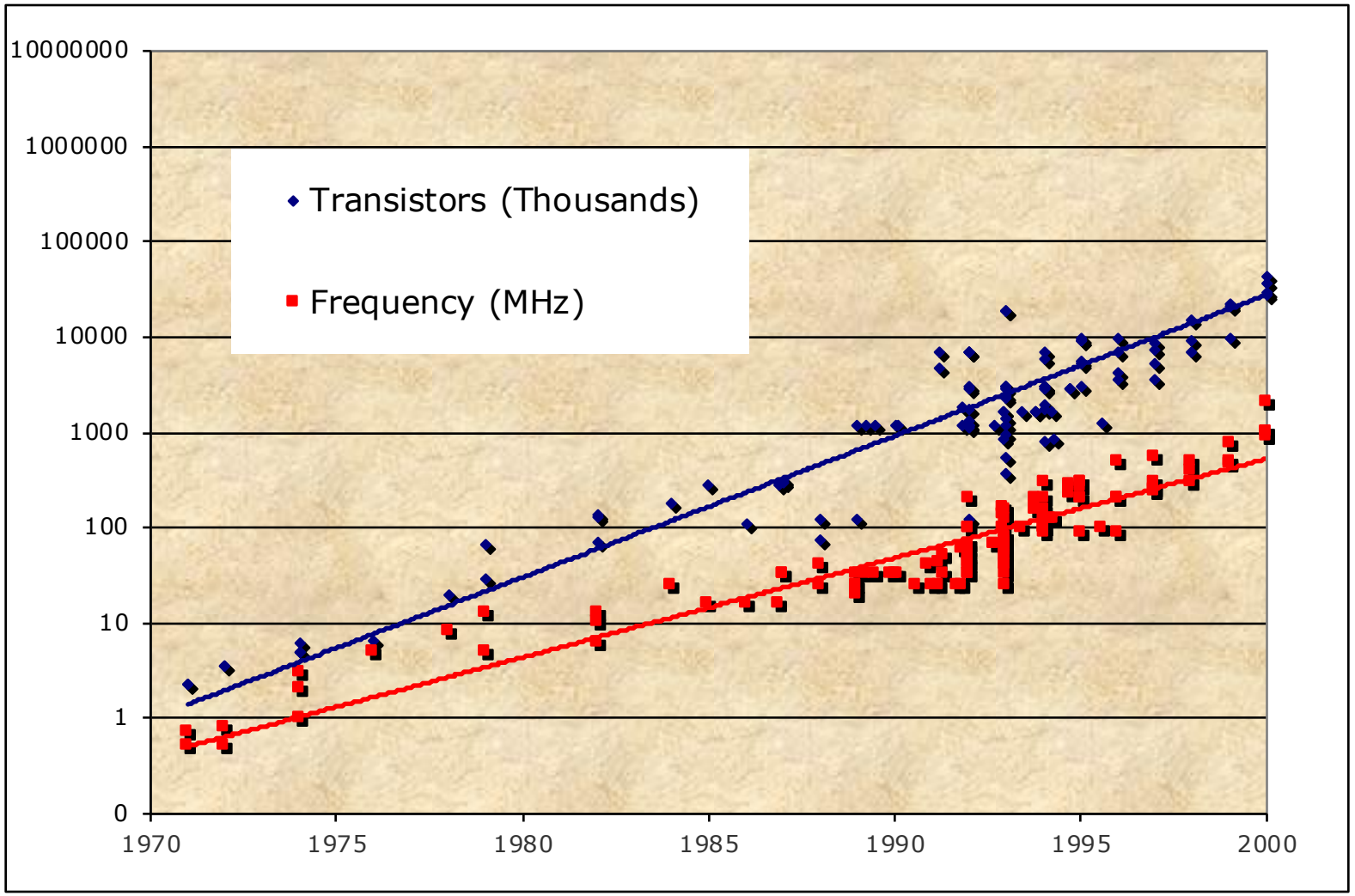
More information to follow in upcoming announcements…

Tentative Topics

- Overview of parallel architectures and programming models
- The computation Dwarfs in parallel algorithms
- Cloud computing
- Performance models
- Domain-specific code generation
- Parallel algorithms
- Parallel computing applications such as machine learning, biology, etc.

Overview of Parallel Computing

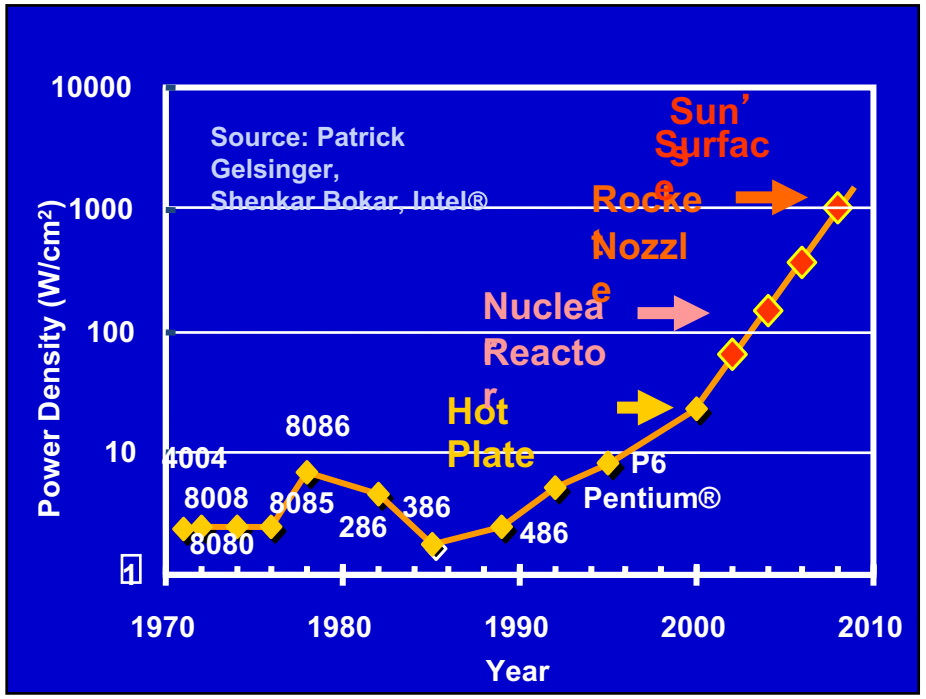
Microprocessor Transistors / Clock (1970-2000)



2X transistors/Chip Every 1.5 years Called "**Moore's Law**"

Power Density Limits Serial Performance

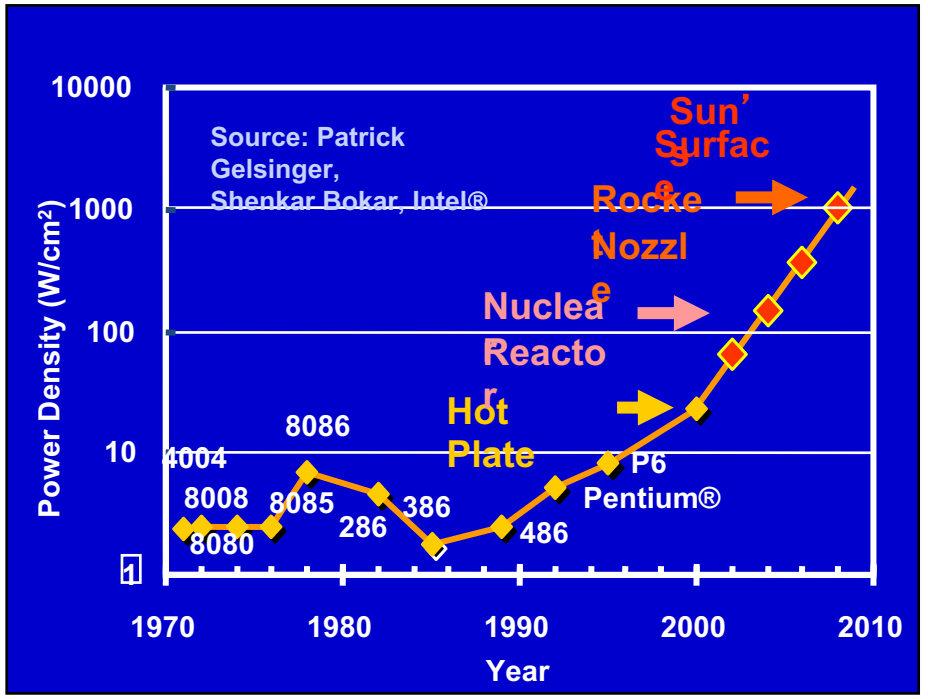
- Concurrent systems are more power efficient
  - Dynamic power is proportional to



Source Demmel

Power Density Limits Serial Performance

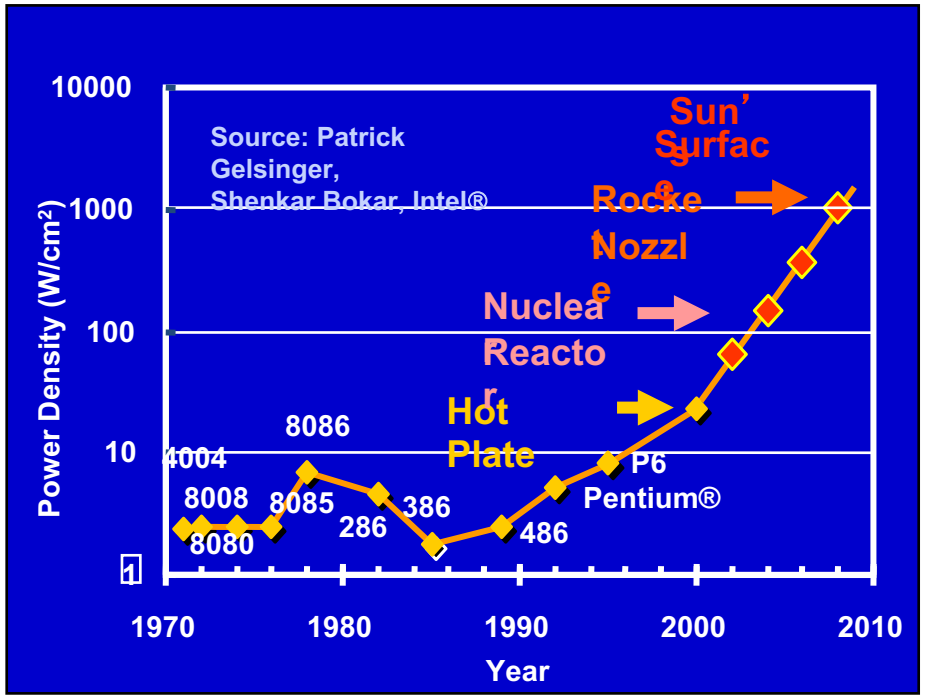
- Concurrent systems are more power efficient
  - Dynamic power is proportional to V^2fC
  - Increasing frequency (f) also increases supply voltage (V) →



Source Demmel

Power Density Limits Serial Performance

- Concurrent systems are more power efficient
  - Dynamic power is proportional to V^2fC
  - Increasing frequency (f) also increases supply voltage (V) → cubic effect
  - Increasing cores increases capacitance (C) but only linearly
  - Save power by lowering clock speed



Source Demmel

- High performance serial processors waste power
  - Speculation, dynamic dependence checking, etc. burn power
  - Implicit parallelism discovery
- More transistors, but not faster serial processors

Revolution in Processors

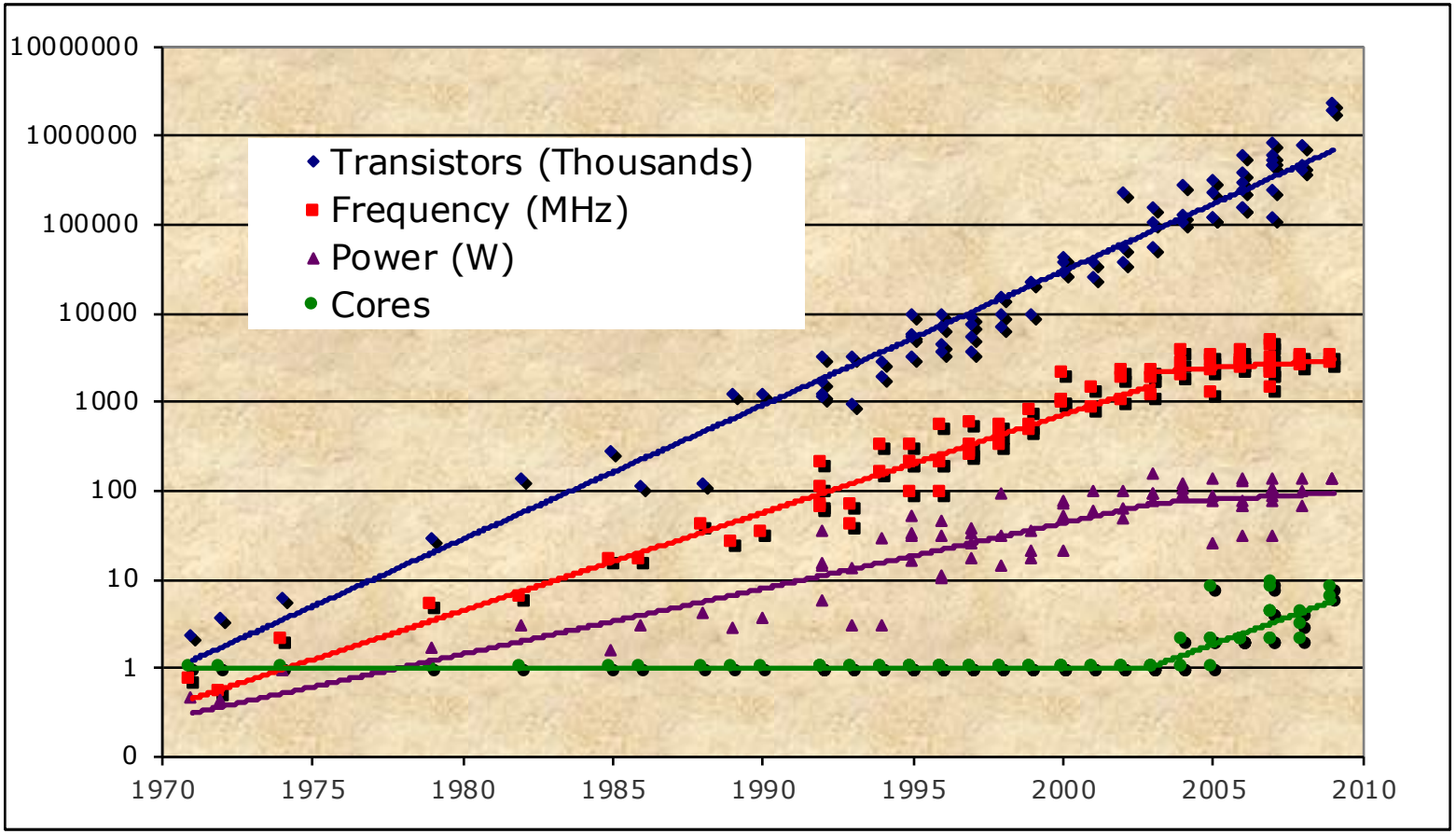
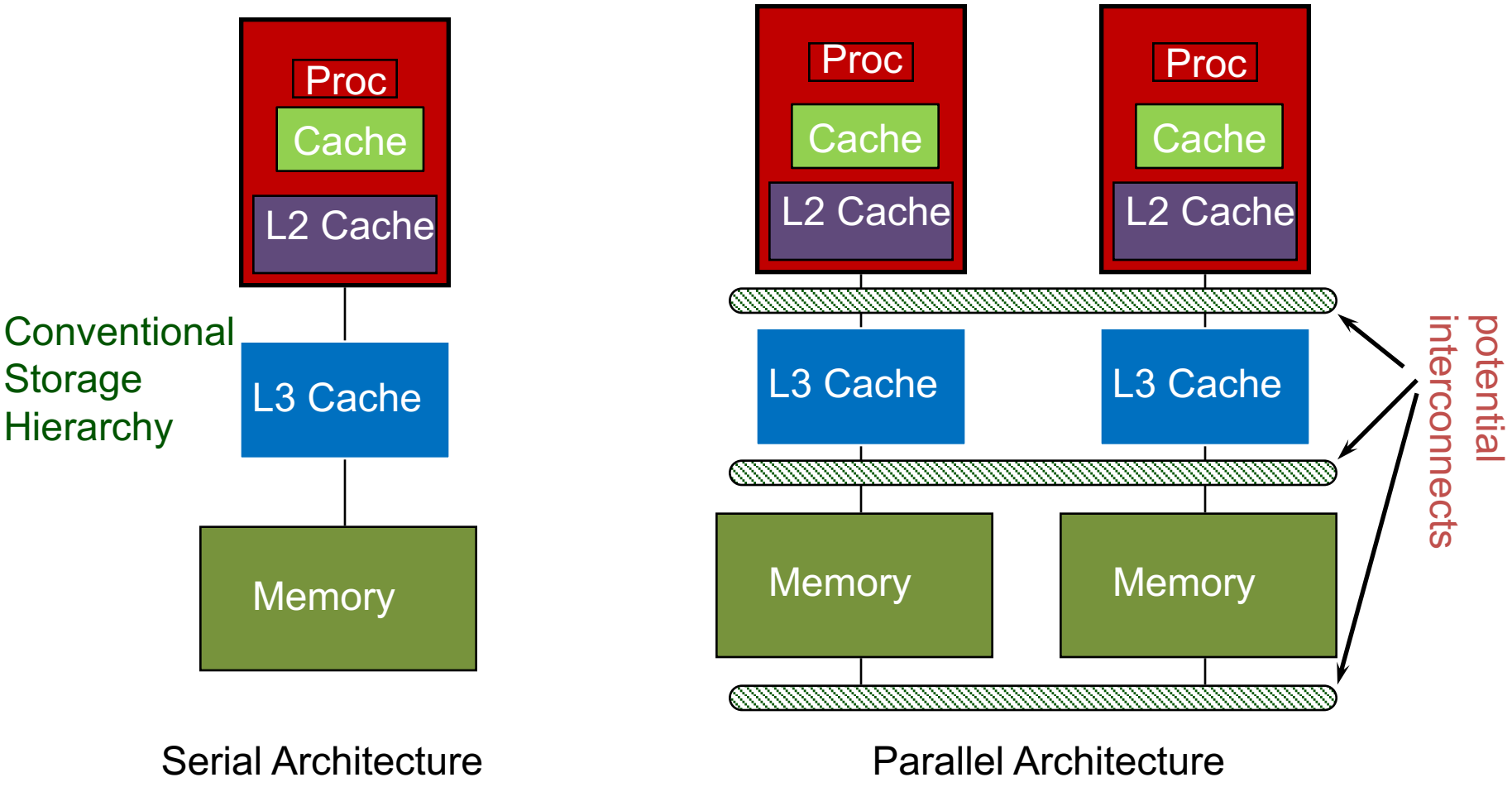


Image source: Leiserson

Revolution in Processors

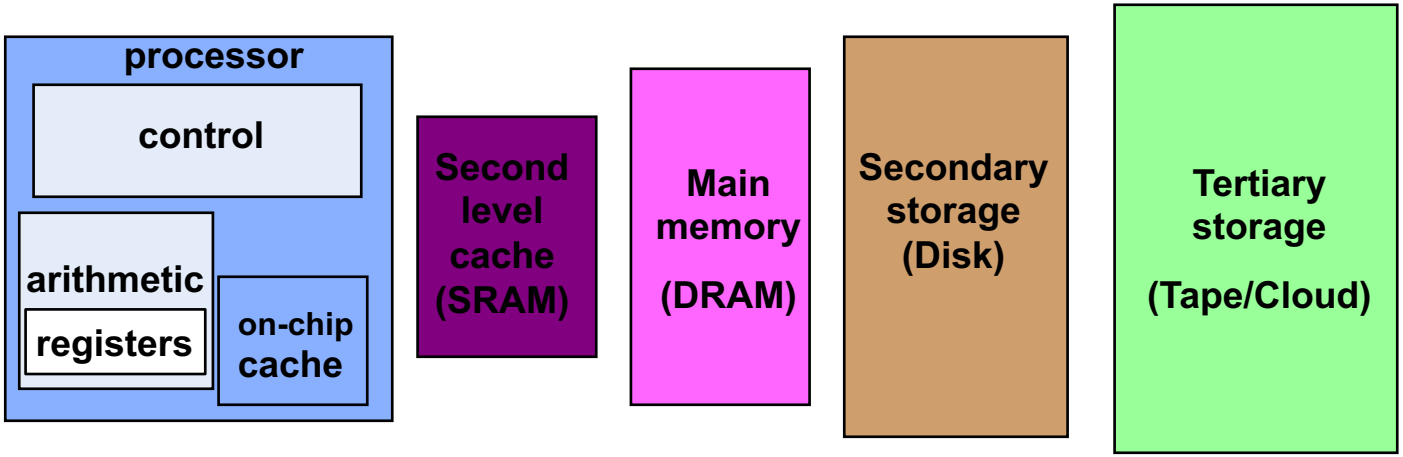
- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

Serial and Parallel Architectures

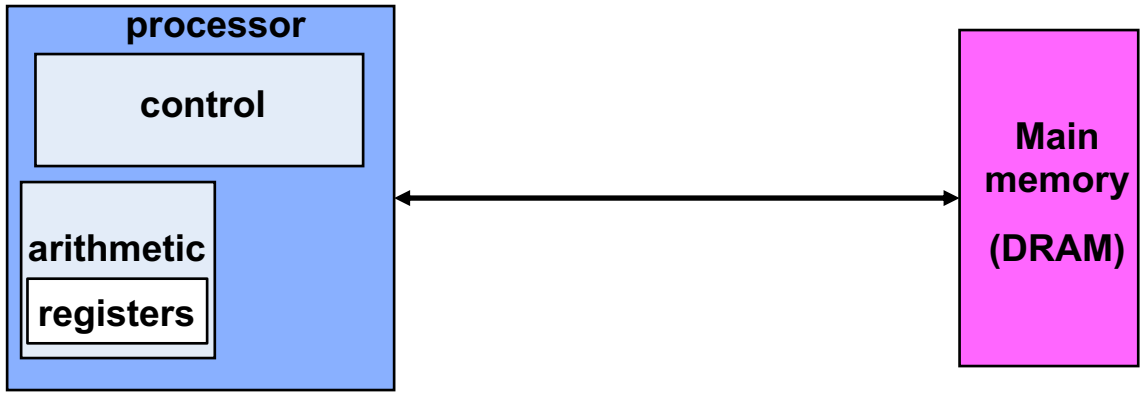


The Memory Hierarchy

The memory hierarchy of a single processor can contain multiple levels of cache, memory, and storage.



Bandwidth and Latency



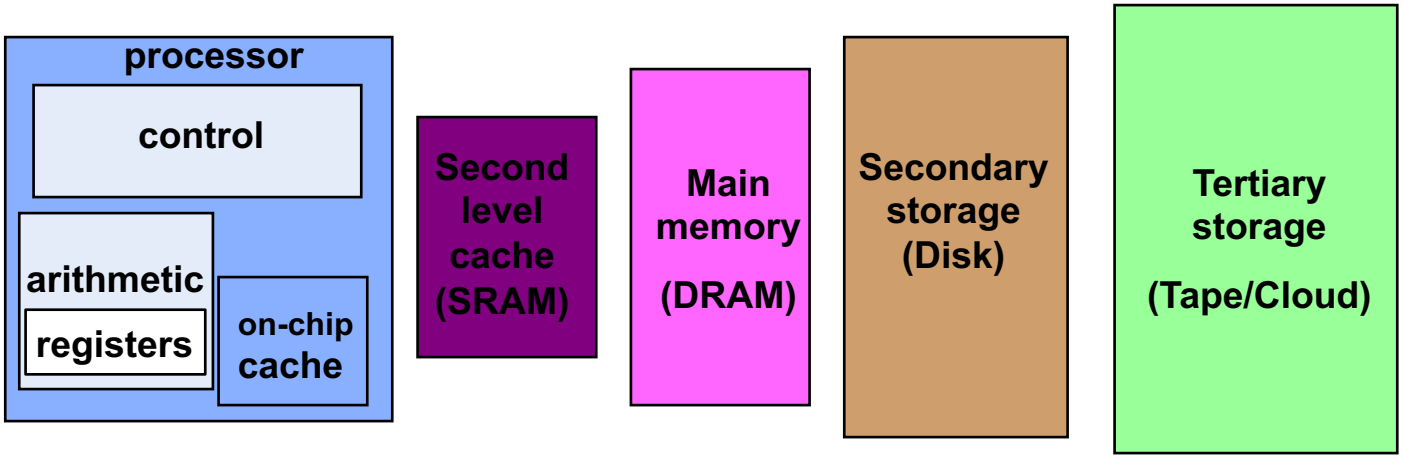
Latency: The time it takes to go from one point to another.

- For example the memory access latency can be 100 ns which means it will take 100ns to retrieve one byte/word from memory.

Bandwidth: The amount of data transferred in a fixed period of time.

- For example the memory bandwidth can be 100MB/s.

Memory Hierarchy and Latency



Latency	1ns	10ns	100ns	10ms	10sec
Size	KB	MB	GB	TB	PB

Caches and Locality (Helps hide latency!)

Reads and writes from main memory are in contiguous blocks, i.e. cache lines.

- *Caches* store previously accessed cache lines to hopefully reduce references to memory!
- If the data required by the processor resides in cache we get a *cache hit*, fast!
- Data accesses that are not in the cache lead to a cache miss, slow!

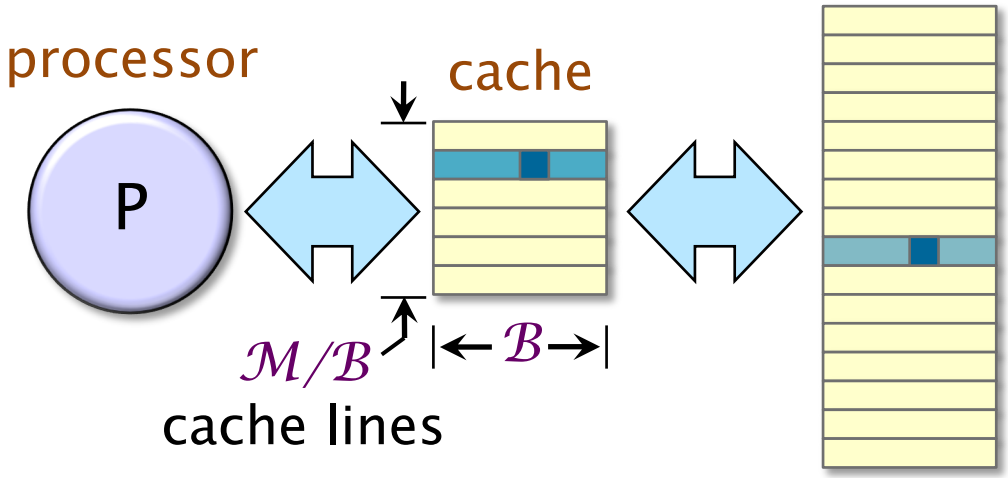


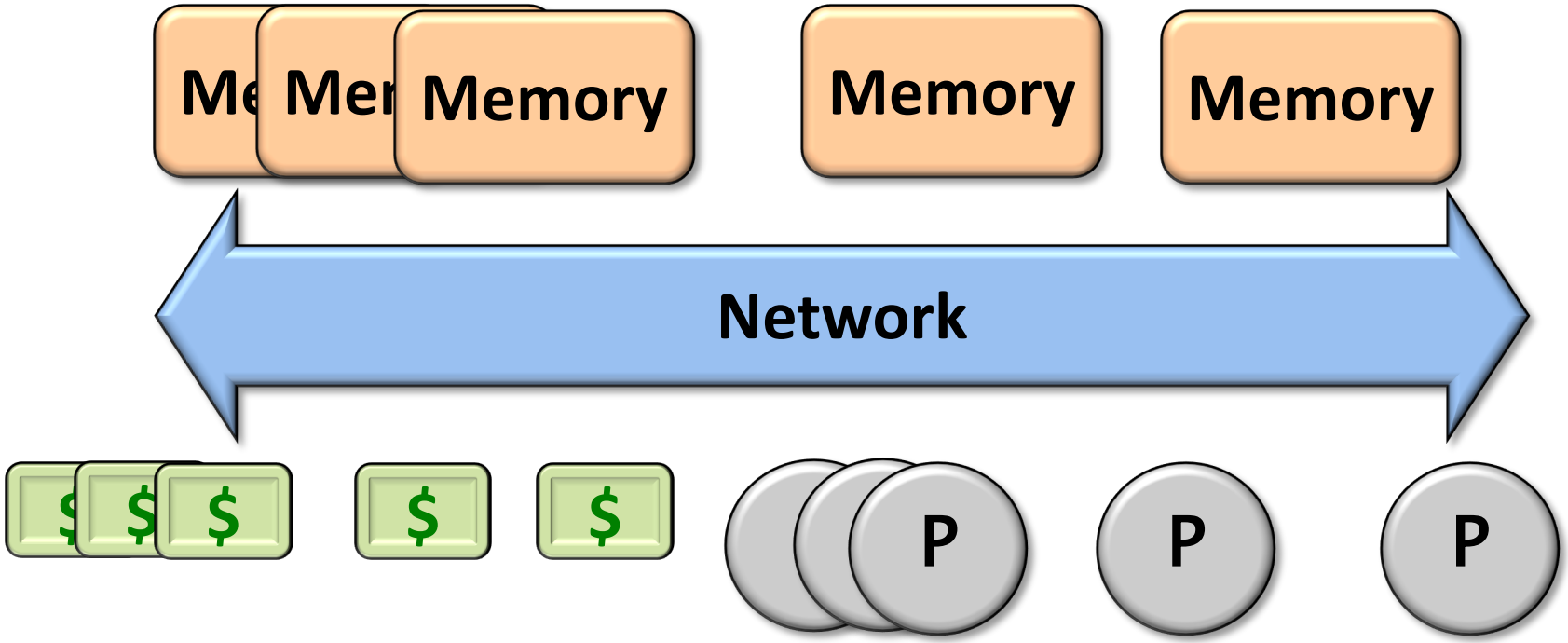
Image source: Leiserson

Approaches to Handling Memory Latency

- Temporal locality: Reuse data that is already in cache.
  - Eliminate memory operations by saving values in small, fast memory (cache or registers) and reusing them (bandwidth filtering).

- Spatial locality: Operate on data that are stored close to each other in memory and can be brought into cache together.
  - Take advantage of better bandwidth by getting a chunk of memory into cache (or registers) and using the whole chunk.

Essential Components of Parallel Architectures



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

Parallel Machine Models

?

Parallel Machine Models

Shared Memory

Distributed Memory

SIMD and Vector

Hybrid

Parallel Machine Models

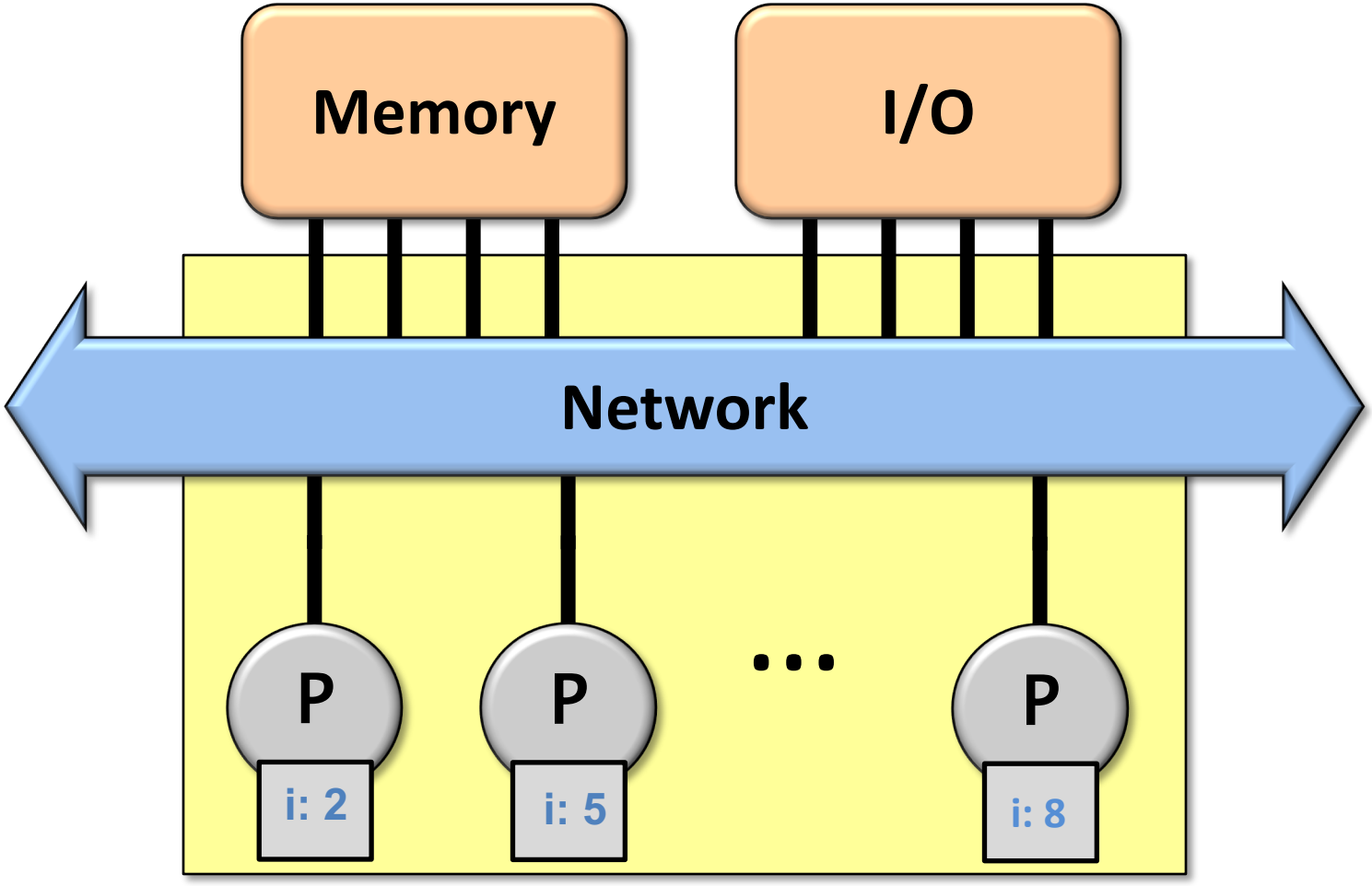
Shared Memory: chip multiprocessors, multicore architectures, etc.

Distributed Memory

SIMD and Vector: GPUs, DSPs, etc.

Hybrid

Shared Memory Architecture



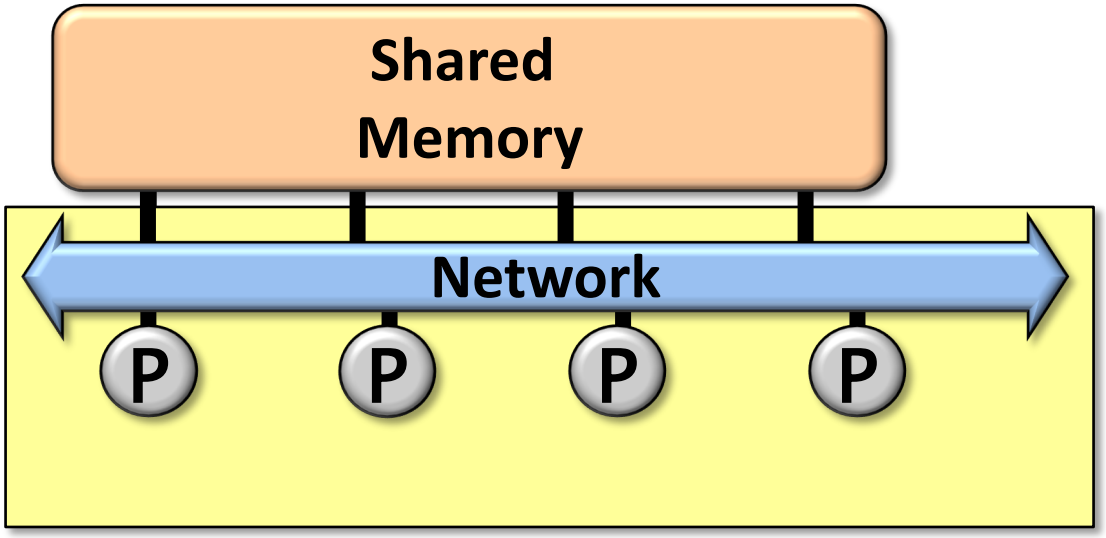
Chip Multiprocessor (CMP)

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?

Programming Model: Shared Memory

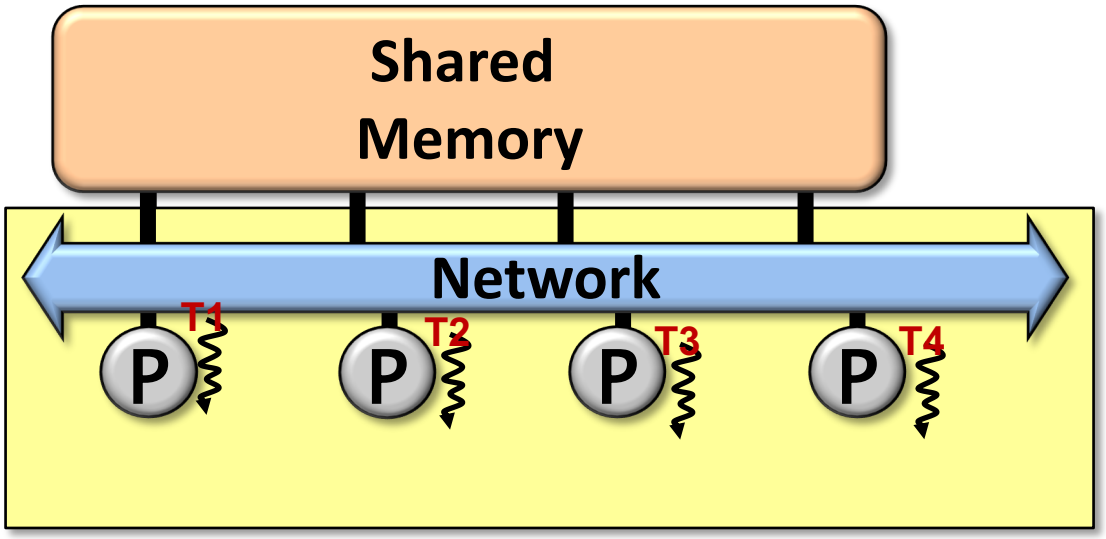
Program is a collection of threads of control, can be created mid-execution.



Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

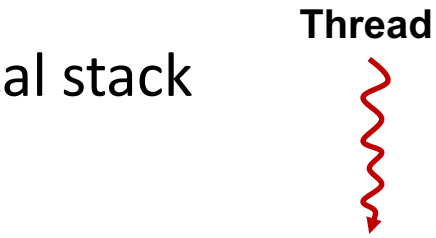
Thread

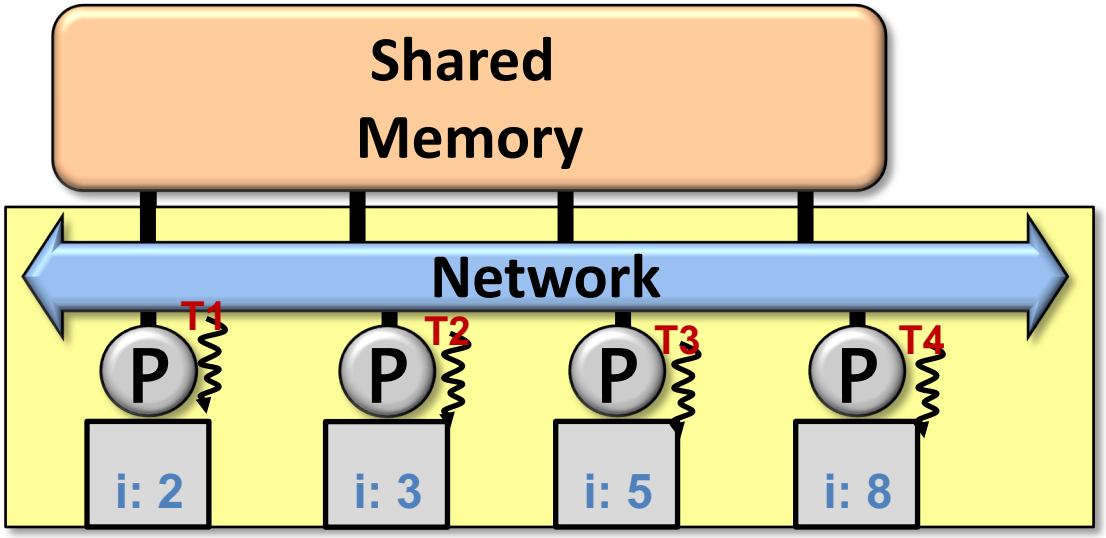


Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.





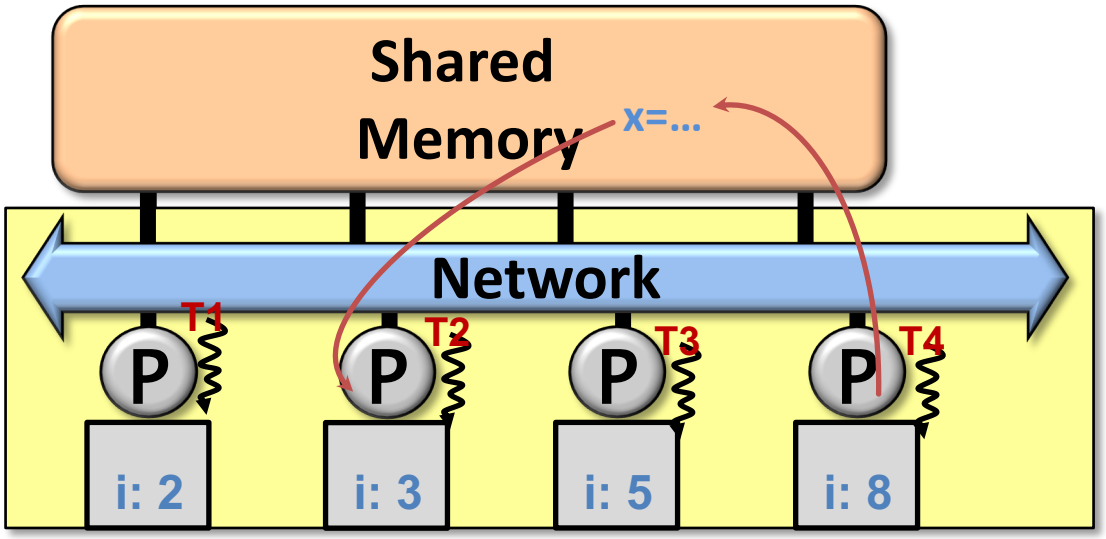
Programming Model: Shared Memory

Program is a collection of threads of control, can be created mid-execution.

Each thread has a set of private variables, e.g., local stack variables.

Also a set of shared variables, e.g., static variables.

- Threads communicate implicitly by writing and reading shared variables.



Shared memory: Challenges

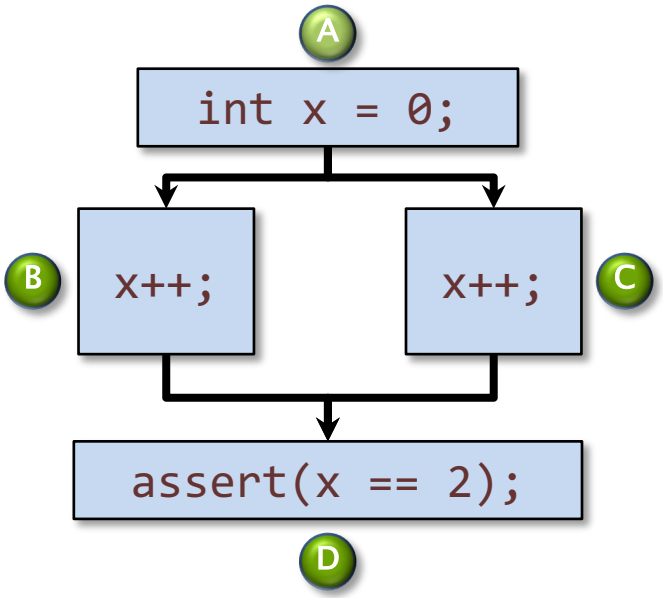
- **Races and Locks**

Races

A race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

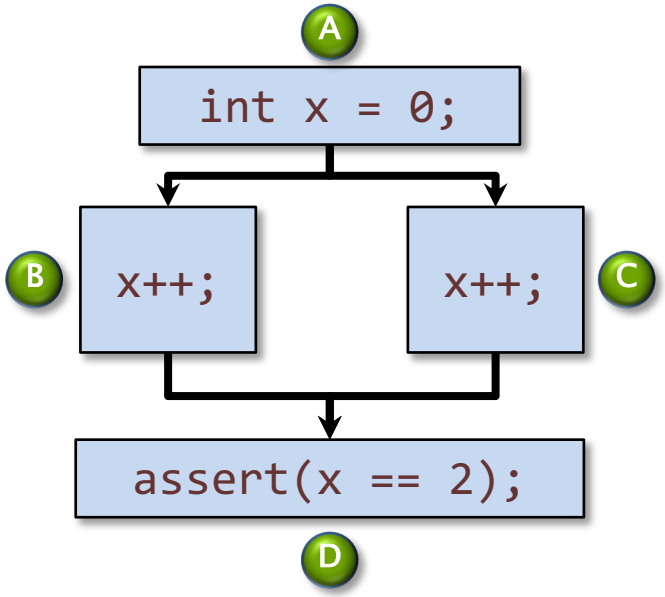
Example

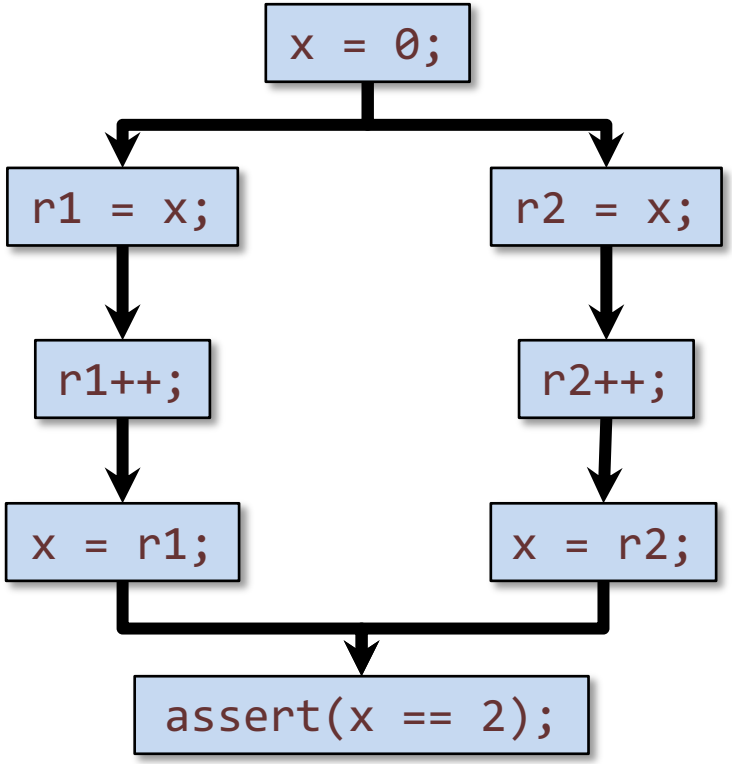
```
A  int x = 0;
   parallel_for (int i=0, i<2, ++i) {
B C    x++;
   }
D  assert(x == 2);
```



dependency graph

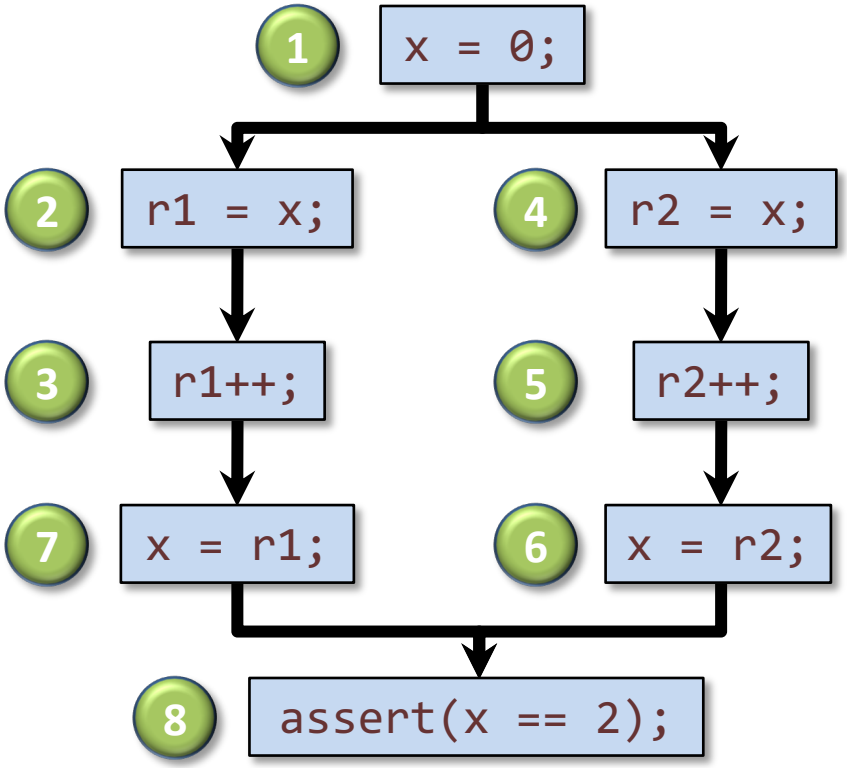
A Closer Look

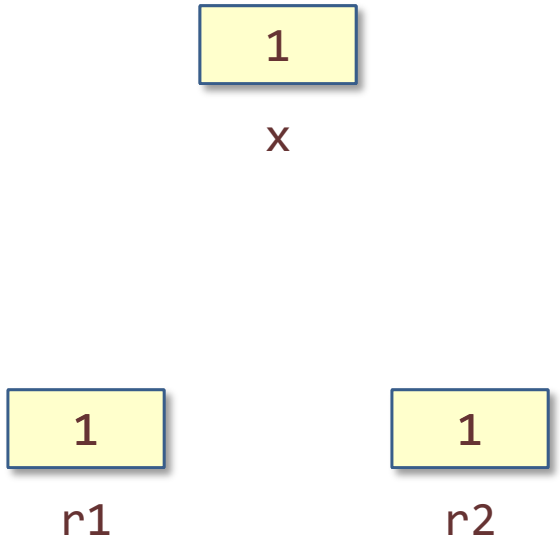




Race Bugs

A race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.





Source: Leiserson

Simple Example

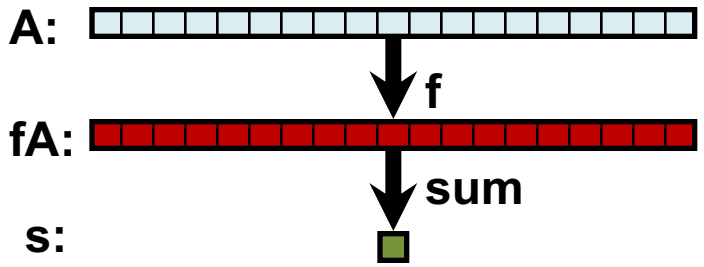
- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

FIR filter: $y[n] = \sum_{i=0}^{n-1} b_i x[n-i]$

- Questions:
  - Where does A live? All in single memory? Partitioned?

A = array of all data
fA = f(A)
s = sum(fA)



Simple Example

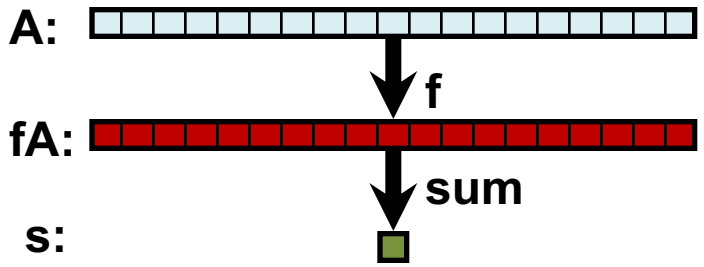
- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

FIR filter: $y[n] = \sum_{i=0}^{n-1} b_i x[n-i]$

- Questions:
  - Where does A live? All in single memory? Partitioned?

A = array of all data
fA = f(A)
s = sum(fA)



Shared Memory Code for Computing a Sum

A= | 3 | 5 |

$f(x) = x^2$

```
static int s = 0;
```

Thread 1

```
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 2

```
for i = n/2, n-1
    s = s + f(A[i])
```

- What is the problem with this program?

Shared Memory Code for Computing a Sum

A= | 3 | 5 |

$f(x) = x^2$

```
static int s = 0;
```

Thread 1		Thread 2	
….		…	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg(	25
reg1 = s	0	reg1 = s	0
reg1 = reg1 + reg0	9	reg1 = reg1 + reg0	25
s = reg1	9	s = reg1	25
…		…	

- Assume A = [3,5], $f(x) = x^2$, and s=0 initially
- For this program to work, s should be $3^2 + 5^2 = 34$ at the end
  - but it may be 34,9, or 25

Improved Code for Computing a Sum

```
static int s = 0;
static lock lk;
```

Why not do lock Inside loop?

Thread 1

```
local_s1= 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
lock(lk);
    s = s + local_s1
unlock(lk);
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2= local_s2 + f(A[i])
lock(lk);
    s = s +local_s2
unlock(lk);
```

- Since addition is associative, it's OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

Shared Memory Programming

Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - openmp.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C "ilk"
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

Example with OpenMP

Matrix Vector Multiply OpenMP

```
#pragma omp parallel default (none) shared (A, W, Y, m) private (i,j,sum) num_threads(4)
for(i=0; i < m; i++)
  { sum = 0.0;
    for(j=0; j < m; j++)
      sum += A[i][j]*W[j];
    Y[i] =sum;
  }
```

Parallel Machine Models

Shared Memory: chip multiprocessors, multicore architectures, etc.

Distributed Memory

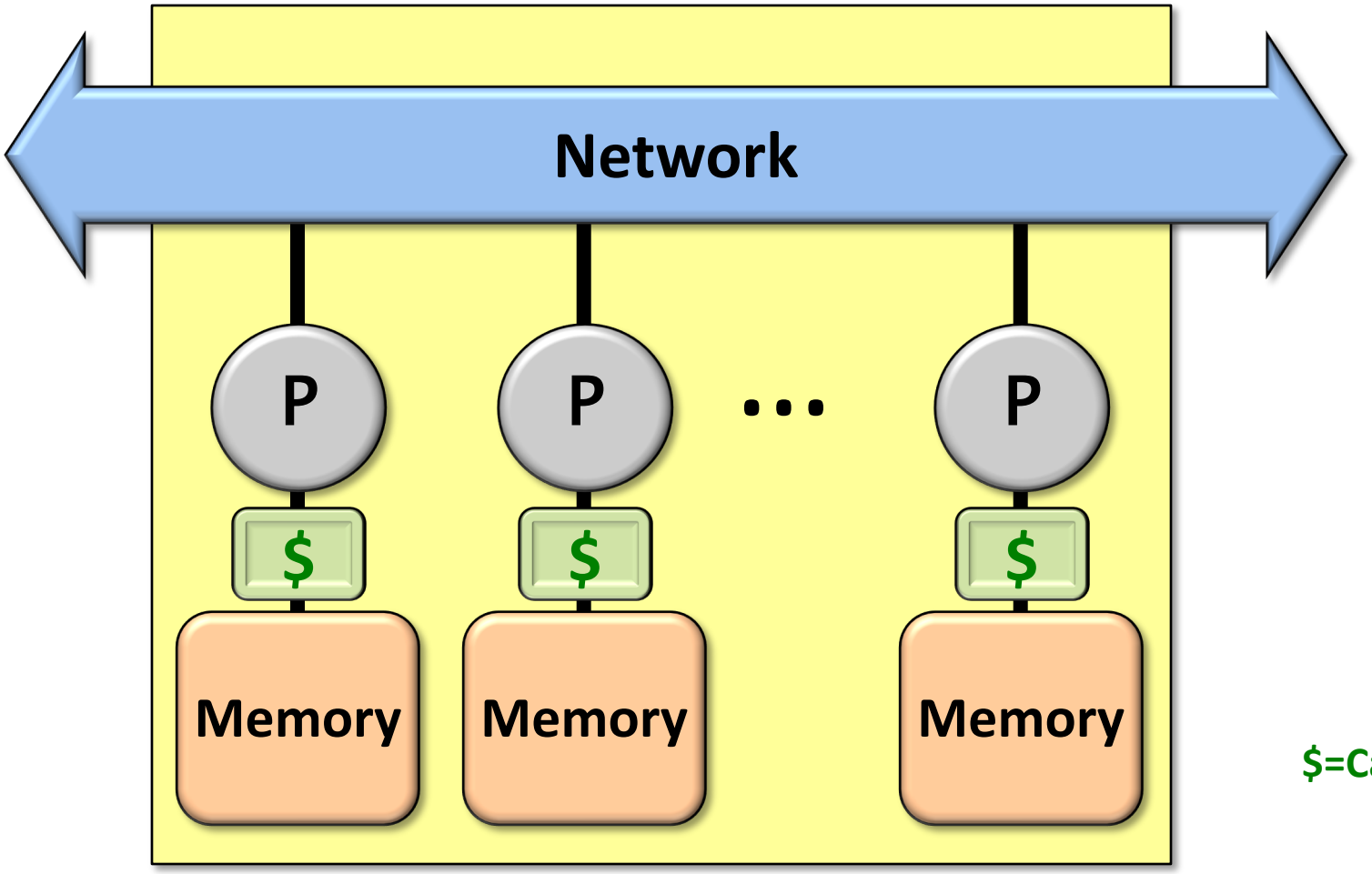
SIMD and Vector: GPUs, DSPs, etc.

Hybrid

Distributed Memory Machines and Programming

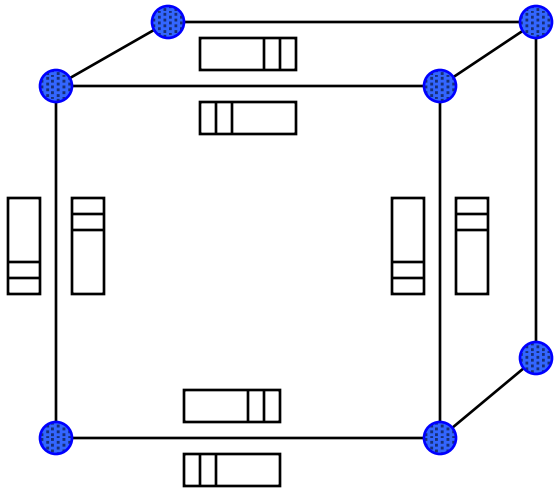
Some Slides from Jonathan Carter, Katherine Yelick, and Bill Gropp

Distributed Memory Architecture



Historical Perspective

- Early distributed memory machines were:
  - Collection of microprocessors.
  - Communication was performed using bi-directional queues between nearest neighbors.
- Messages were forwarded by processors on path.
  - "Store and forward" networking
- There was a strong emphasis on topology in algorithms, in order to minimize the number of hops = minimize time

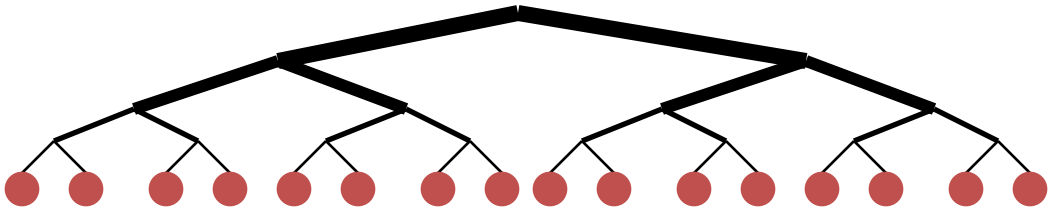


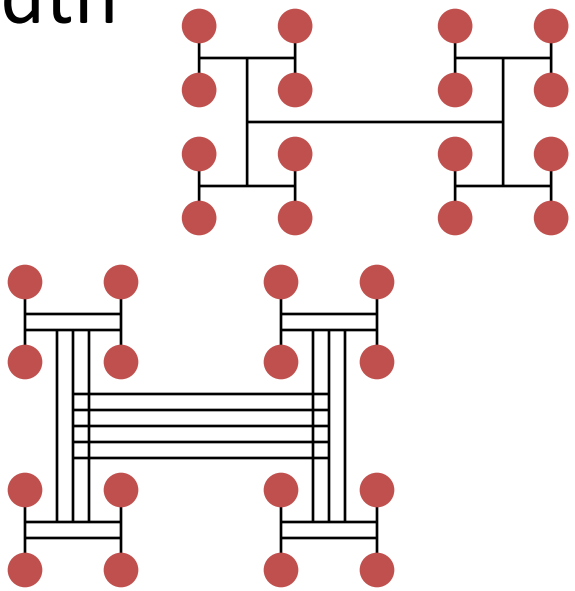
Design Characteristics of a Network

- Topology (how things are connected)
  - Crossbar; ring; 2-D, 3-D, higher-D mesh or torus; hypercube; tree; butterfly; perfect shuffle, dragon fly, …
- Routing algorithm:
  - Example in 2D torus: all east-west then all north-south (avoids deadlock).
- 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, or internet
- Flow control (what if there is congestion):
  - Stall, store data temporarily in buffers, re-route data to other nodes, tell source node to temporarily halt, discard, etc.

Example Network Topology: Trees

- Many tree algorithms
- Fat trees avoid bisection bandwidth problem:
  - More (or wider) links near top.





Programming Distributed Memory Machines with Message Passing

Message Passing Libraries

- Many "message passing libraries" were once available
  - Chameleon, from ANL.
  - CMMD, from Thinking Machines.
  - Express, commercial.
  - MPL, native library on IBM SP-2.
  - NX, native library on Intel Paragon.
  - Zipcode, from LLL.
  - PVM, Parallel Virtual Machine, public, from ORNL/UTK.
  - Others...
  - MPI, Message Passing Interface, now the industry standard.
- Need standards to write portable code.

Finding Out About the Environment

- Two important questions that arise early in a parallel program are:
  - How many processes are participating in this computation?
  - Which one am I?
- MPI provides functions to answer these questions:
  - **MPI_Comm_size** reports the number of processes.
  - **MPI_Comm_rank** reports the *rank*, a number between 0 and size-1, identifying the calling process

Hello (C)

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

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

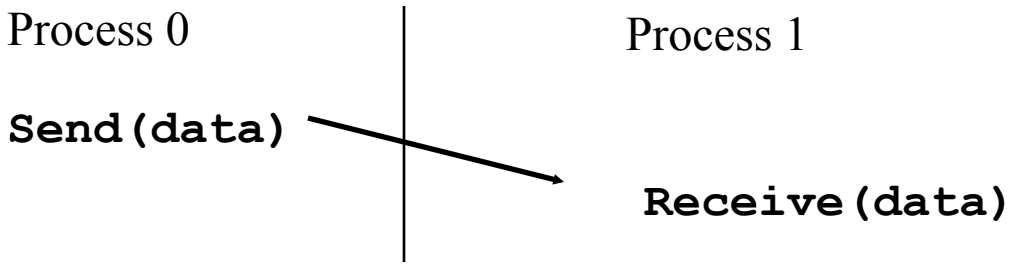
Notes on Hello World

- All MPI programs begin with MPI_Init and end with MPI_Finalize
- MPI_COMM_WORLD is defined by mpi.h (in C) or mpif.h (in Fortran) and designates all processes in the MPI "job"
- Each statement executes independently in each process
  - including the printf/print statements
- The MPI-1 Standard does not specify how to run an MPI program, but many implementations provide

  mpirun –np 4 a.out

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

MPI Datatypes

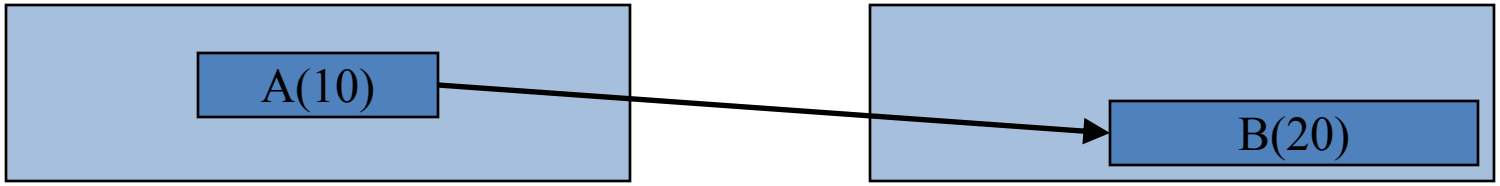
- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI datatype is defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes

MPI Tags

- Messages are sent with an accompanying user-defined integer tag, to assist the receiving process in identifying the message

- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying MPI_ANY_TAG as the tag in a receive

MPI Basic (Blocking) Send



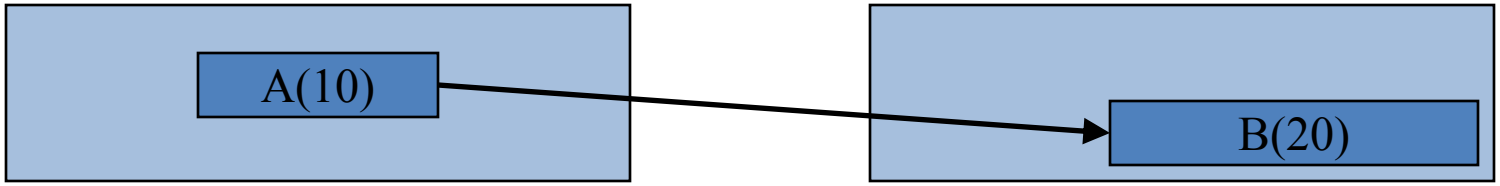
MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

MPI_SEND(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**start, count, datatype**).
- The target process is specified by **dest**, which is the rank of the target process in the communicator specified by **comm**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive



MPI_Send(A, 10, MPI_DOUBLE, 1, …)

MPI_Recv(B, 20, MPI_DOUBLE, 0, …)

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both **source** and **tag**) message is received from the system, and the buffer can be used
- **source** is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**
- **tag** is a tag to be matched or **MPI_ANY_TAG**
- receiving fewer than **count** occurrences of **datatype** is OK, but receiving more is an error
- **status** contains further information (e.g. size of message)

A Simple MPI Program

```
#include “mpi.h”
#include <stdio.h>
int main( int argc, char *argv[])
{
  int rank, buf;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank );

  /* Process 0 sends and Process 1 receives */
  if (rank == 0) {
    buf = 123456;
    MPI_Send( &buf, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (rank == 1) {
    MPI_Recv( &buf, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
              &status );
    printf( “Received %d\n”, buf );
  }

  MPI_Finalize();
  return 0;
}
```

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_SEND**
  - **MPI_RECV**

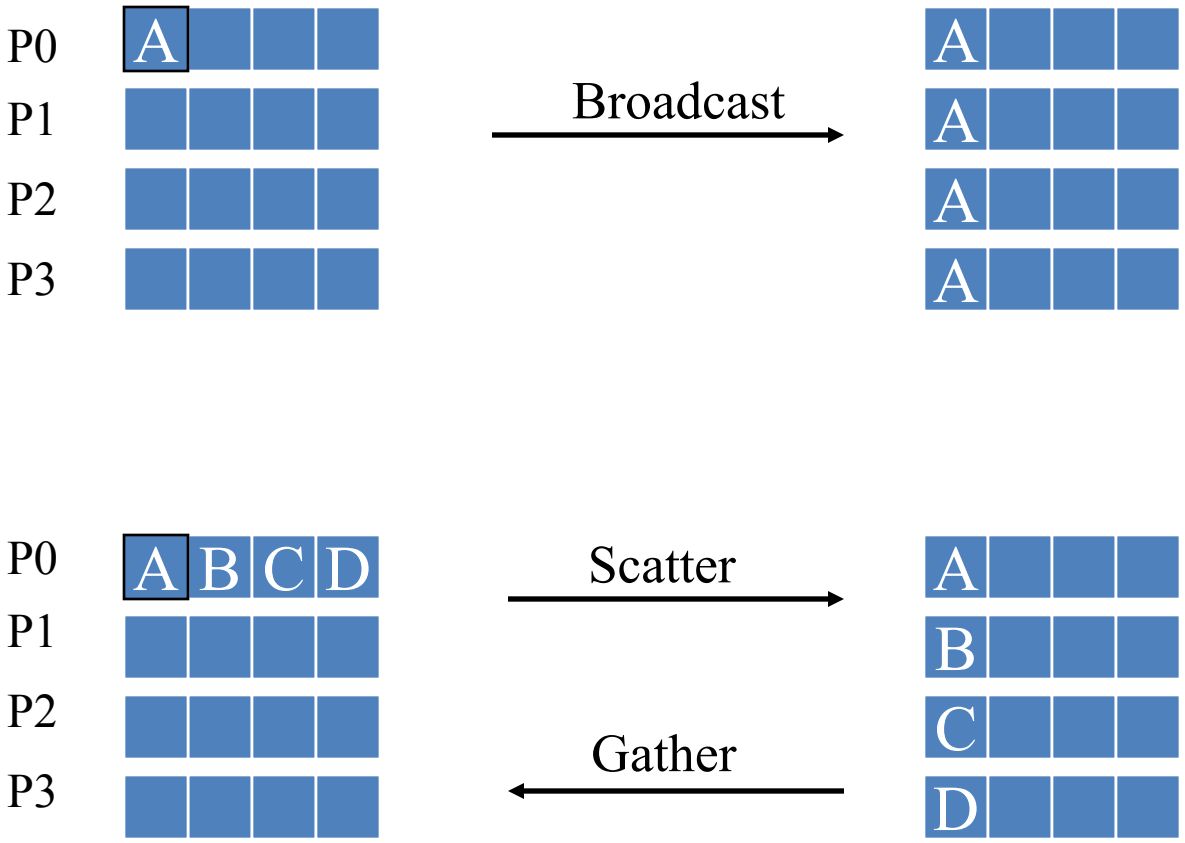
Another Approach to Parallelism

- *Collective* routines provide a higher-level way to organize a parallel program
- Each process executes the same communication operations
- MPI provides a rich set of collective operations…

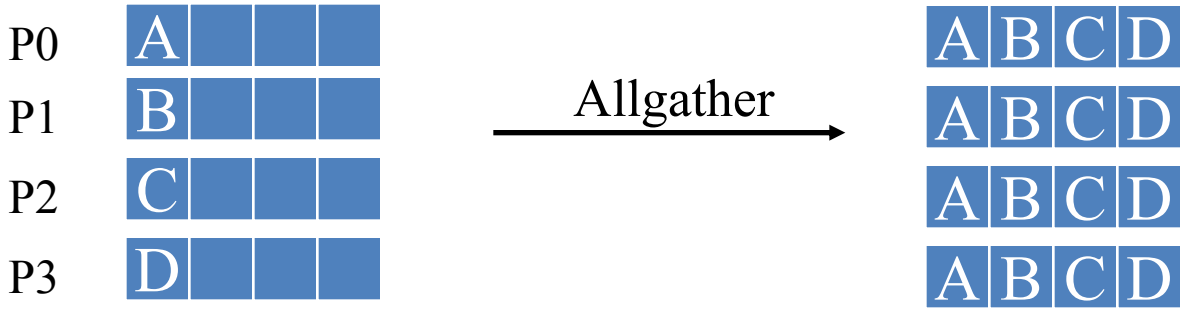
Collective Operations in MPI

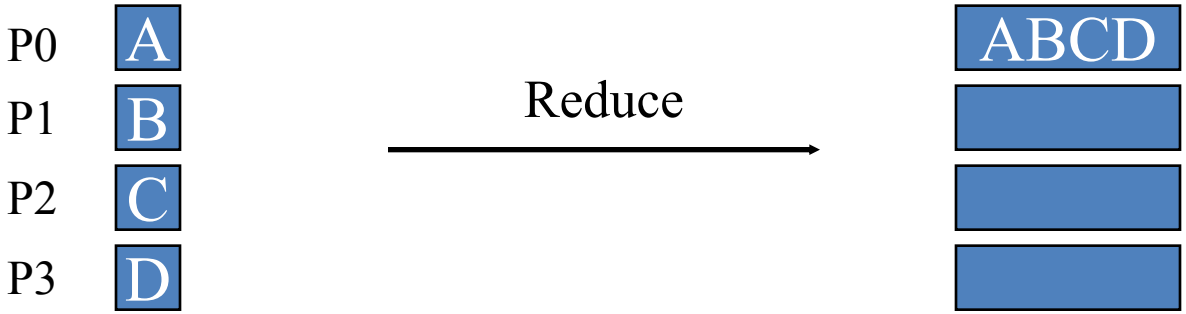
- Collective operations are called by all processes in a communicator
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process

Collective Data Movement



More Collective Data Movement





MPI References

- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Programming Models

Message Passing Models (MPI)

Fine-grained messages + computation

Hard to write code to deliver high coordination
Hard to maintain loadbalance
Hard to deal with disk locality, failures, stragglers

“Cloud” Programming Models

- Data Parallel Models
- Restrict the programming interface
- Automatically handle failures, locality etc.

“Here’s an operation, run it on all of the data”

- I don’t care *where* it runs (you schedule that)
- In fact, feel free to run it *retry* on different nodes

- Programming models: Map Reduce
- Frameworks and Engines: Hadoop , Spark, and many more.

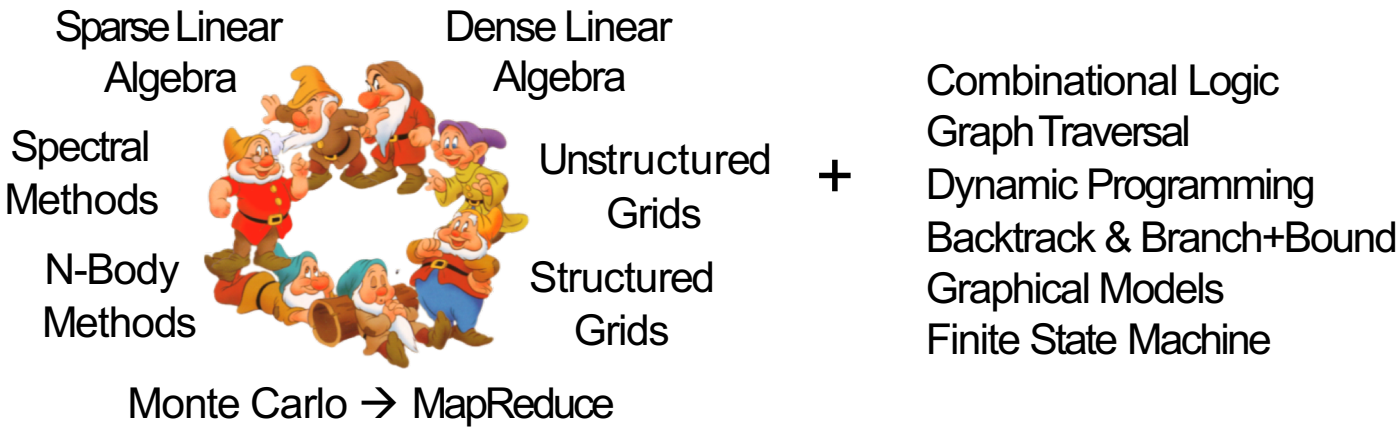
Seven/thirteen DwarFs/Patterns of Computing!

Thanks to W. feng and Alvin Lebeck

What are the 13 Dwarfs?

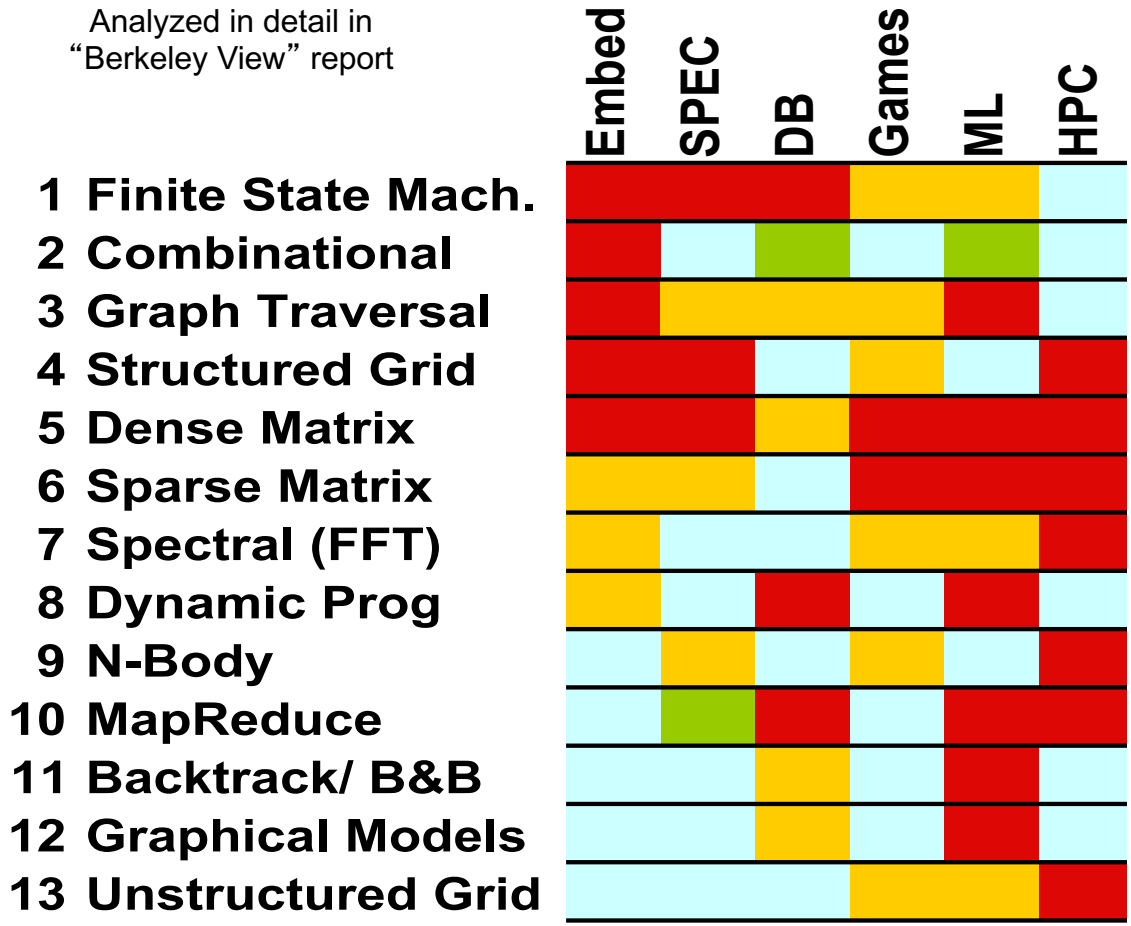
- A (computational) dwarf is an algorithmic method that *captures a pattern of computation and communication*
  - Inspired by Phil Colella, who identified *seven* numerical methods important for science and engineering

- Benchmark Suite of 13 Computational Dwarfs & Apps



What do commercial and CSE applications have in common?

Analyzed in detail in "Berkeley View" report

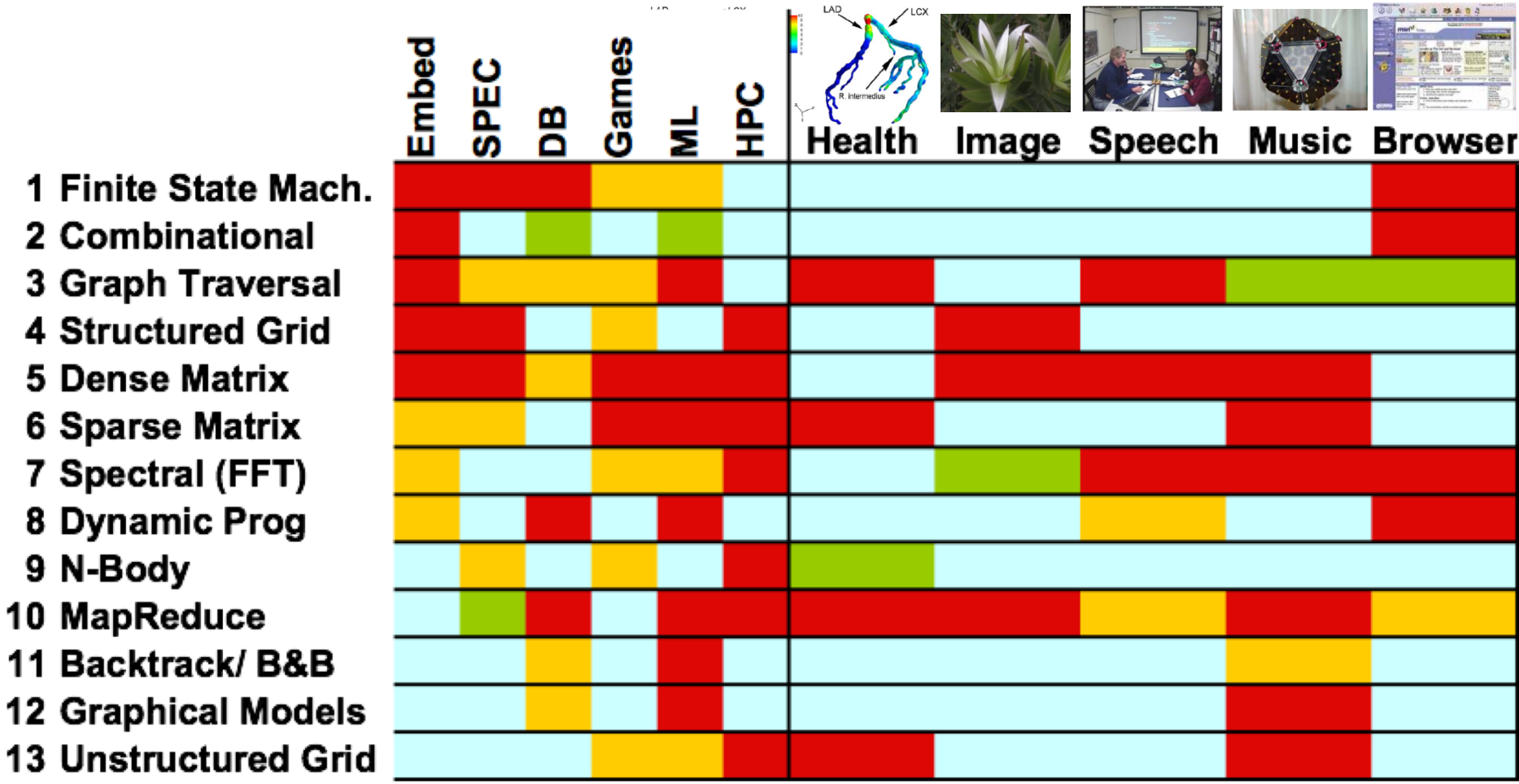


Analyzed in detail in "Berkeley View" report www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

What do commercial and CSE applications have in common?

Motif/Dwarf: Common Computational Methods

(Red Hot → Blue Cool)



N–Body Methods

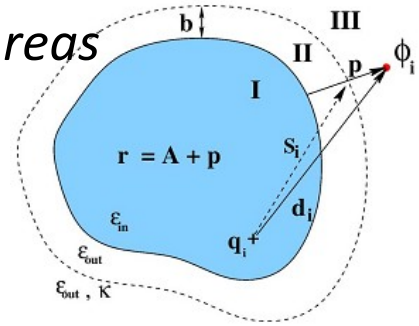
Calculations that depend on interactions between discrete points

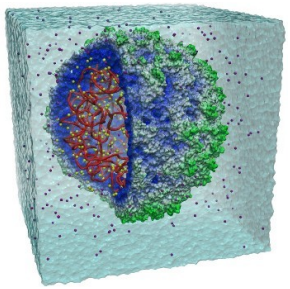
Approaches

- In particle--particle methods
  - Every point depends on all others
    - O(N2): Brute--force method
- Hierarchical particle methods
  - Combine forces or potentials from multiple points
    - O(N log N): Barnes--Hutt
    - O(N log N): Hierarchical charge partitioning
    - O(N): Fast multipole

Application Areas

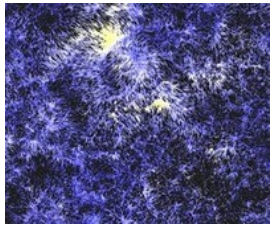
Molecular Modeling





Molecular Dynamics

Cosmology
(Roadrunner Universe @ LANL)



Sparse Linear Algebra

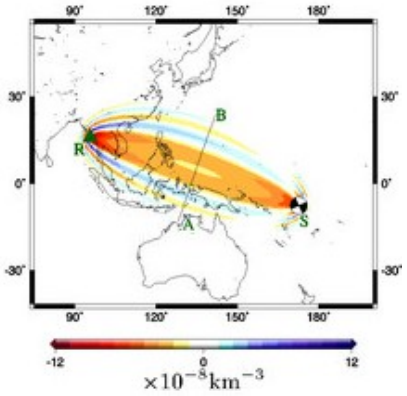
Multiplication involving matrices composed primarily of zeros

$$\begin{pmatrix} 0 & 3 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 5 & 0 & 0 & 0 & 8 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 & 0 & 4 & 0 \\ 0 & 1 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 7 & 0 & 0 & 0 & 0 & 5 & 0 & 0 \end{pmatrix}$$

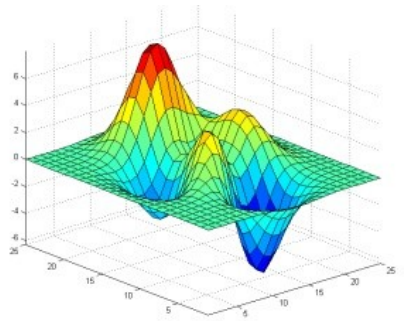
Approaches

- Direct solving
  - Solve the problem with a series of operations
    - Large overhead
- Iterative methods
  - Use a series of successive approximations
    - Begins with a guess
    - Repeats until error is not significant

Application Areas & Kernels



Finite element analysis



Partial differential equations

Dense Linear Algebra

Classic vector and matrix operations

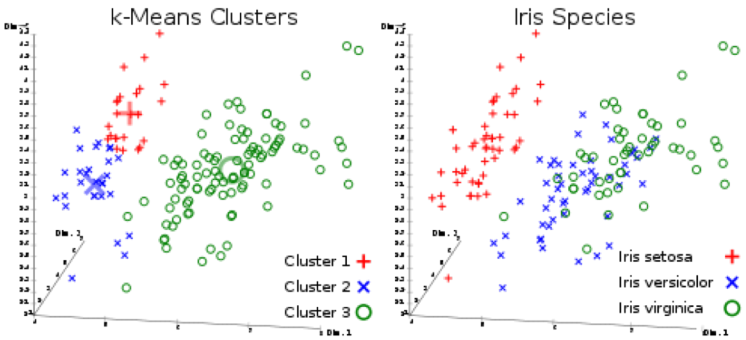
$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \\ 0 & 1 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 30 \\ 40 \\ 80 \\ 20 \end{bmatrix}$$

Approaches

- Uniprocessor Mapping
  - Naïve Method
    - Loops down rows & columns
  - Blocked Method
    - Makes use of spatial locality
- Parallel Mapping
  - 2–D Block Cyclic Distribution
    - Provides better data distribution and load balancing

Application Areas

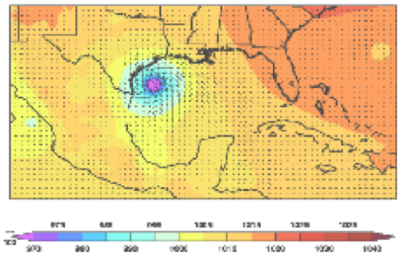
- Linear Algebra
  - LAPACK
  - ATLAS
- Data Mining
  - Streamcluster
  - K–means



Source: http://view.eecs.berkeley.edu

Structured Grids

Computation steps update data in a regular multi--dimensional grid

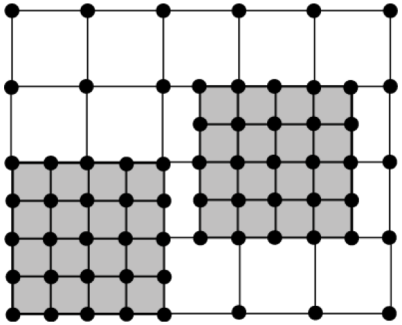


Approaches

- Regular Static Grid
  - Grid dimensions remain as they are from the start
    - High spatial locality, limited temporal locality
    - Parallel processors can be assigned subgrids
- Adaptive Mesh Refinement
  - Overlay higher resolution grids on areas of interest
    - Expensive boundary computations

Application Areas

- Image Processing
  - SRAD
- Physics Simulations
  - HotSpot



Unstructured Grids

Computations that depend on neighbors in an irregular grid

- *Approaches*
  - In particle--particle methods
    - Every point depends on its neighbors
      - O(N): Brute--force method
- *Application Areas*
  - Computational fluid dynamics
  - Belief propagation

MapReduce

Process subsets of data independently and merge results

Approaches

- Manual MapReduce
  - Application--specific "distribute" and "merge" functionality
    - mpiBLAST
- MapReduce Frameworks
  - Ease scaling of embarrassingly parallel applications
    - Hadoop and BOINC

Application Areas

- Distributed Searching

- Sequence Alignment

- Parallel Monte Carlo Simulations

Combinational Logic

Simple computation on large data sets, exhibiting bit–level parallelism

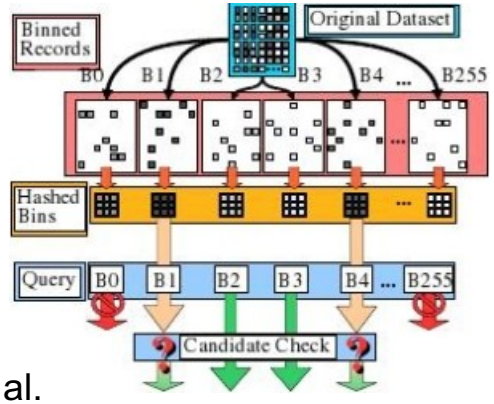
Approaches

- Often naturally parallel
  - Can use pipelining techniques to increase throughput
    - O(N): Naive

Application Areas

- Encryption & Decryption

- Hashing



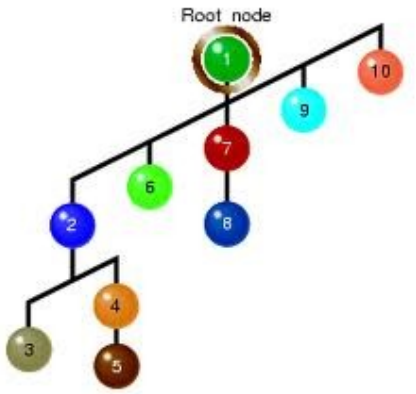
Source:
John Owens et al.

Graph Traversal

Traverse objects and examine them as they are traversed

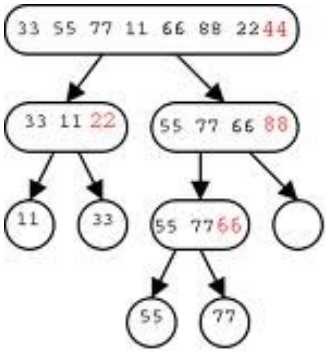
Approaches

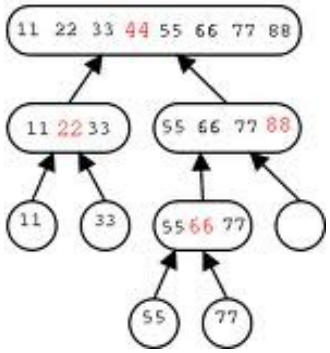
- Wavefront Movement
  - Traversal from single source
    - O(N): Brute--force method
- Divide and Conquer
  - Sorting Algorithms
    - O(Nlog(N)): Quicksort



Application Areas

- Searching
- Sorting
- Collision Detection





Dynamic Programming

Compute solutions by solving simpler overlapping subproblems

Approaches

- Uniprocessor Mapping
  - Solve subproblem once & store
    - Assumes fast store and lookup times of solutions
- Parallel Mapping
  - Solutions to overlapping subproblems communicated or re--computed
    - Trade--off between compute and communicate costs

Application Areas

- Graph problems
  - Floyd's All--Pairs shortest path
  - Bellman--Ford algorithm
- Sequence alignment
  - Needleman--Wunsch
  - Smith--Waterman

Backtracking and Branch & Bound

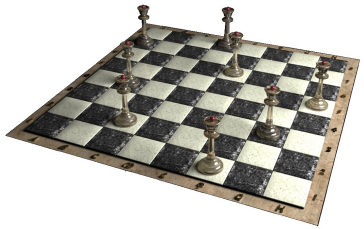
Branch–and–bound algorithms used to solve search and global optimization

Approaches

- Heuristic
- Dynamic Load Balance
  - Divide search space (fairly) among available processors.

Application Areas

- Artificial Intelligence: N–Queens



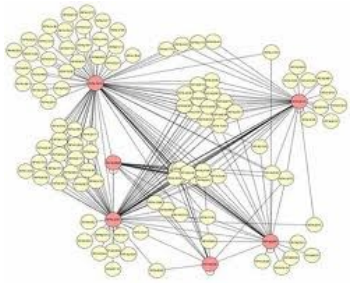
- Integer Linear Programming
- Boolean Satisfiability
- Combinatorial Optimization

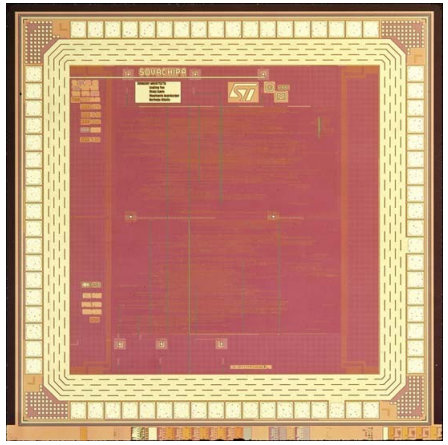
Graphical Models

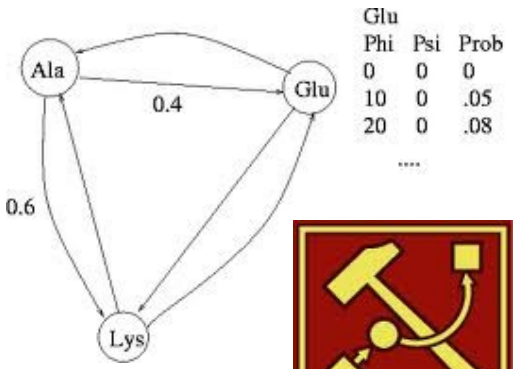
Construct graphs that represent random variables as nodes and conditional dependencies as edges

Application Areas

- Computational Biology
  - Sequence homology search
- Machine Learning
  - Hidden Markov models
- Embedded Computing
  - Viterbi decode





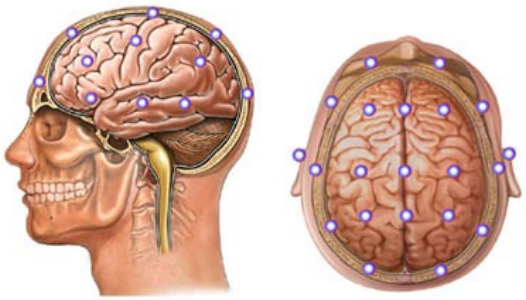


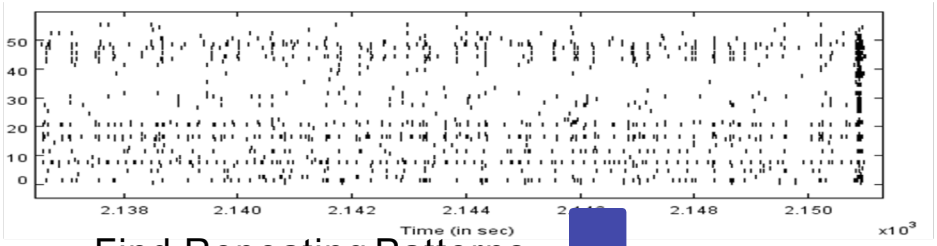
Finite State Machine

Interconnected states which transition between one another

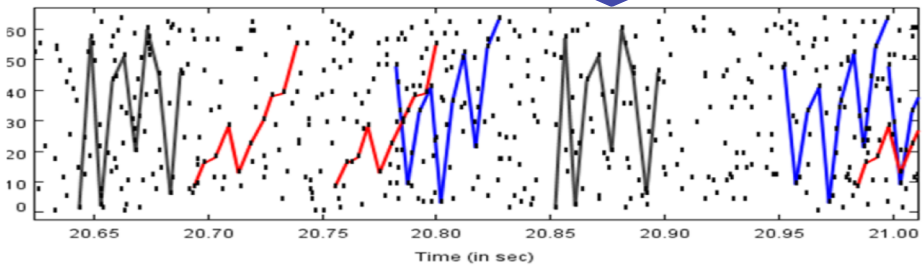
Application Areas

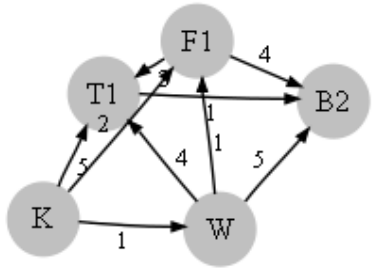
- Video Decoding, Parsing, Compression
- Data Mining → Reverse Engineering the Brain





Find Repeating Patterns





Infer
Network Connectivity

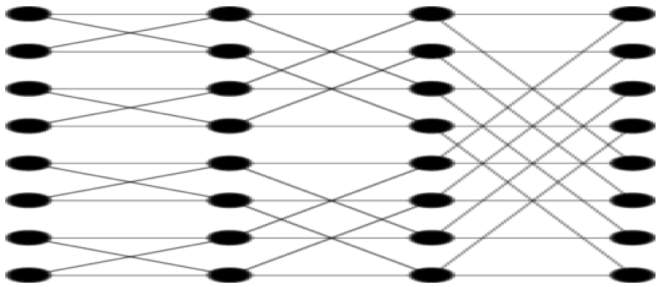
"Temporal Data Mining for Neuroscience," *GPU Computing Gems*, Morgan Kaufmann, Feb. 2011.
"High--Performance Biocomputing for Simulating the Spread of Contagion over Large Contact Networks," *BMC Genomics*, 2011.

Spectral Methods

Spectral domain computations transformed from temporal or spatial domains and solved numerically

Approach

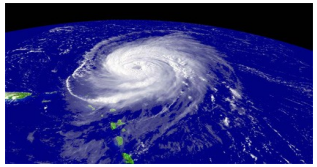
- 'Butterfly' pattern dependencies between stages of transformation
  - MAD operations on complex #
    - O(N log N): FFT



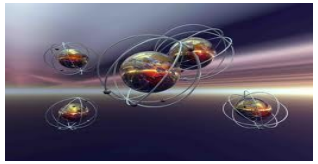
Computational Organization

Application Areas

Fluid Dynamics



Quantum Mechanics



Weather Prediction