Shared Memory Architectures

Programming and Synchronization

Discuss paper on
Cosmic Cube (message passing)

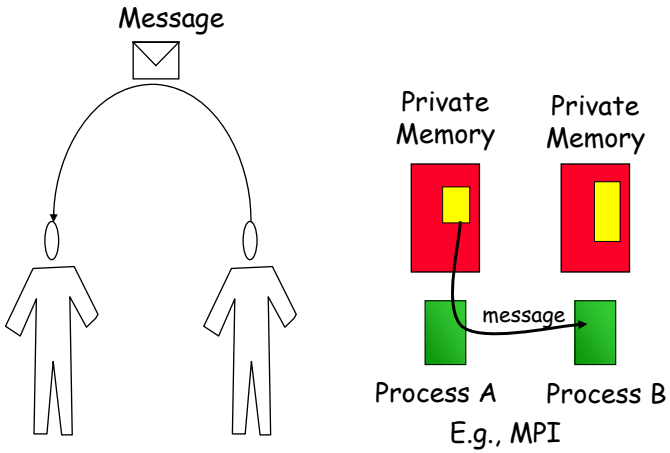
Agarwal

6.173
Fall 2010
L07

Today's Outline

- Message passing review
- Cosmic Cube discussion
  - Message passing machine
- Shared memory model
  - Communication
  - Synchronization
- Ultracomputer/RP3 discussion
  - Shared memory machine
- Shared memory programming
- Fine grain versus coarse grain parallelism
- How do caches change things
  - Improve and complicate!
  - Beehive

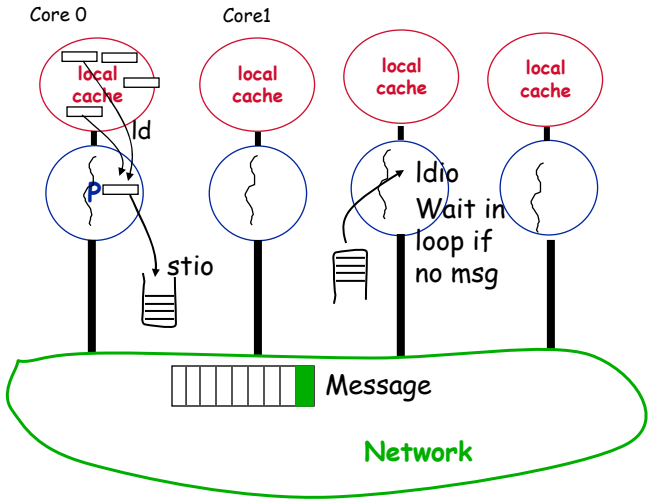
Review
Message Passing Parallel Programming Model



Communication: via messages
Synchronization: via messages

How to Receive a Message

Beehive uses polling

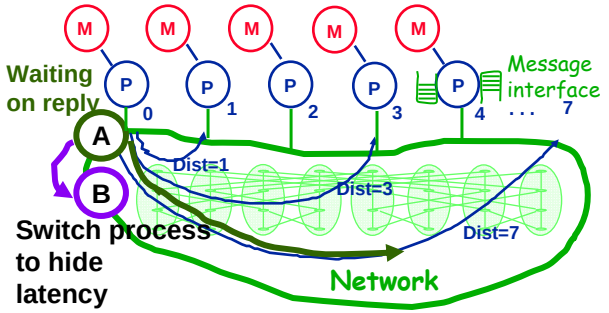


The Cosmic Cube
The Earliest Message Passing Machine

The nodes are packaged as one circuit board per node in the long card frame on the bench top. The six communication channels from each node are wired in a binary 6-cube on the backplane on the underside of the card frame. The separate units on the shelf above the long 6-cube box are the power supply and an "intermediate host" (IH) that connects through a communication channel to node 0 in the cube.

FIGURE 6. The 64-Node Cosmic Cube in Operation

The Cosmic Cube



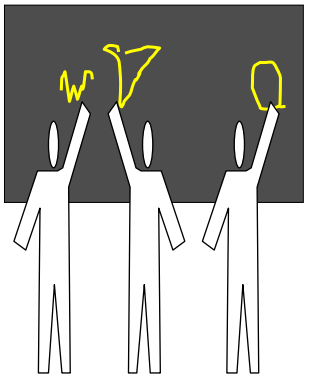
64 "nodes" (remember, multicores – on single chip – arrived circa 2000)

- Direct network – hypercube (details later in course)
- Private memories
- Message sends by calling into OS
- Routing in software
- Sequential programming on each processor & message send/receive (much like Beehive)
- Hide comm latency by switching processes
- Simple hardware

Discuss paper

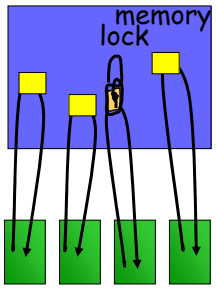
Next, Recall, Shared Memory Parallel Programming Model

Blackboard captures state



Designers

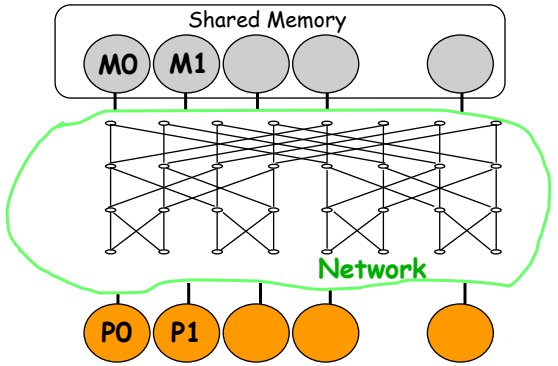
Shared memory



Threads
E.g., pthreads

Communication: via shared memory
Synchronization: shared memory locks

Ultracomputer Design

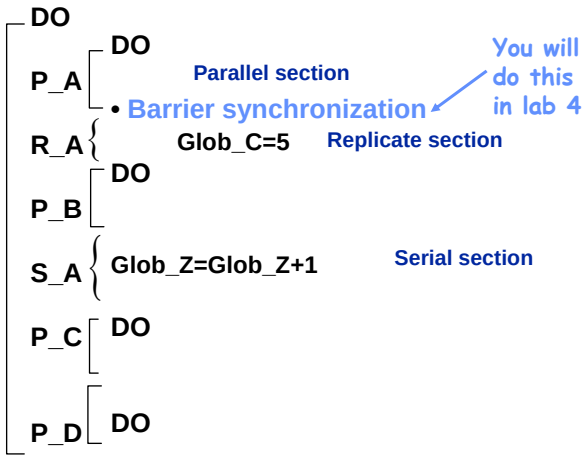


This is a butterfly network – variant of Omega network

- Indirect network – Omega network (details later in course)
- Shared memory machine
- Communication/synchronization through shared memory
- Hardware routing of memory requests
- No latency hiding – wait for memory request

Concept built as IBM RP3 machine (we will see this later)

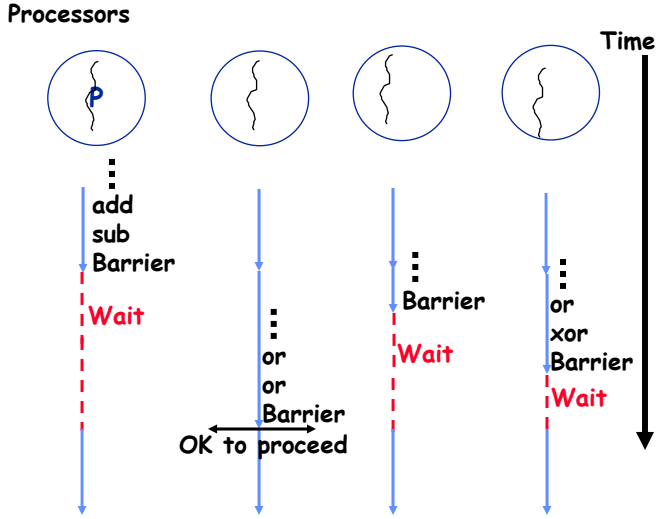
Popularized SPMD Programming (Single-program multiple-data)



Annotate sequential programs

Quick Detour

Barrier Synchronization



Barrier synchronization applies to a set of processes

A process that executes a barrier must wait until all other processes have executed their barrier

Discuss how to do barrier on Beehive using message passing

SPMD Programming Approach
Single Program Multiple Data

You should learn this!

Most parallel programs written for commodity multicores use this style (all commodity multicores happen to be shared memory machines!)*

All processors run a copy of the same program (commonly a slightly modified version of the sequential program)

Processor-specific behavior created using unique processor IDs

Also need to introduce synchronization as necessary

Let's do a simple example to build intuition

*Note that, in general, SPMD style of programming can be applied to either shared memory or message passing machines

Adding a Pair of Vectors – A Sequential Program

```
# define LENGTH 1000000

int a[LENGTH], b[LENGTH], c[LENGTH];
int i=0;

main()
{
          /* Initializations */
          . . .

          /* read in the two vectors */
          . . .

          i = 0;
          while (i < LENGTH)
          {
                    c[i] = a[i] + b[i];
                    i = i + 1;
          }


          /* output the answer */
          . . .
}
```

Sequential addition of two vectors

Parallel SPMD Version

Assume Ultracomputer model.
Assume no caches, single word memory access

```
# define LENGTH 1000000
int a[LENGTH], b[LENGTH], c[LENGTH];
int i=0;
int L=0;

main()
{
          /* create parallel processes */
          . . .

          /* Initializations */
          if (myPId == 0) . . .

          /* read in the two vectors */
          if (myPID == 0) . . .

          int myi;
          myi = getwork();
          while (i < LENGTH)
          {
                    c[myi] = a[myi] + b[myi];
                    myi = getwork();
          }

          /* output the answer */
          if (myPID == 0) . . .
}
int getwork()
{
          getlock();
          i = i + 1; /* increment is atomic */
          releaselock();
          return(i);
}
```

Assume each process runs the rest of the same program

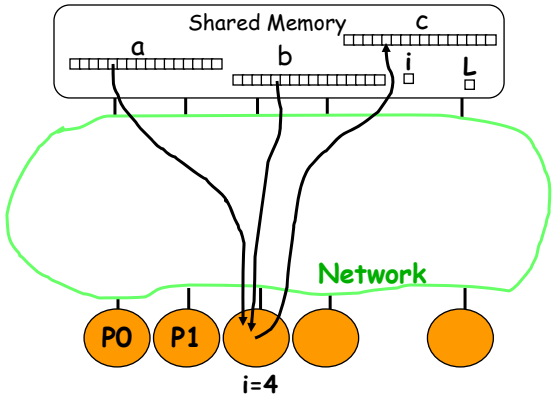
Only process 0 runs this

Get an index on which to work. Example of self scheduling

Sequential addition of two vectors

Pure Shared Memory

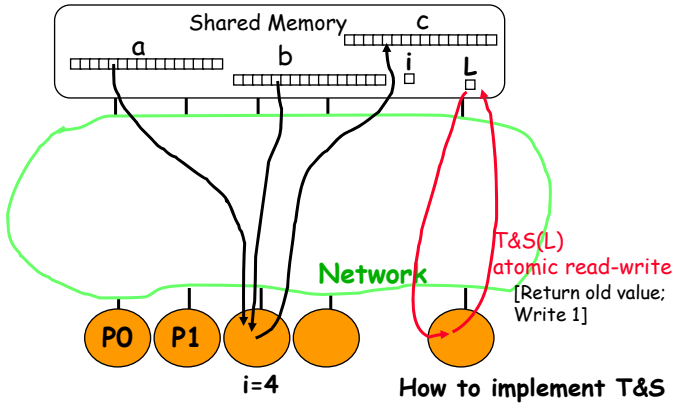
No caches, single word reads/writes



Pure Shared Memory

Lock: Using Test-and-Set Instruction

Example of a "spin lock"

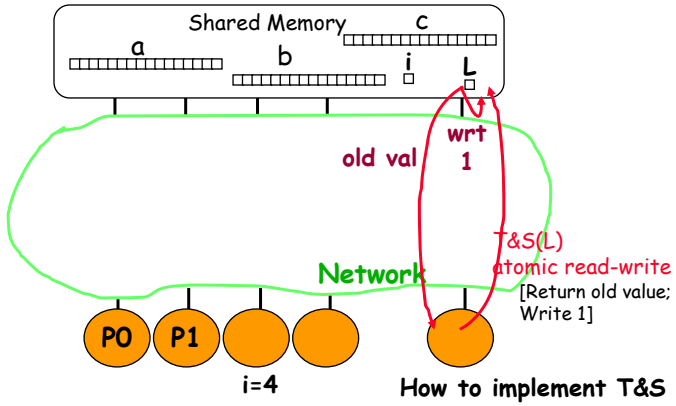


How to implement T&S in HW? In SW?

```
void getlock()
{
        while (T&S(L) == 1) {}; /* loop till you get the lock */
        return();
}


void releaselock()
{
        L = 0; /* release the lock */
        return();
}
```


Pure Shared Memory

Test-and-Set Instruction Implementation



How to implement T&S in SW? Dekker's Alg.

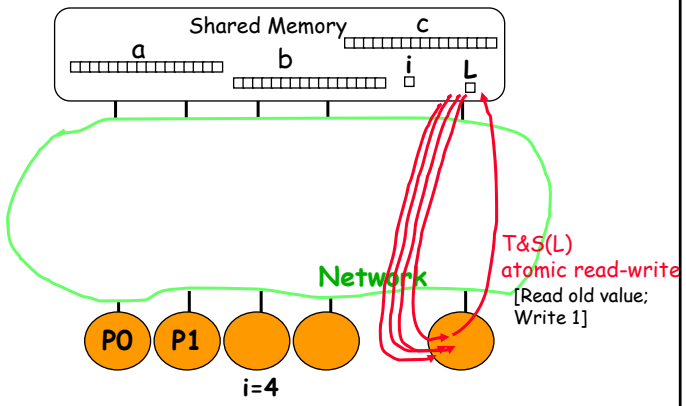
```
void getlock()
{
        while (T&S(L) == 1) {}; /* loop till you get the lock */
        return();
}


void releaselock()
{
        L = 0; /* release the lock */
        return();
}
```

Problem: Lock is held for load-store cycle!
Locks out even the lock releaser.

Can we do better? Ideas?

Pure Shared Memory

Test & Test & Set

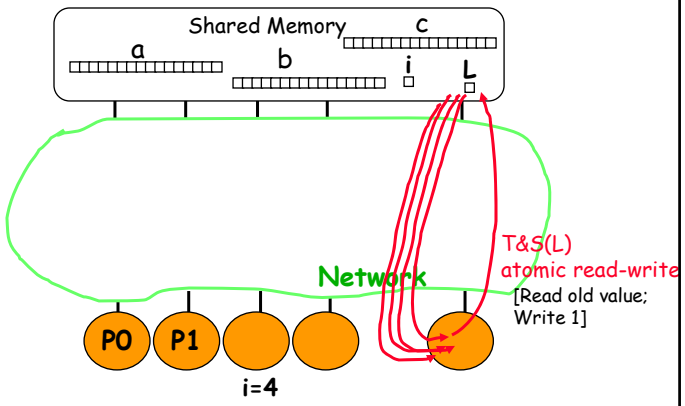



```
void getlock()
{
        while (L == 1) {};
        while (T&S(L) == 1) {}; /* loop till you get the lock */
        return();
}

void releaselock()
{
        L = 0; /* release the lock */
        return();
}
```

Any other problems?

Pure Shared Memory

Backoff concept



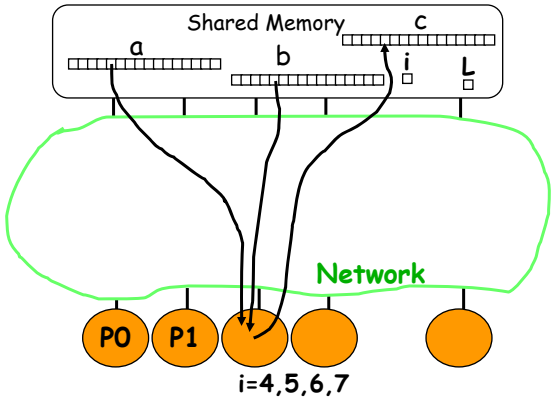

```
void getlock()
{
        while (L == 1) {}; /* introduce backoff here */
        while (T&S(L) == 1) {}; /* loop till you get the lock */
        return();
}

void releaselock()
{
        L = 0; /* release the lock */
        return();
}
```

Can do exponential backoff
Quadratic backoff
Random backoff, etc.

We engineers love to optimize!

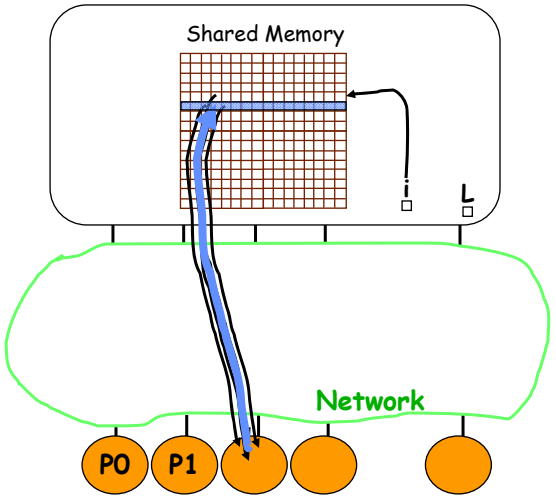
Pure Shared Memory

So, getting a work item is not so cheap after all, is it? Any ideas?



Coarse grain parallelism (versus fine grain parallelism): Get a block of 4 or 16 or more indices each time to amortize the overhead of locking

Jacobi, Same Basic Concept



Getwork() grabs an index to a row (e.g.)
Synchronization as before
Loads and stores to shared array

Finish row. How do I know when to start next jacobi iteration?

Use barrier after you finish your row

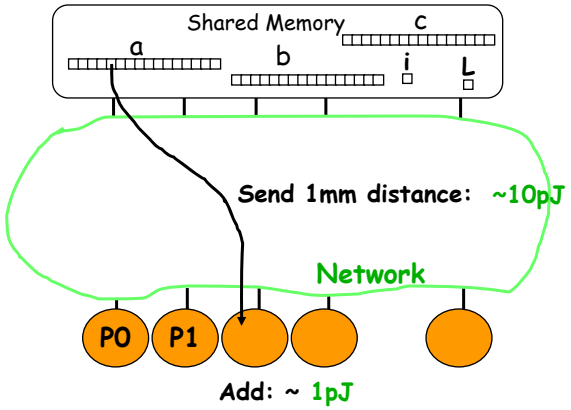
Lots of communication over the network

And very energy inefficient

Pure Shared Memory

32-bit energy costs in 40nm

DRAM read: ~1000pJ



Add: ~ 1pJ

Register read: ~1pJ

Ideas?

Caches!

Cache read (small L1): ~10pJ