Lecture 9
Thread Level Parallelism (3)

EEC 171 Parallel Architectures

John Owens
UC Davis

Credits

- © John Owens / UC Davis 2007–17.
- Thanks to many sources for slide material: Computer Organization and Design (Patterson & Hennessy) © 2005, Computer Architecture (Hennessy & Patterson) © 2007, Inside the Machine (Jon Stokes) © 2007, © Dan Connors / University of Colorado 2007, © Kathy Yelick / UCB 2007, © Wen-Mei Hwu/David Kirk, University of Illinois 2007, © David Patterson / UCB 2003–7, © John Lazzaro / UCB 2006, © Mary Jane Irwin / Penn State 2005, © John Kubiatowicz / UCB 2002, © Krste Asinovic/Arvind / MIT 2002, © Morgan Kaufmann Publishers 1998.

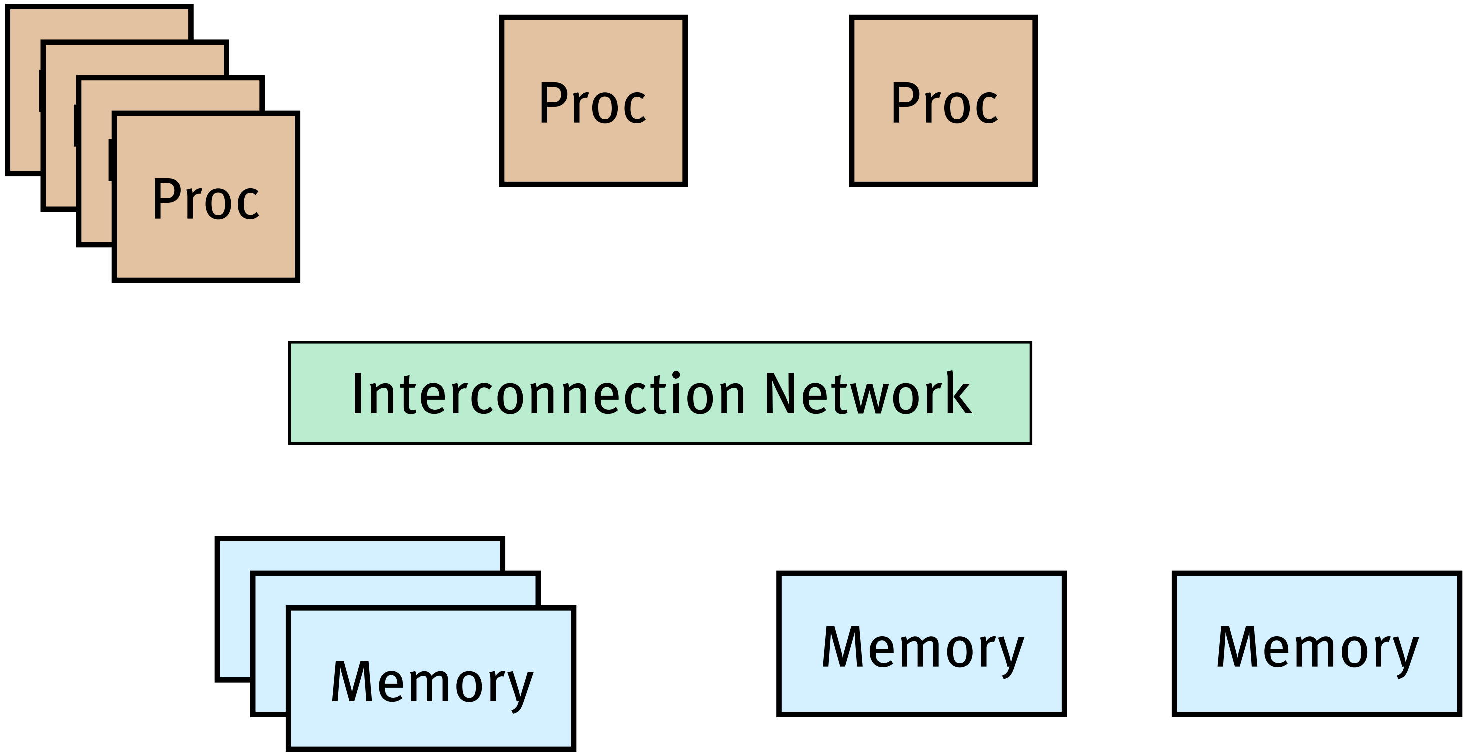
Transition to Today's Topic

- On Monday we looked at machines that were optimized for running many thread-parallel programs in parallel.
- Today we are looking at how to run one program with many threads in parallel.
    - Why is this harder?

Outline

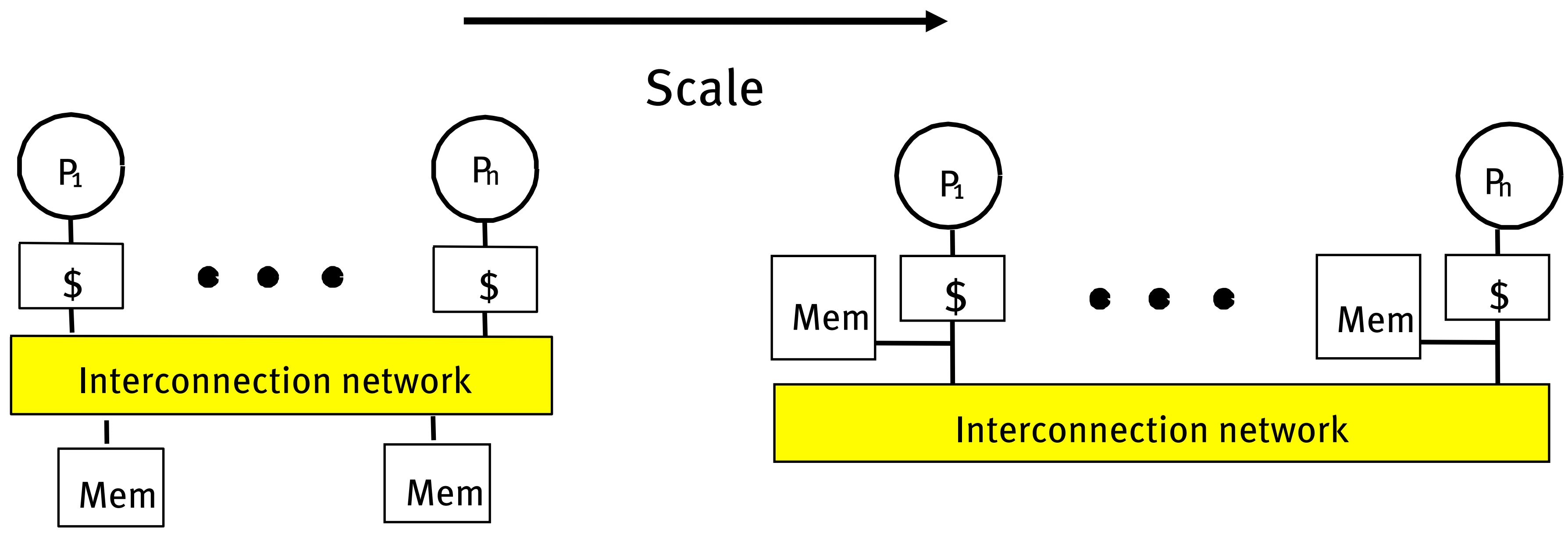
- Overview of parallel machines (~hardware) and programming models (~software)
  - Shared memory
  - Shared address space
  - Message passing
  - Data parallel
  - Clusters of SMPs
  - Grid
- Parallel machine may or may not be tightly coupled to programming model
  - Historically, tight coupling
  - Today, portability is important
- Trends in real machines

A generic parallel architecture



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

Centralized vs. Distributed Memory



Centralized Memory

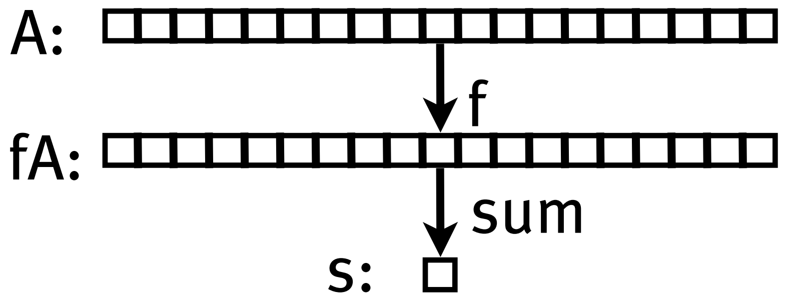
Distributed Memory

Simple Example

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

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

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

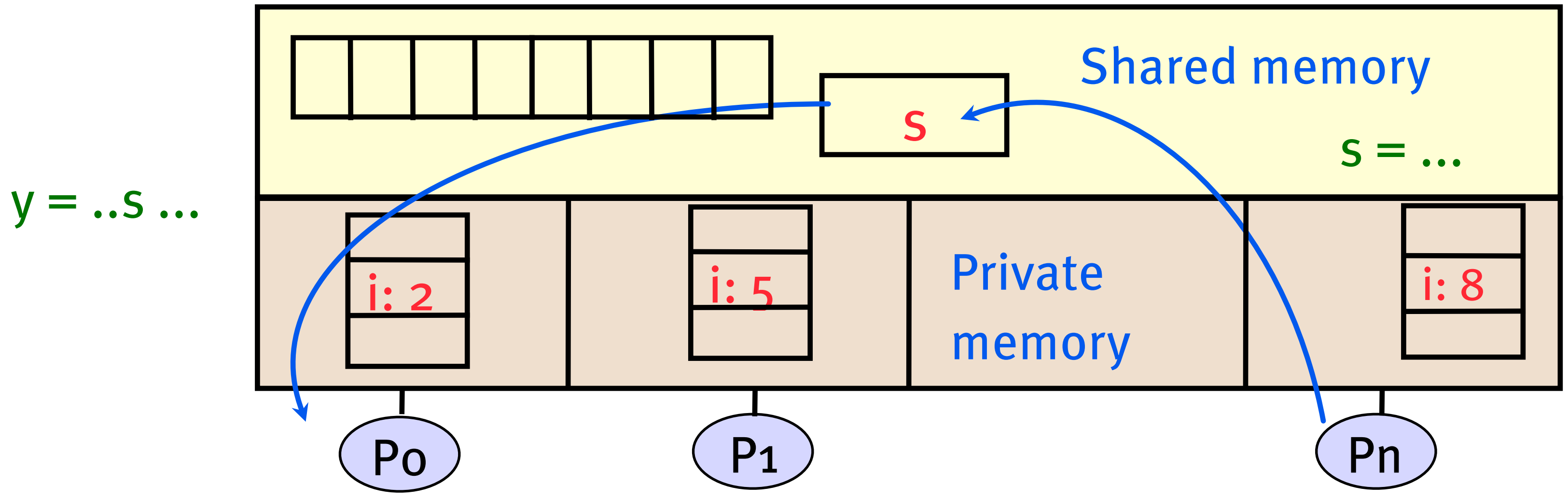


- Questions:
  - Where does A live? All in single memory? Partitioned?
  - How do we divide the work among processors?
  - How do processors cooperate to produce a single result?

Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads *communicate* implicitly by writing and reading shared variables.
  - Threads *coordinate* by synchronizing on shared variables

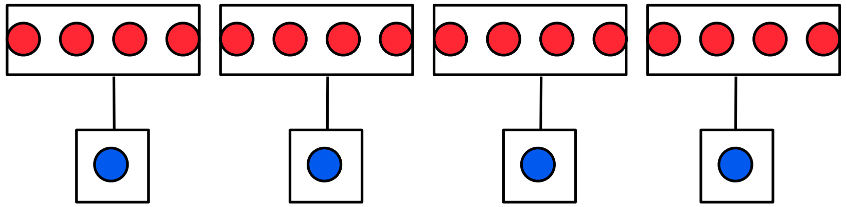
Shared Memory



Simple Example

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

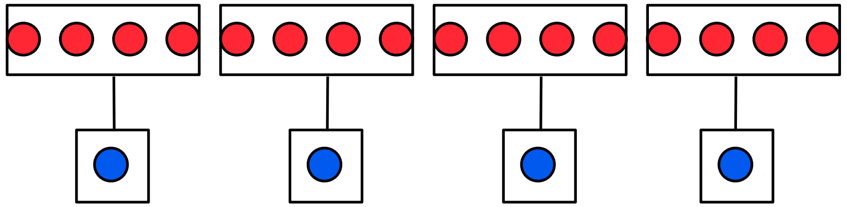
- Shared memory strategy:
  - small number $p \ll n$=size(A) processors
  - attached to single memory



- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
  - Each computes independent "private" results and partial sum.
  - Collect the p partial sums and compute a global sum.

Simple Example

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



- Two Classes of Data:
  - Logically Shared
    - The original *n* numbers, the global sum.
  - Logically Private
    - The individual function evaluations.
    - What about the individual partial sums?

Shared Memory “Code” for Computing a Sum

```
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])
```

- Each thread is responsible for half the input elements
- For each element, a thread adds that element to the a shared variable *s*
- When we’re done, *s* contains the global sum

Shared Memory "Code" for Computing a Sum

```
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])
```

- Problem is a race condition on variable *s* in the program
- A race condition or data race occurs when:
  - Two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Shared Memory Code for Computing a Sum

A	3	5

f = square

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

Thread 1		Thread 2	
....		...	
compute f([A[i]) and put in reg0	9	compute f([A[i]) and put in reg0	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 is the square function, and s=0 initially
- For this program to work, s should be 34 at the end
- but it may be 34, 9, or 25 (how?)
- The atomic operations are reads and writes
- += operation is not atomic
- All computations happen in (private) registers

Improved Code for Computing a Sum

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

```
Thread 1

  local_s1= 0
  for i = 0 to n/2-1
      local_s1 = local_s1 + f(A[i])
  s = s + local_s1
```

```
Thread 2

  local_s2 = 0
  for i = n/2 to n-1
      local_s2 = local_s2 + f(A[i])
  s = s + local_s2
```

- 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
- Is this safe?

Improved Code for Computing a Sum

```
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);
```

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

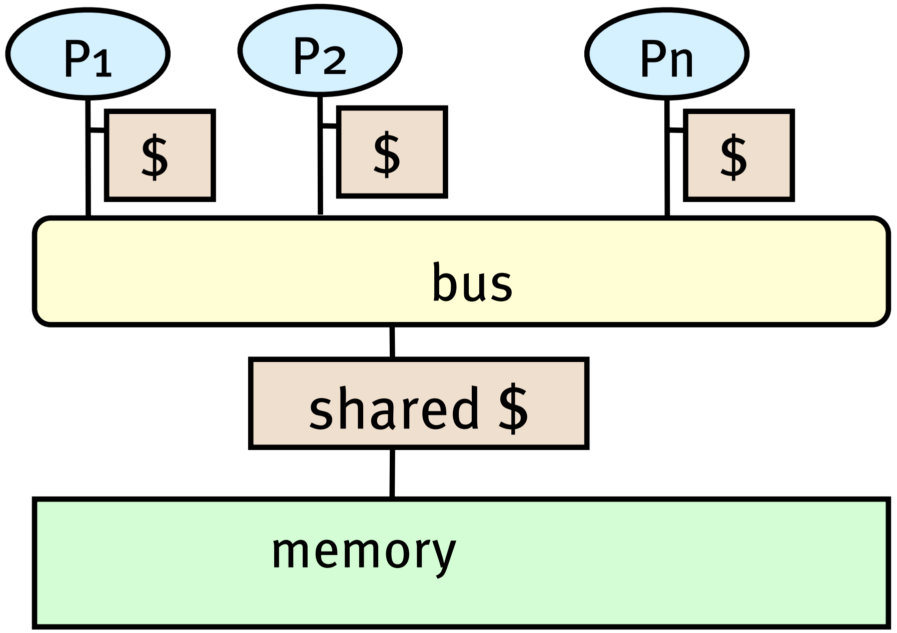
```
Thread 2

  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)

Machine Model 1a: Shared Memory

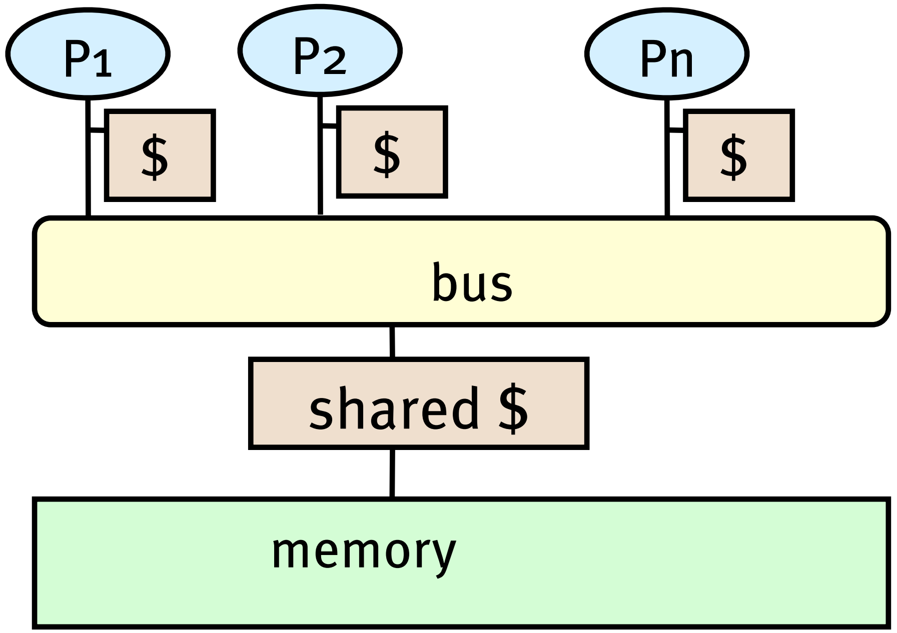
- Processors all connected to a large shared memory
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun (Oracle), HP, Intel, IBM SMPs (nodes of Millennium, SP)
  - Multicore chips, except that caches are often shared in multicores



Note: $ = cache

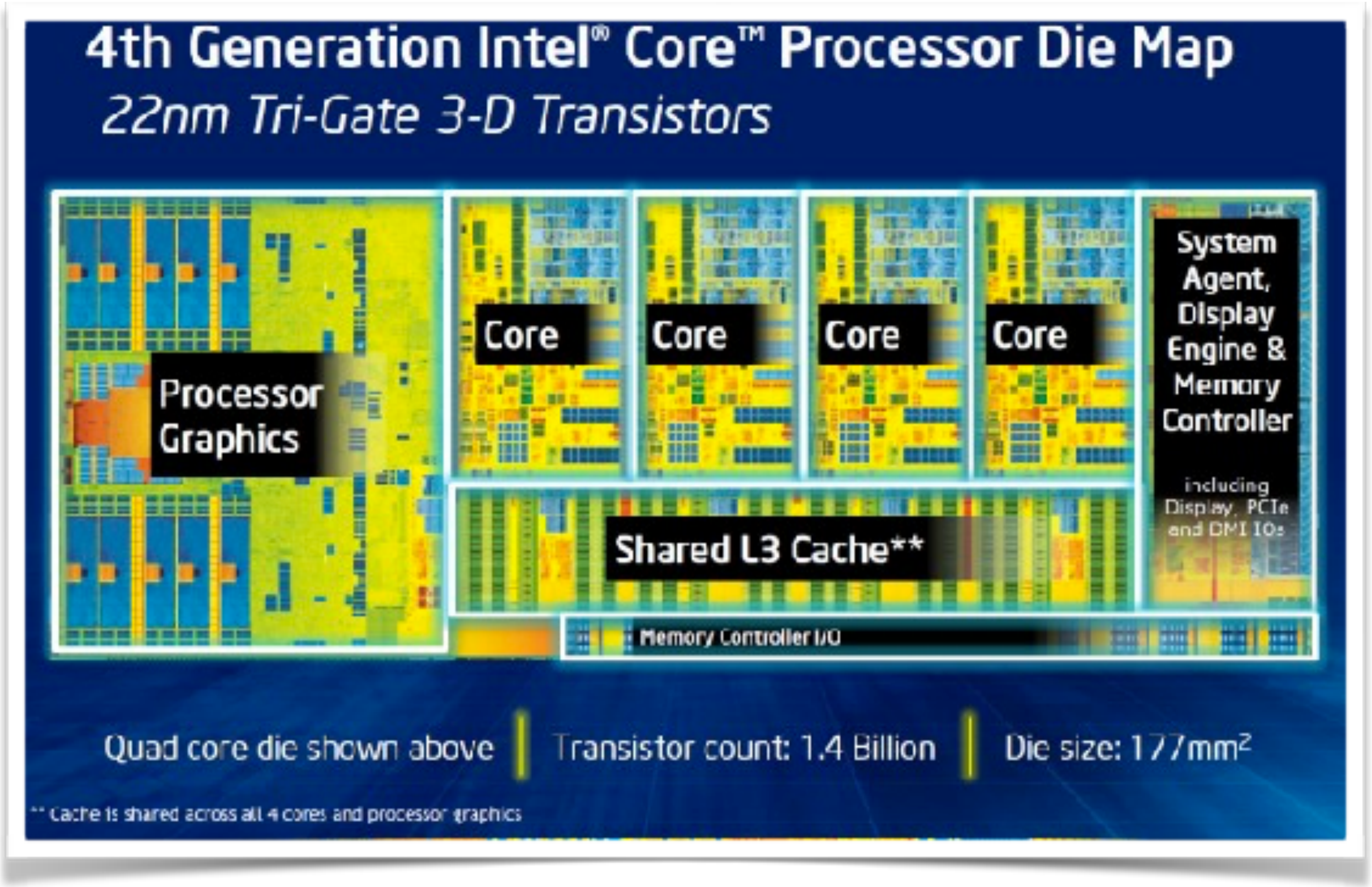
Machine Model 1a: Shared Memory

- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory.



Note: $ = cache

Intel Haswell


4th Generation Intel® Core™ Processor Die Map
22nm Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O
Quad core die shown above
Transistor count: 1.4 Billion
Die size: 177mm²
** Cache is shared across all 4 cores and processor graphics

- Private L1/L2 per core, shared L3, arbitration logic
- Saves power: share data without going to memory

Problems Scaling Shared Memory Hardware

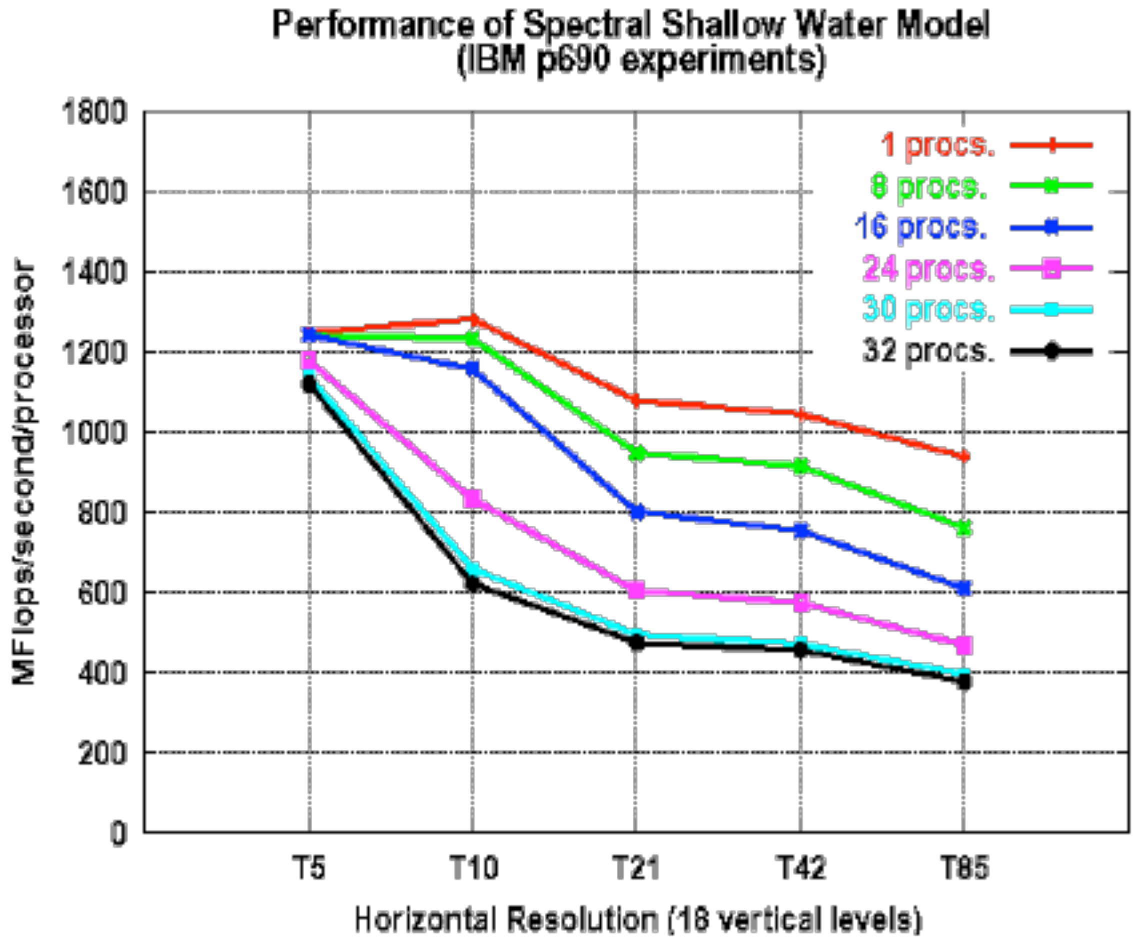
- Why not put more processors on (with larger memory?)
    - The memory bus becomes a bottleneck
        - We’re going to look at interconnect performance in a future lecture. For now, just know that “busses are not scalable”.
    - Caches need to be kept coherent

Problems Scaling Shared Memory Hardware

- Example from a Parallel Spectral Transform Shallow Water Model (PSTSWM) demonstrates the problem
  - Experimental results (and slide) from Pat Worley at ORNL
  - This is an important kernel in atmospheric models
    - 99% of the floating point operations are multiplies or adds, which generally run well on all processors
    - But it does sweeps through memory with little reuse of operands, so uses bus and shared memory frequently
  - These experiments show serial performance, with one “copy” of the code running independently on varying numbers of procs
    - The best case for shared memory: no sharing
    - But the data doesn’t all fit in the registers/cache

Example: Problem in Scaling Shared Memory

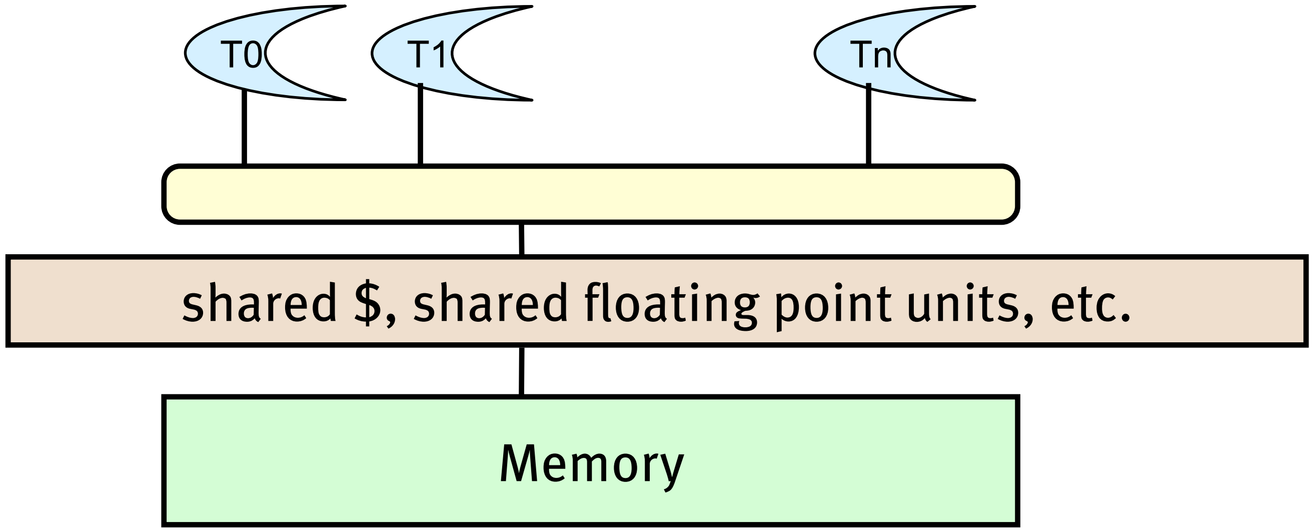
- Performance degradation is a "smooth" function of the number of processes.
- No shared data between them, so there should be perfect parallelism.
- (Code was run for a 18 vertical levels with a range of horizontal sizes.)
- From Pat Worley, ORNL via Kathy Yelick, UCB



Process scaling on IBM p690

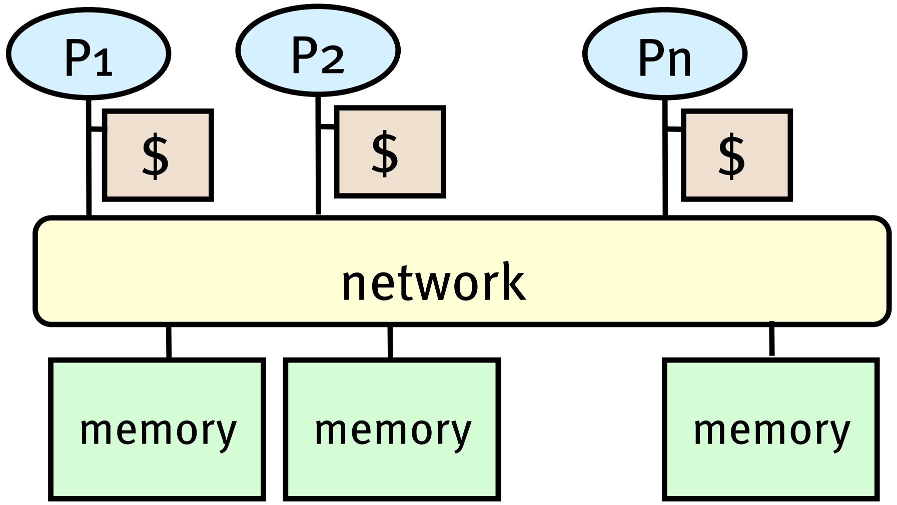
Machine Model 1b: Multithreaded Processor

- Multiple thread "contexts" without full processors per thread
- Memory and some other state is shared
- Sun Niagara processor (for servers)
    - Up to 32 threads all running simultaneously
    - In addition to sharing memory, they share floating point units
    - Why? Switch between threads for long-latency memory operations
- Cray MTA and Eldorado processors (for HPC)



Machine Model 1c: Distributed Shared Memory

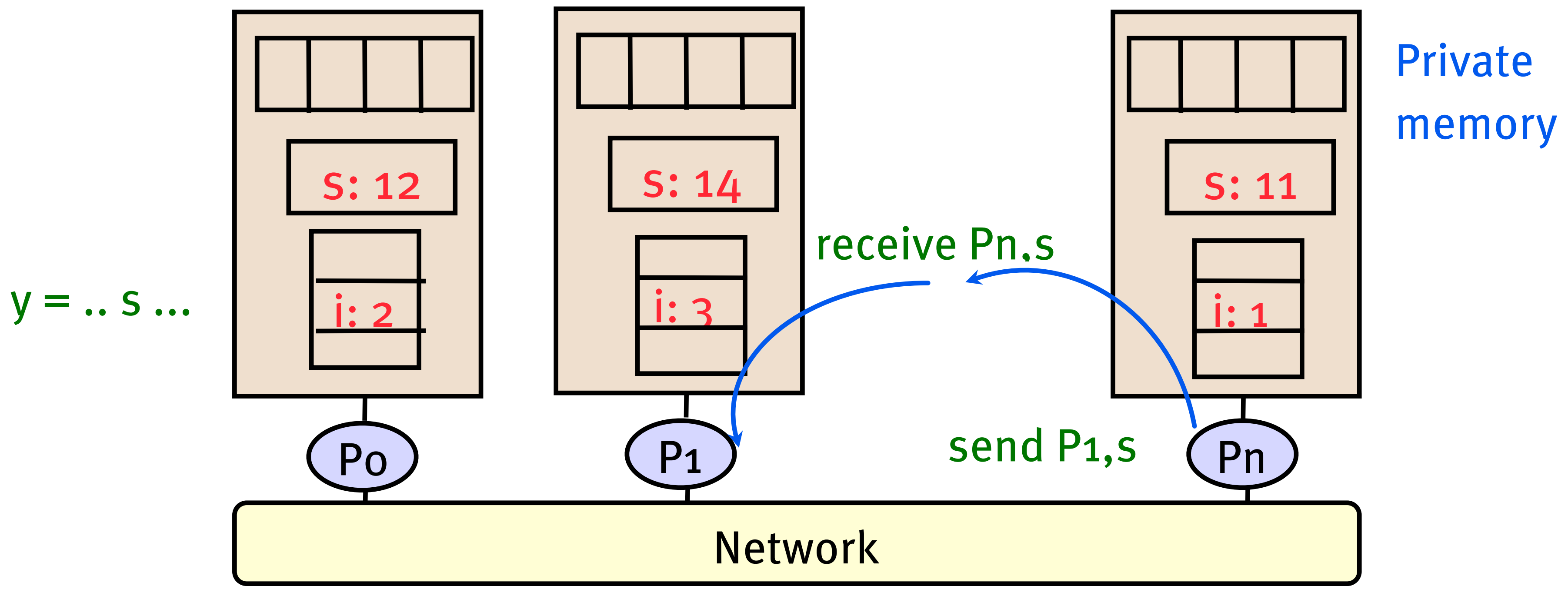
- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI Origin is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is cache coherency protocols—how to keep cached copies of the same address consistent (next lecture)



Cache lines (pages) must be large to amortize overhead—locality is critical to performance

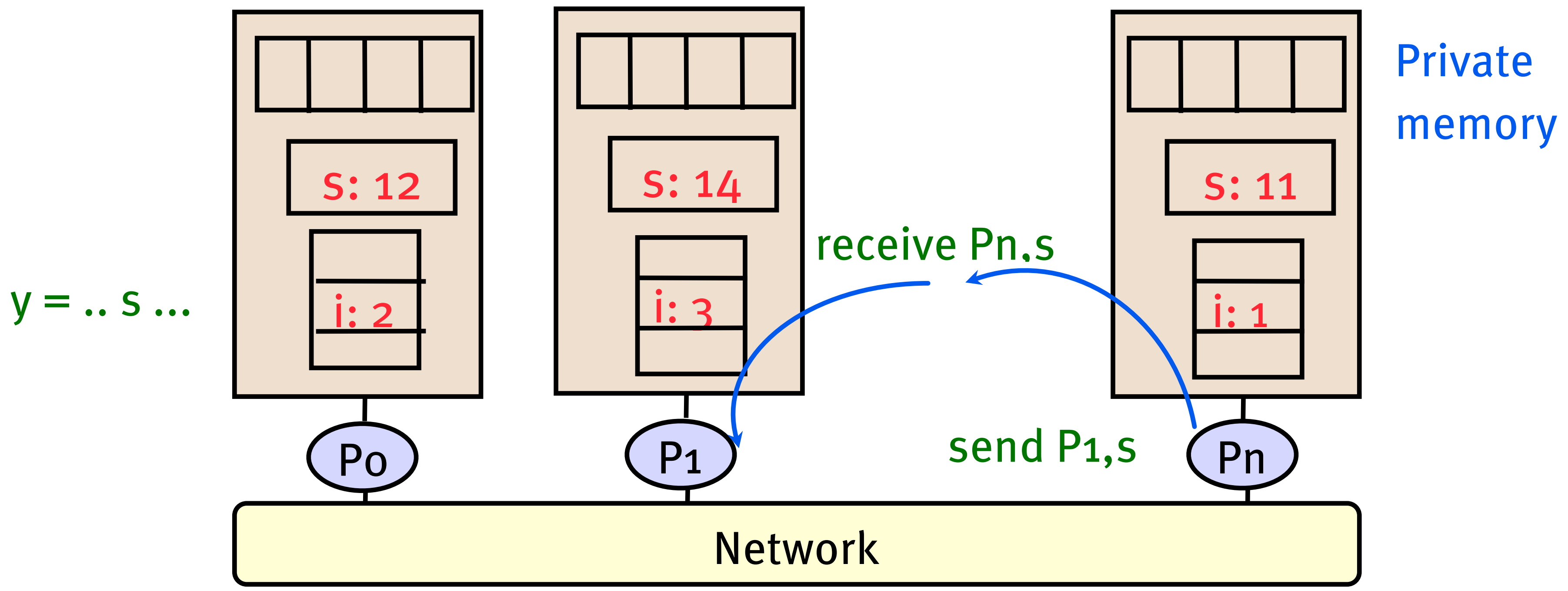
Programming Model 2: Message Passing

- 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.



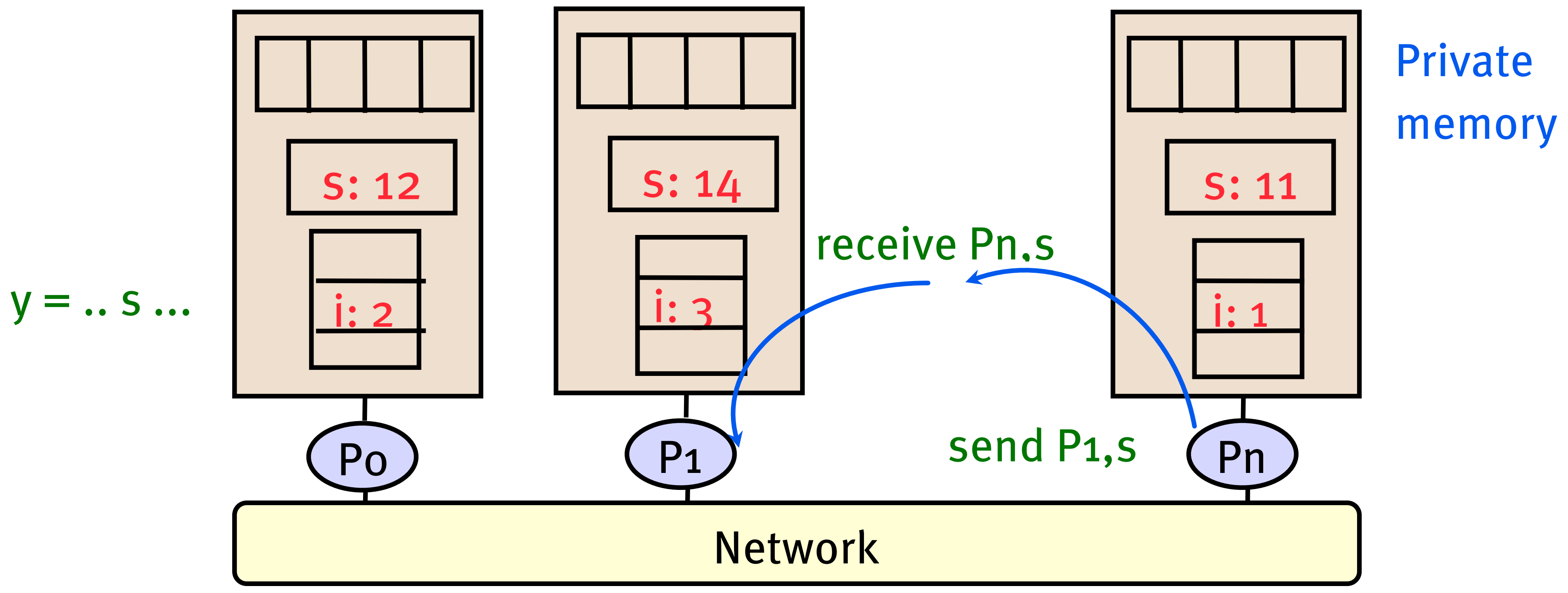
Programming Model 2: Message Passing

- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW



Programming Model 2: Message Passing

- MPI programs are structured in the following way:
  - All processes start executing identical program.
  - By calling get_my_processor_ID_number(), they can figure out who they are,
  - They then branch to run different programs.



Computing s = A[1]+A[2] on each processor

- First possible solution—what could go wrong?

```
Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

```
Processor 2
xlocal = A[2]
send xlocal, proc1
receive xremote, proc1
s = xlocal + xremote
```

- If send/receive acts like the telephone system? The post office?
- Second possible solution

```
Processor 1
xlocal = A[1]
send xlocal, proc2
receive xremote, proc2
s = xlocal + xremote
```

```
Processor 2
xlocal = A[2]
receive xremote, proc1
send xlocal, proc1
s = xlocal + xremote
```

- What if there are more than 2 processors?

MPI—the de facto standard

- MPI has become the de facto standard for parallel computing using message passing
- Pros and Cons of standards
  - MPI created finally a standard for applications development in the HPC community ⇢ portability
  - The MPI standard is a least common denominator building on mid-80s technology, so may discourage innovation
- Programming Model reflects hardware!

MPI Hello World

```
int main(int argc, char *argv[])
{
  char idstr[32];
  char buff[BUFSIZE];
  int numprocs;
  int myid;
  int i;
  MPI_Status stat;

  MPI_Init(&argc,&argv); /* all MPI programs start with MPI_Init; all 'N' processes 
exist thereafter */
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs); /* find out how big the SPMD world is */
  MPI_Comm_rank(MPI_COMM_WORLD,&myid); /* and this processes' rank is */

  /* At this point, all the programs are running equivalently, the rank is used to
     distinguish the roles of the programs in the SPMD model, with rank 0 often used
     specially... */
```

MPI Hello World

```
if(myid == 0)
{
  printf("%d: We have %d processors\n", myid, numprocs);
  for(i=1;i<numprocs;i++)
  {
    sprintf(buff, "Hello %d! Do you like Goat Simulator?\n", i);
    MPI_Send(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD);
  }
  for(i=1;i<numprocs;i++)
  {
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, i, TAG, MPI_COMM_WORLD, &stat);
    printf("%d: %s\n", myid, buff);
  }
}
```

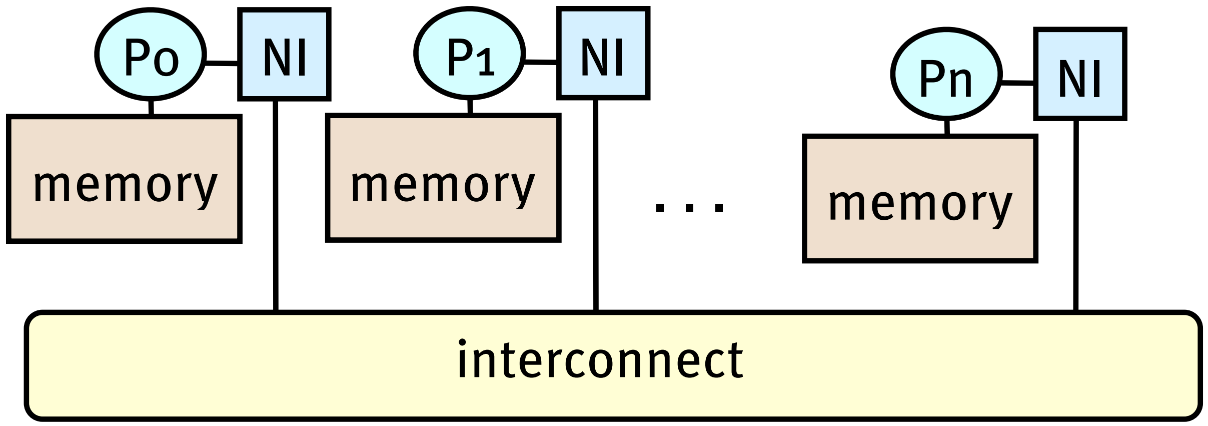
MPI Hello World

```
  else
  {
    /* receive from rank 0: */
    MPI_Recv(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD, &stat);
    sprintf(idstr, "Processor %d ", myid);
    strcat(buff, idstr);
    strcat(buff, "reporting for duty\n");
    /* send to rank 0: */
    MPI_Send(buff, BUFSIZE, MPI_CHAR, 0, TAG, MPI_COMM_WORLD);
  }

  MPI_Finalize(); /* MPI Programs end with MPI Finalize; this is a weak
synchronization point */
  return 0;
}
```

Machine Model 2a: Distributed Memory

- Cray T3E, IBM SP2
- 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.
- Each "node" has a Network Interface (NI) for all communication and synchronization.

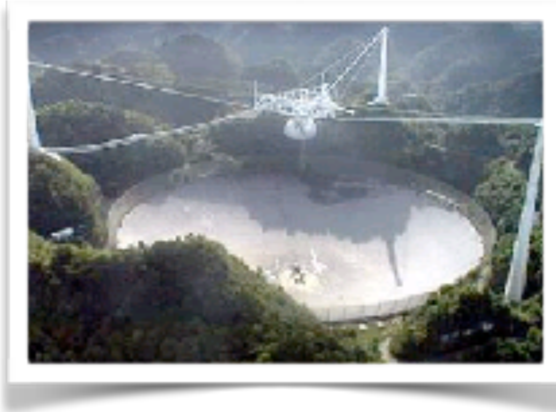


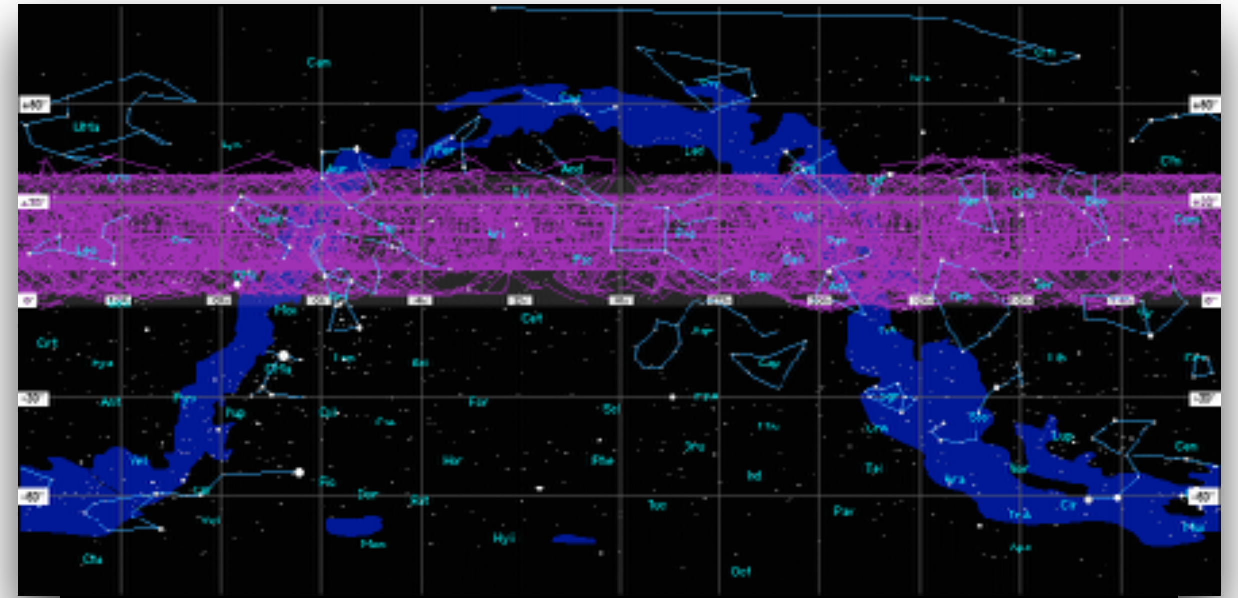
Top500 Cluster Examples

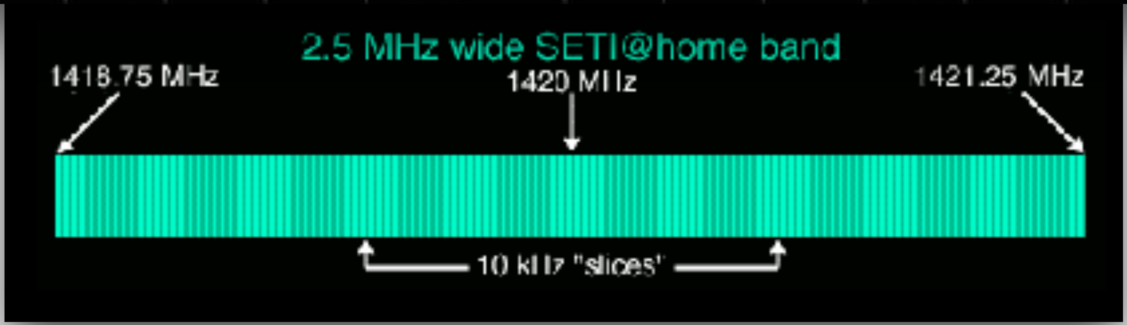
- 72% of Top 500 (Nov 2005), 2 of top 10; 82% of Top 500 (Nov 2011); 84.6% (Nov. 2013), 86.4% (Nov. 2016)
- Representative cluster: Titan (#3), at Oak Ridge National Laboratory
    - **Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x
    - 560,640 cores / 710,144 GB DRAM / built by Cray
- More:
    - #13: NASA Ames **Pleiades** - SGI ICE X, Intel Xeon E5-2670/E5-2680v2/E5-2680v3/E5-2680v4 2.6/2.8/2.5/2.4 GHz, Infiniband FDR
    - #17: UT Austin **Stampede** - PowerEdge C8220, Xeon E5-2680 8C 2.700GHz, Infiniband FDR, Intel Xeon Phi SE10P
    - #22: Petroleum Geo-Services **Abel** - Cray XC30, Xeon E5-2698v3 16C 2.3GHz, Aries interconnect
    - #334: **Amazon EC2 C3 Instance cluster** - Amazon EC2 Cluster, Intel Xeon E5-2680v2 10C 2.800GHz, 10G Ethernet

Machine Model 2b: Internet/Grid Computing

- 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

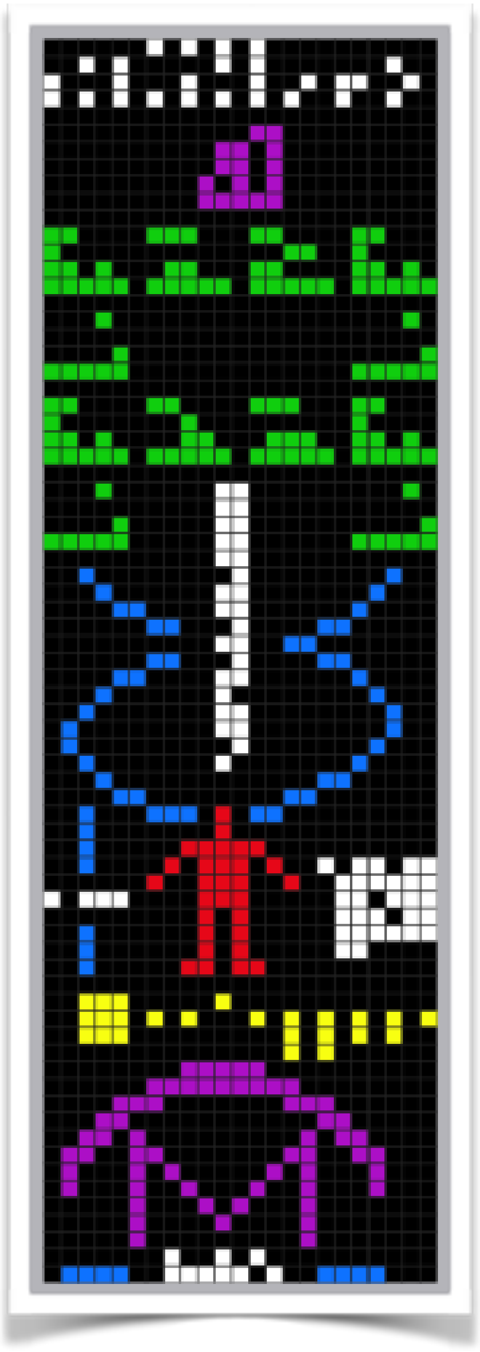






Next Step—
Allen Telescope
Array

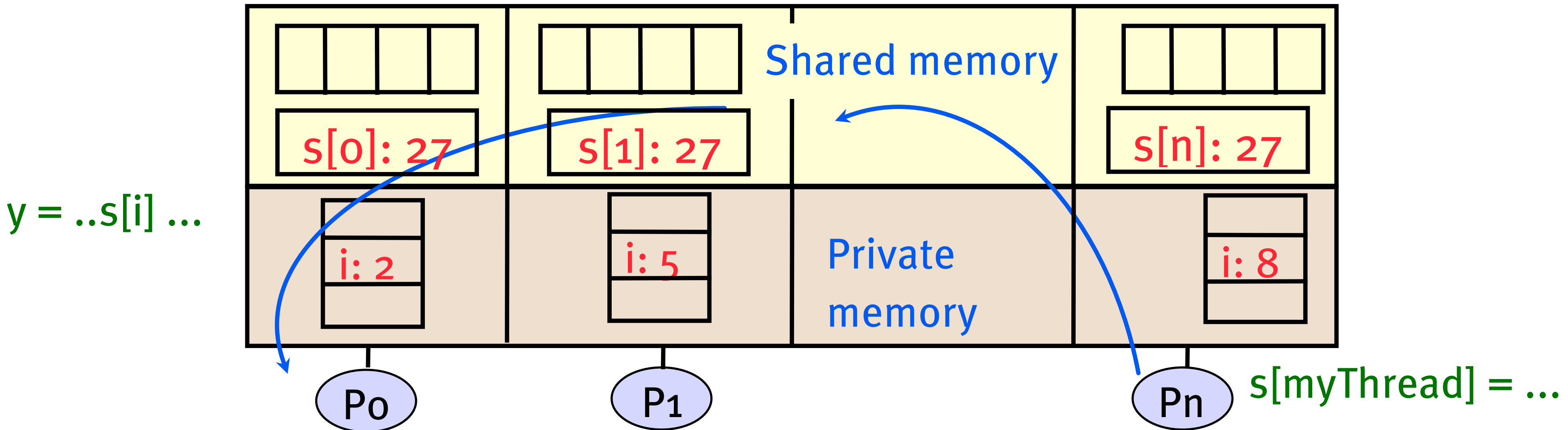
Arecibo message



http://en.wikipedia.org/wiki/Image:Arecibo_message.svg

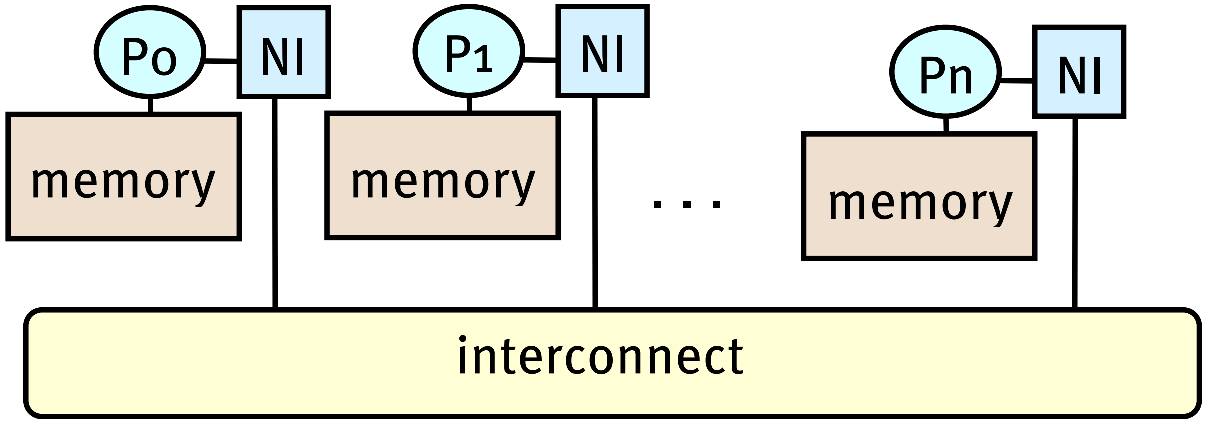
Programming Model 2c: Global Address Space

- Program consists of a collection of named threads.
  - Usually fixed at program startup time
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes
  - Cost model says remote data is expensive
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory



Machine Model 2c: Global Address Space

- Cray T3D, T3E, X1, and HP Alphaserver cluster
- Clusters built with Quadrics, Myrinet, or Infiniband
- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally

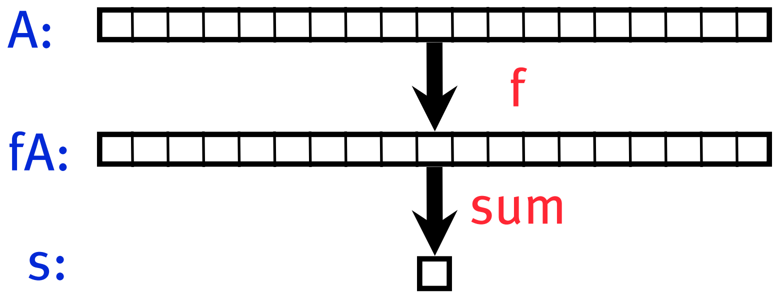


Global address space may be supported in varying degrees

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
  - Similar to Matlab language for array operations
- Drawbacks:
  - Not all problems fit this model
  - Difficult to map onto coarse-grained machines

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



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 versa
  - 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

Machine Model 4: Clusters of SMPs

- SMPs are the fastest commodity machine, so use them as a building block for a larger machine with a network
- Common names:
  - CLUMP = Cluster of SMPs
  - Hierarchical machines, constellations
- Many modern machines look like this:
  - Millennium, IBM SPs, ASCI machines
  - And it seems likely exascale machines will also look like this
  - What about accelerators?
- What is an appropriate programming model for #4?
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.