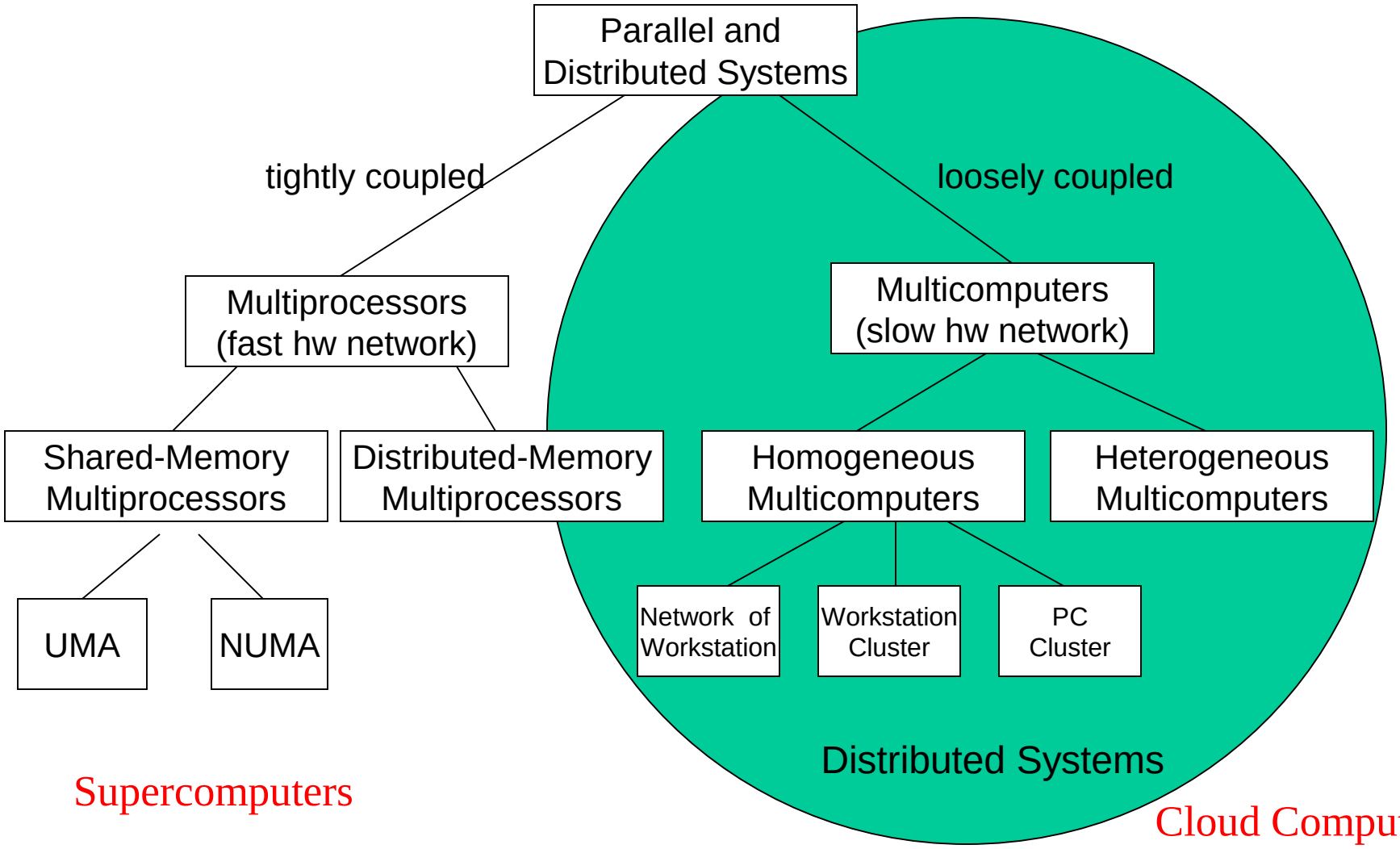
CS546 Parallel and Distributing Processing

- Instructor: Professor Xian-He Sun
  - Email: sun@iit.edu, Phone: (312) 567-5260
  - Office hours: 4:30pm-5:30pm Tuesday, Thursday
    SB235C
- TA: Xiaoyang Lu
  - Email: xlu40@hawk.iit.edu
  - Office hour: 2:30pm-3:30pm MW, SB003
- Blackboard:
  - http://blackboard.iit.edu
- Additional Web site
  - http://www.cs.iit.edu/~sun/cs546.html

Software Support Concepts



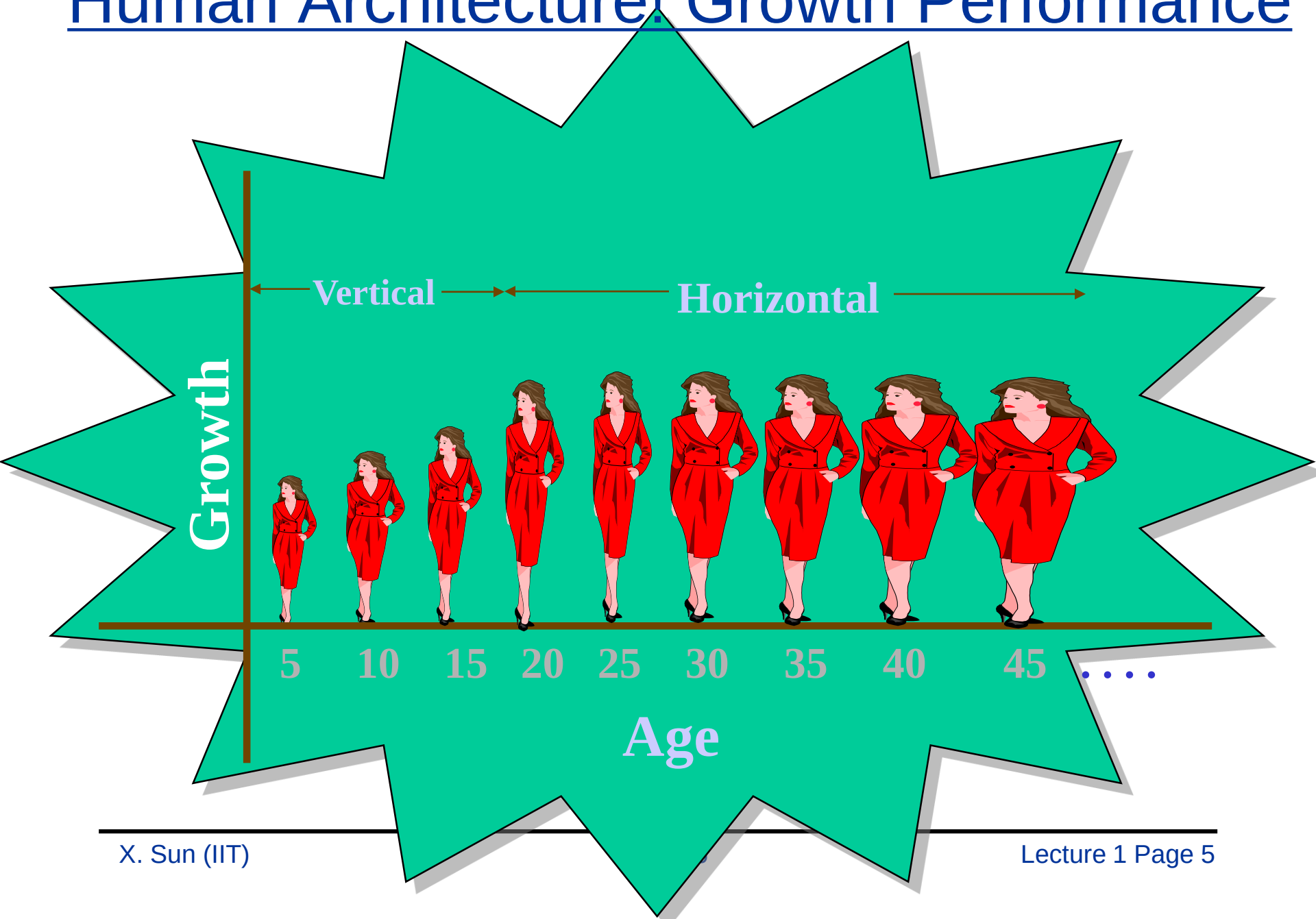
Any Questions?

- What is the current communication consideration?
- What are the contention consideration of static and dynamic connections?

Where's the Parallelism

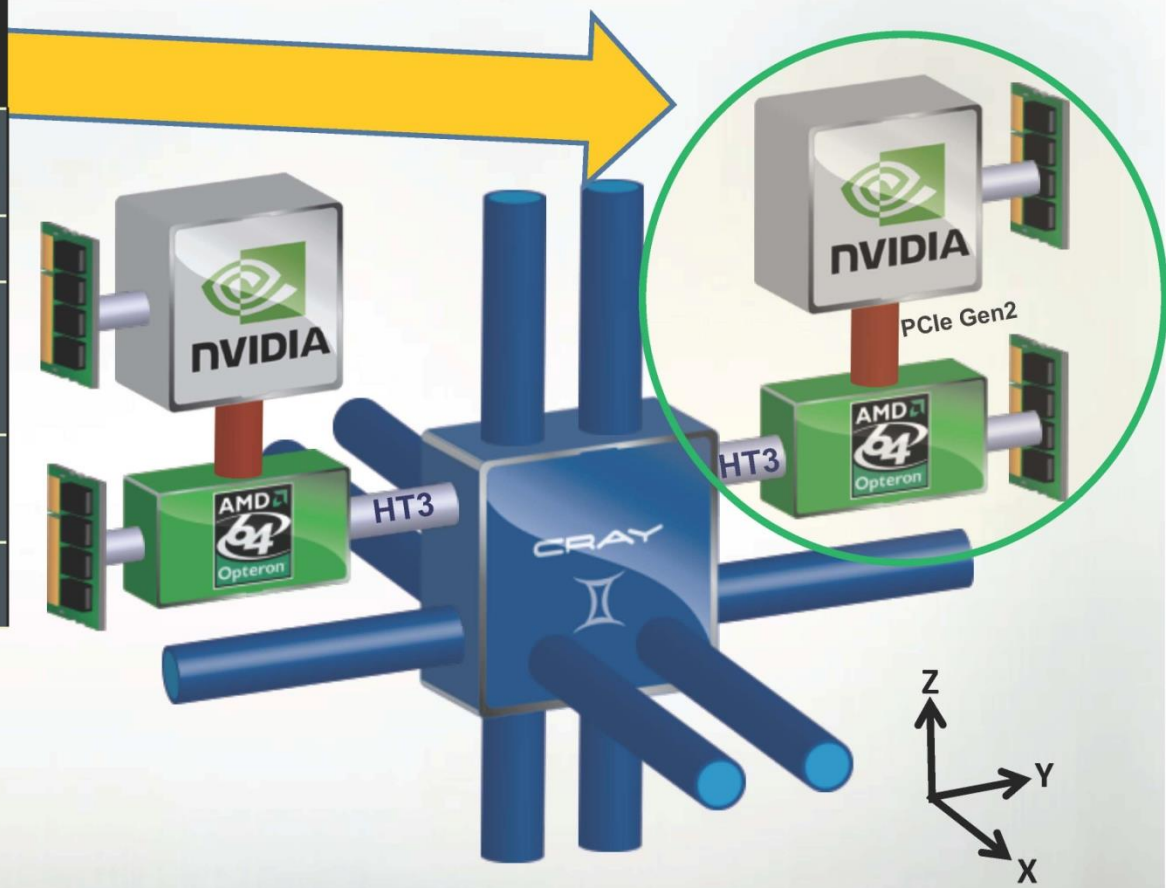
- Internode
  - Multiple nodes
  - Primary level for commodity clusters
  - Secondary level for constellations
- Multi socket, intra-node
  - Routinely 1,2,4,8
  - Heterogeneous computing with accelerators
- Multi-core, intra-socket
  - 8,16 cores per socket
- Multi-thread, intra-core
  - None or two usually
- ILP, intra-core
  - Multiple operations issued per instruction
- Out of order, reservation stations
- Pre-fetching
- Accelerators (GPU)
- Concurrent data access (new, from data viewpoint)

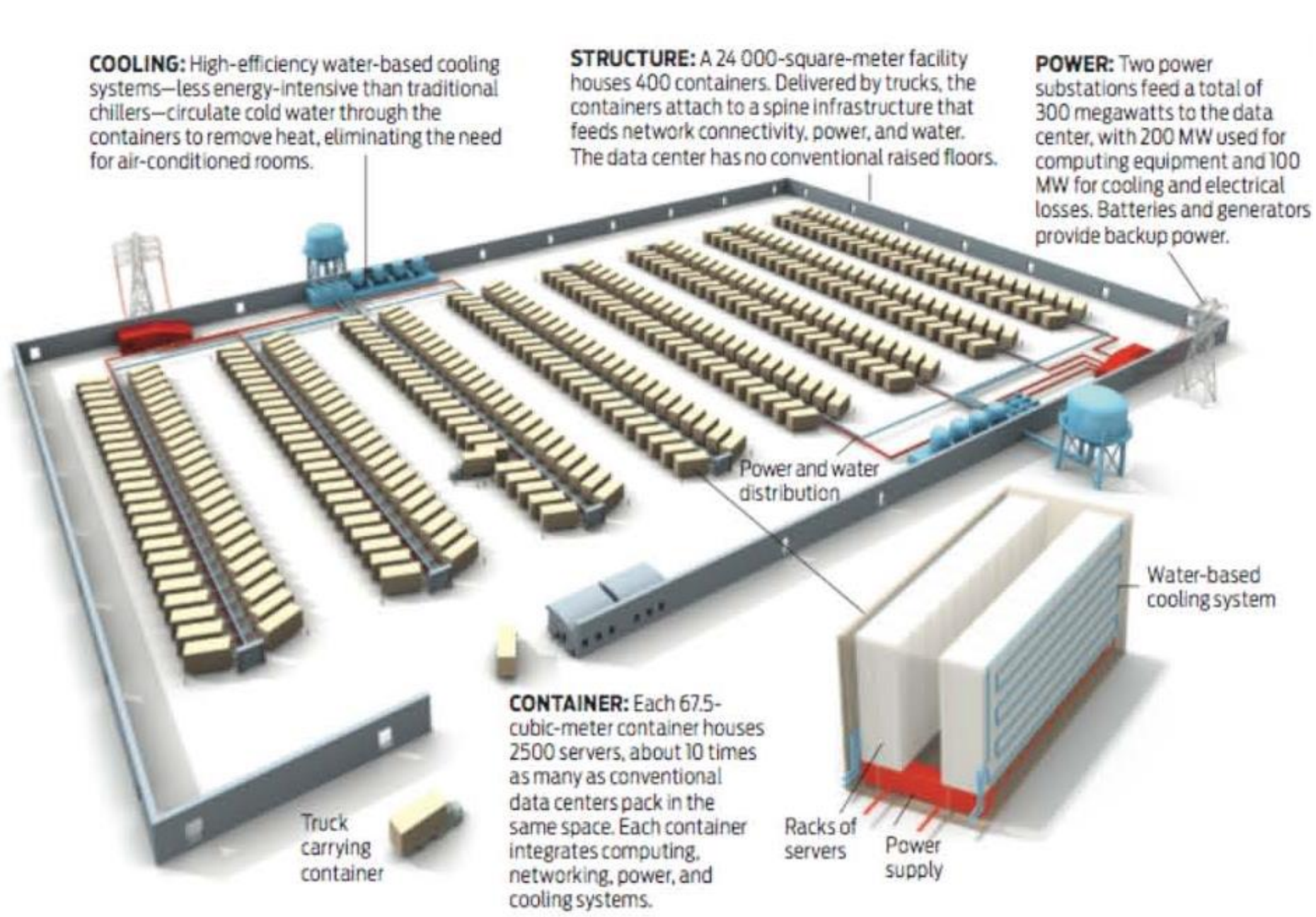
Human Architecture| Growth Performance



Cray XK7 Compute Node

XK7 Compute Node Characteristics
AMD Opteron 6274 Interlagos 16 core processor
Tesla K20x @ 1311 GF
Host Memory 32GB 1600 MHz DDR3
Tesla K20x Memory 6GB GDDR5
Gemini High Speed Interconnect




COOLING: High-efficiency water-based cooling systems—less energy-intensive than traditional chillers—circulate cold water through the containers to remove heat, eliminating the need for air-conditioned rooms.
STRUCTURE: A 24 000-square-meter facility houses 400 containers. Delivered by trucks, the containers attach to a spine infrastructure that feeds network connectivity, power, and water. The data center has no conventional raised floors.
POWER: Two power substations feed a total of 300 megawatts to the data center, with 200 MW used for computing equipment and 100 MW for cooling and electrical losses. Batteries and generators provide backup power.
Power and water distribution
Water-based cooling system
CONTAINER: Each 67.5-cubic-meter container houses 2500 servers, about 10 times as many as conventional data centers pack in the same space. Each container integrates computing, networking, power, and cooling systems.
Truck carrying container
Racks of servers
Power supply

Software Support Concept

- OS, Compiler, Libaries
- Tools
  - Timing and profiling
  - Scheduler
  - Debugger
- Distributed File Systems
- Parallel File Systems
  - Why parallel I/O?
  - I/O libraries and middleware
  - Data distribution

Compilers & Debuggers

- Compilers :
  - Intel C/C++/ Fortran
  - PGI C/C++/ Fortran
  - GNU C/C++/ Fortran
- Libraries :
  - Each compiler is linked against MPICH and
  - Mesh/Grid partitioning software :METIS etc.
  - Math Kernel Libraries
  - Intel MKL, AMD, GNU Scientific Library (GSL)
  - Data format libraries : Net CDF, HDF 5 etc
  - Linear Algebra Packages : BLAS, LAPACK etc
- Debuggers
  - Gdb
  - Totalview
- Performance & Profiling tools :
  - PAPI
  - TAU
  - Gprof
  - perfctr

“Dependence and Parallelism”

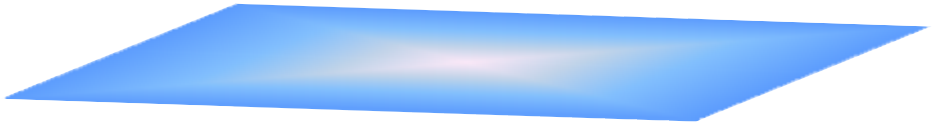
Outline

- Flavors of parallelism
- Data dependence
  - True dependence
  - Anti-dependence
  - Output dependence
  - Loop-carried dependence

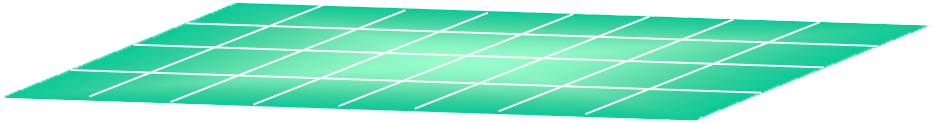
Algorithms

- Divide a given problem into sub-problems
- Sub-problems must have a high degree of concurrency
- Must minimize synchronization among sub-problems
- Key aspect?

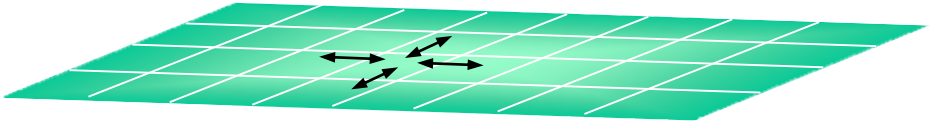
Parallel Processing



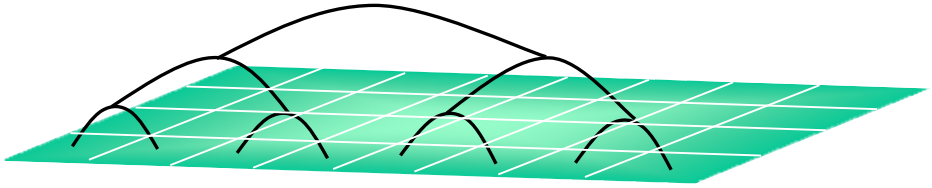
Partition Workload



Communication for Coordination



Overhead Increases with the Ensemble Size (scalability)



Style of Parallel Processing

- Concurrent Parallel Processing
  - Concurrency exploits spatial parallelism by utilizing several processors executing multiple independent tasks simultaneously
- Pipelined Parallel Processing
  - Pipelining exploits temporal parallelism in which each processor operates on its input and passes its output data to the succeeding processor as the latter's input
  - Vector processing is pipelined processing

Style of Parallel Processing

- Partitioning
  - Data partitioning
  - Function partitioning
- Parallelism
  - Concurrent
  - Pipelined
- Example

  For i=1 to 1000 do

  D[i] = L*A[i-1] + B[i]*C[i+1]

Data and Task Parallelization

- Data parallel:

```
For i=1 to 499 do
 D[i] = L*A[i-1] + B[i]*C[i+1]
For i=500 to 1000 do
 D[i] = L*A[i-1] + B[i]*C[i+1]
```

- Task parallel:

```
For i=0 to 1000 do          For i=0 to 1000 do
   D[i]=L*A[I-1]               E[I]=B[i]*C[i+1]
…
For i=0 to 1000 do
   D[I]=D[i]+E[i]
```

Pipelining

- Example

  For i=1 to 1000 do

  D[i] = L*A[i-1] + B[i]*C[i+1]

- T1 L*A[0]
- T2 L*A[1], B[1]*C[2]
- T3 L*A[2], B[2]*C[3], L*A[0]+B[1]*C[2]
- T4 L*A[3], B[3]*C[4], L*A[1]+B[2]*C[3]
- …….

Data Dependence & Parallelization

- Coarse Grain computing

```
for (i=0;i<500;i++)                for (i=500;i<1000;i++)
  a[i]=b[i]+c[i]                     a[i]=b[i]+c[i]
```

- Fine Grain computing:

```
a[0]=b[0]+c[0];
a[1]=b[1]+c[1];
…..
a[999]=b[999]+c[999];
```

- Can each iteration be executed in parallel?

Approaches to Parallel Programming

- Write sequential program, leave it to compiler to parallelize
- Write sequential program that calls parallel subroutines (e.g. matlab)
- Write a parallel program for parallel processing
  - Shared memory programming & Multithreaded
  - Message passing
  - Data parallel languages

Parallelism

- Ability to execute different parts of a program concurrently on different processors
- Processors execute independently: no control over order of execution between processors

- Goal: performance

When can 2 statements execute in parallel?

- On one processor:

  statement 1;

  statement 2;

- On two processors:

  processor1:

  statement1;

  processor2:

  statement2;

When can 2 statements execute in parallel?

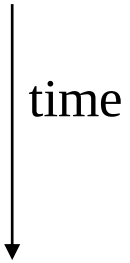
- Possibility 1

Processor1:

statement1;

Processor2:

statement2;



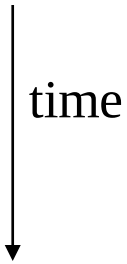
- Possibility 2

Processor1:

statement1;

Processor2:

statement2:



When can 2 statements execute in parallel?

- Their order of execution must not matter!

- In other words,

  statement1; statement2;

  must be equivalent to

  statement2; statement1;

Example 1

```
a = 1;
b = 2;
```

- Statements can be executed in parallel.

Example 2

```
a = 1;
b = a;
```

- Statements cannot be executed in parallel
- Program modifications may make it possible.

Example 3

```
a = f(x);
b = a;
```

- May not be wise to change the program (sequential execution would take longer).

Example 4

```
b = a;
a = 1;
```

- Statements cannot be executed in parallel.

Example 5

```
a = 1;
a = 2;
```

- Statements cannot be executed in parallel.

True dependence

Statements S1, S2

S2 has a true dependence on S1

iff

S2 reads a value written by S1

Anti-dependence

Statements S1, S2.

S2 has an anti-dependence on S1

iff

S2 writes a value read by S1.

Output Dependence

Statements S1, S2.

S2 has an output dependence on S1

iff

S2 writes a variable written by S1.

When can 2 statements execute in parallel?

S1 and S2 can execute in parallel

iff

there are no dependences between S1 and S2

- true dependences
- anti-dependences
- output dependences

Some dependences can be removed.

Example 6

- Most parallelism occurs in loops.

```
for(i=0; i<100; i++)
    a[i] = i;
```

- No dependences.
- Iterations can be executed in parallel.

Example 7

```
for(i=0; i<100; i++) {
    a[i] = i;
    b[i] = 2*i;
}
```

Iterations and statements can be executed in parallel.

Example 8

```
for(i=0;i<100;i++) a[i] = i;
for(i=0;i<100;i++) b[i] = 2*i;
```

Iterations and loops can be executed in parallel.

Example 9

```
for(i=0; i<100; i++)
    a[i] = a[i] + 100;
```

- There is a dependence … on itself!
- Loop is still parallelizable.

Example 10

```
for( i=0; i<100; i++ )
  a[i] = f(a[i-1]);
```

- Dependence between a[i] and a[i-1].
- Loop iterations are not parallelizable.

Loop-carried dependence

- A loop-carried dependence is a dependence that is present only if the statements are part of the execution of a loop.
- Otherwise, we call it a loop-independent dependence.
- Loop-carried dependences prevent loop iteration parallelization.

Example 11

```
for(i=0; i<100; i++ )
  for(j=0; j<100; j++ )
      a[i][j] = f(a[i][j-1]);
```

- Loop-independent dependence on i.
- Loop-carried dependence on j.
- Outer loop can be parallelized, inner loop cannot.

Example 12

```
for( j=0; j<100; j++ )
  for( i=0; i<100; i++ )
      a[i][j] = f(a[i][j-1]);
```

- Inner loop can be parallelized, outer loop cannot.
- Less desirable situation.
- Loop interchange is sometimes possible.

Level of loop-carried dependence

- Is the nesting depth of the loop that carries the dependence.
- Indicates which loops can be parallelized.

Be careful … Example 13

```
printf("a");
printf("b");
```

Statements have a hidden output dependence due to the output stream.

Be careful … Example 14

```
a = f(x);
b = g(x);
```

Statements could have a hidden dependence if f and g update the same variable.

Be careful … Example 15

```
for(i=0; i<100; i++)
  a[i+10] = f(a[i]);
```

- Dependence between a[10], a[20], …
- Dependence between a[11], a[21], …
- …
- Some parallel execution is possible.

Be careful … Example 16

```
for( i=1; i<100;i++ ) {
        a[i] = …;
        ... = a[i-1];
}
```

- Dependence between a[i] and a[i-1]
- Complete parallel execution impossible
- Pipelined parallel execution possible

Be careful … Example 17

```
for( i=0; i<100; i++ )
        a[i] = f(a[indexa[i]]);
```

- Cannot tell for sure.
- Parallelization depends on user knowledge of values in indexa[ ].
- User can tell, compiler cannot.

An aside

- Parallelizing compilers analyze program dependences to decide parallelization.
- In parallelization by hand, user does the same analysis.
- Compiler more convenient and more correct
- User more powerful, can analyze more patterns.

To remember

- Statement order must not matter.
- Statements must not have dependences.
- Some dependences can be removed.
- Some dependences may not be obvious.

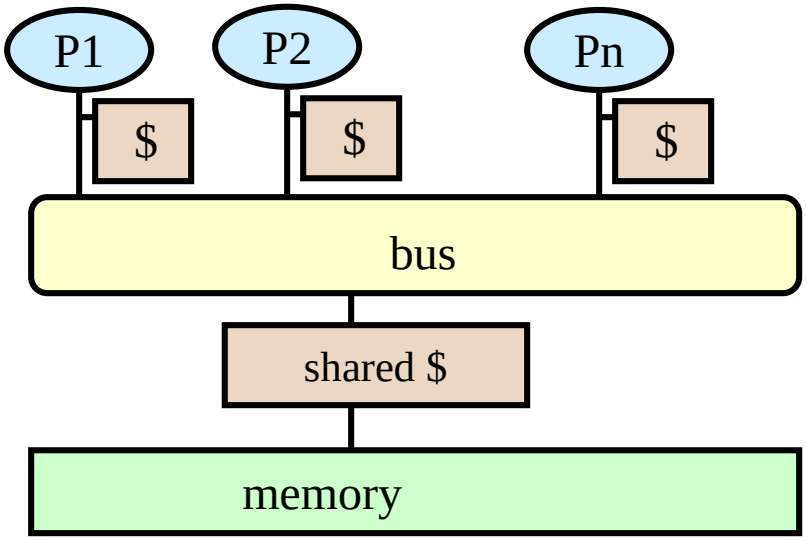
Summary

- No dependence (can run in parallel)
  S1: X=K+3;
  S2: Y=Z*5;
- True dependence (cannot run in parallel)
  S1: X=3;
  S2: Y=X*5;
- Anti dependence (cannot run in parallel)
  S1: Y=X*4;
  S2: X=3;
- Output dependence (cannot run in parallel)
  S1: X=Y*4;
  S2: X=3;
- Loop-carried dependence

Parallel Programming Models

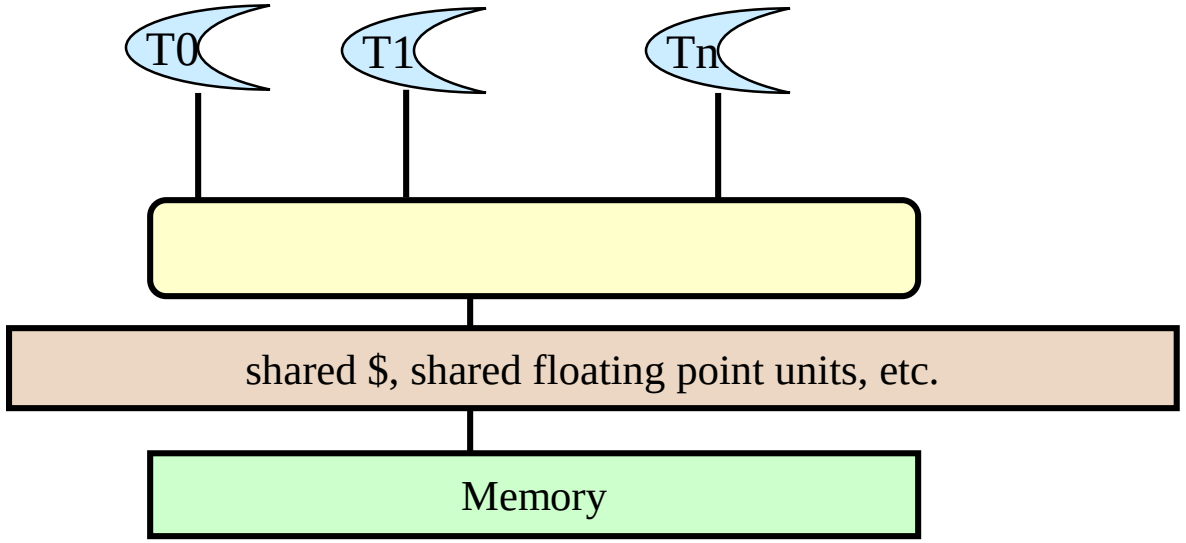
Machine Model 1: Shared Memory

- Processors all connected to a large shared memory
  - Typically called Symmetric Multiprocessors (SMPs)
  - Multicore chips, except that all caches are shared
  - E.g., UMA
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Cost: much cheaper to access data in cache than main memory.



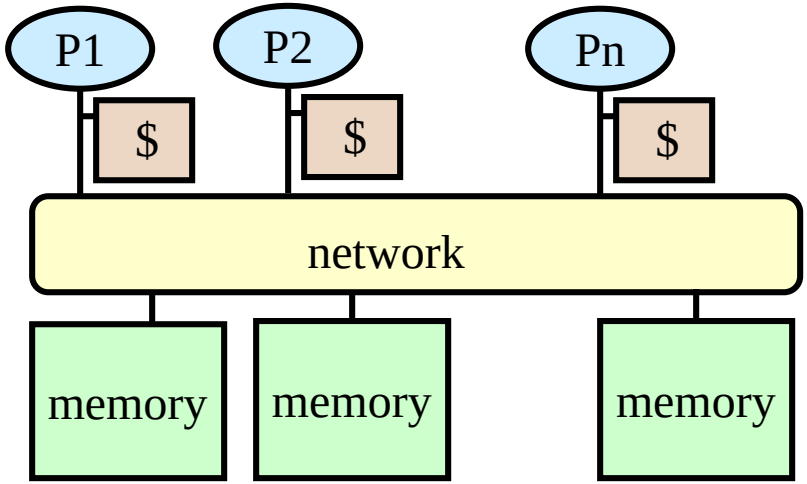
Machine Model 1b: Multithreaded Processor

- Multiple thread “contexts” without full processors
- Memory and some other state is shared
- Sun Niagra processor
  - Up to 64 threads all running simultaneously (8 threads x 8 cores)
  - In addition to sharing memory, they share floating point units
- Cray MTA and Eldorado processors (for HPC)



Machine Model 1c: Distributed Shared Memory

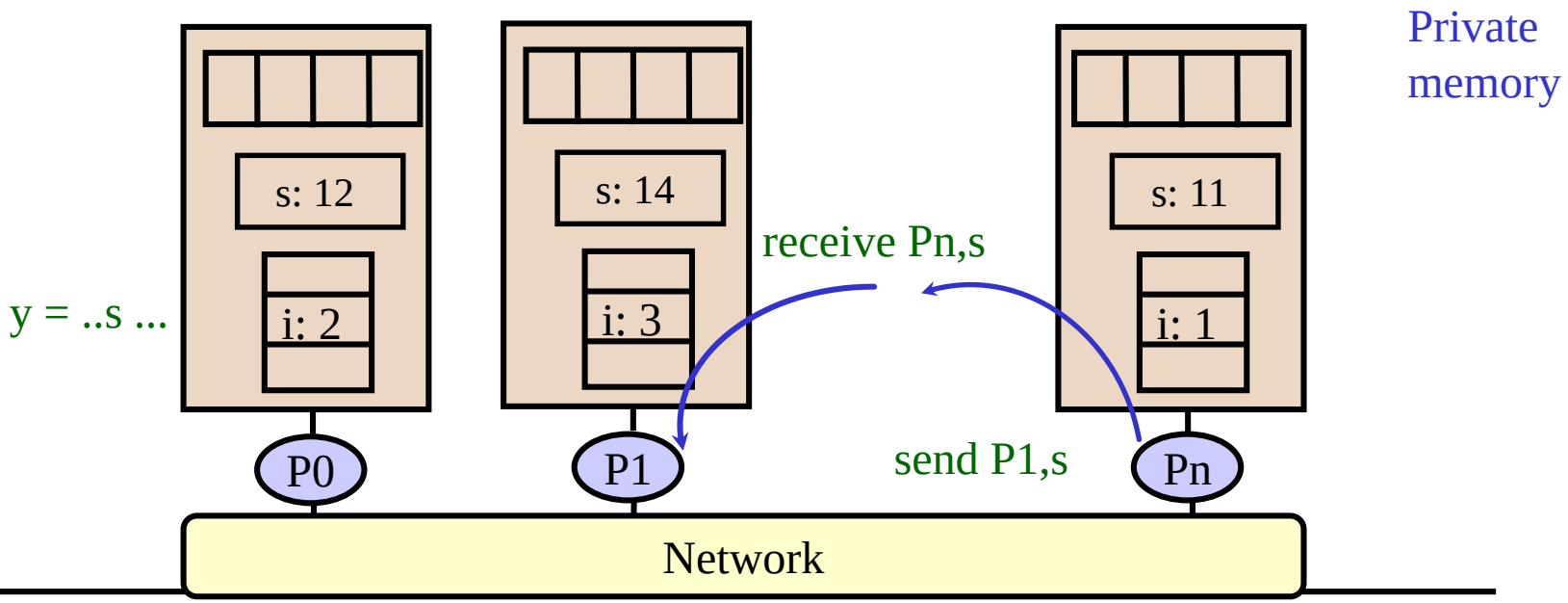
- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
  - E.g., NUMA
- SGI is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is *cache coherency protocols*



Cache lines (pages) must be large to amortize overhead
➔
locality still critical to performance

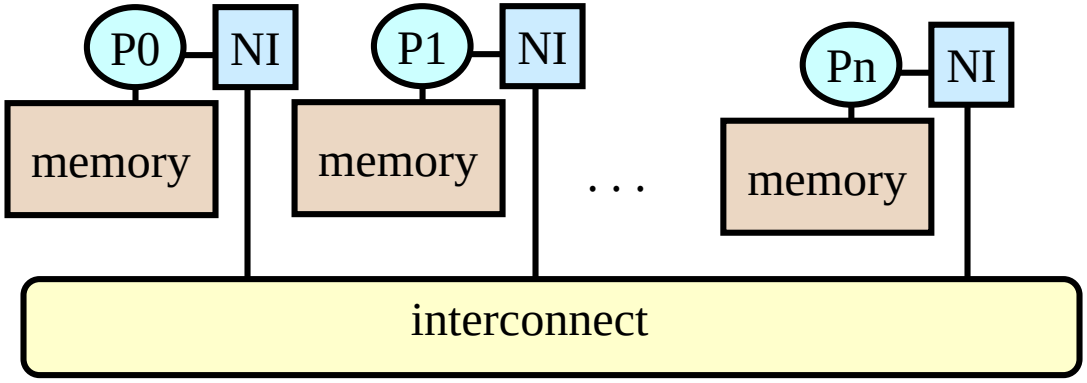
Programming Model 2: Message Passing

- Program consists of a collection of named processes
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data
  - Logically shared data is partitioned over local processes.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event



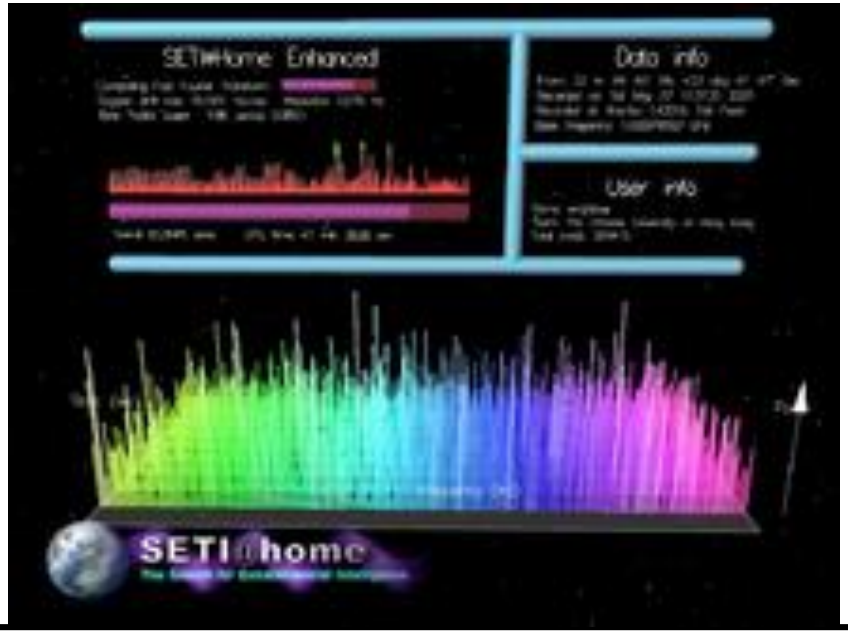
Machine Model 2a: Distributed Memory

- Cray XT4, XT 5
- PC Clusters (Berkeley NOW, Beowulf)
- IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor's memory
  - Aka "distributed address space machines"
- Each "node" has a Network Interface (NI) for all communication and synchronization.



Machine Model 2b: Internet/Cloud

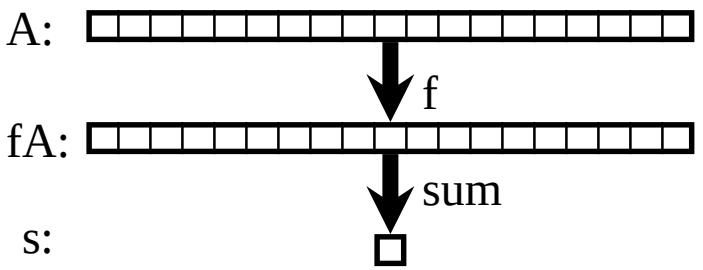
- SETI@Home: Running on 500,000 PCs
  - ~1000 CPU Years per Day
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope



Programming Model 3: Data Parallel

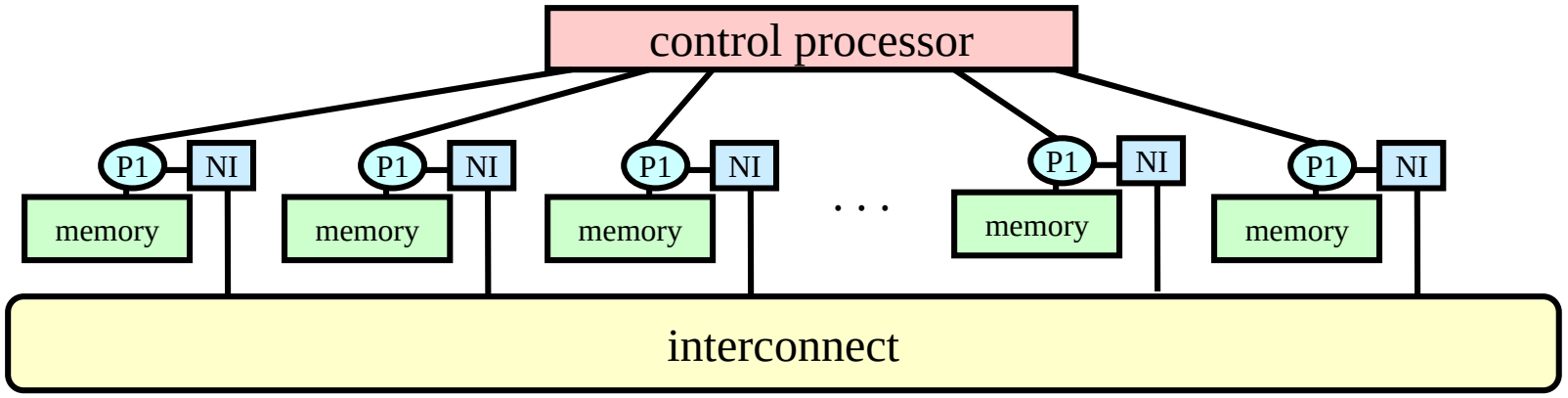
- Single thread of control consisting of parallel operations.
- Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  - Communication is implicit in parallel operators
  - Elegant and easy to understand and reason about
  - Coordination is implicit – statements executed synchronously
- Drawbacks:
  - ???

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



Machine Model 3a: SIMD System

- A large number of (usually) small processors.
  - A single “control processor” issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.
- Originally machines were specialized to scientific computing, few made (e.g.,CM2)
- Programming model can be implemented in the compiler



Machine Model 3b: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specific large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel
- Historically important
  - Overtaken by MPPs in the 90s
- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors
    - SSE, SSE2 (Intel: Pentium/IA64)
  - At a larger scale in GPUs
- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn't have to

Programming Model 4: Hybrids

- These programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice verse
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise
- For better or worse….

Together…

- Parallel programming models
  - Shared memory (Pthreads and openMP)
  - Message passing (MPI)
  - Massive data parallelism (GPU programming)
  - Hybrid
- Distributed programming models
  - Cloud programming, MapReduce, Spark