Thoughts on Massively Parallel Computing

Jay Reynolds Freeman

Jay_Reynolds_Freeman@mac.com

http://web.mac.com/Jay_Reynolds_Freeman

This Talk Is On Line

- My public web page is:

  http://public.me.com/Jay_Reynolds_Freeman

- The document you want is:

  CS.159.Talk.pdf

- The talk itself is a ".key" document, hard to read on non-Macintoshes.

Who Am I?

- Used astrophysicist. (Cheap!)
- Career programmer.
- Spent much of my career dealing with massively parallel systems.
- Worked mostly below the application level, in system programming.

Preview of Talk

- Massively Multicore Machines (let's call them MMM, for short).
- SIMD Machines.
- Sun's "Phaser": SIMDish MIMD.
- Sharing Memory by Unix "mmap".
- Catch-all, depends how long it takes.

Massively Multicore Machines

- Single systems, not clusters.
- Too many cores to think about.
- Million-core systems are possible; that is a full million MIMD cores.
- Graphics cards are common, often with hundreds of (SIMD) cores.

MMM Particular Issues:

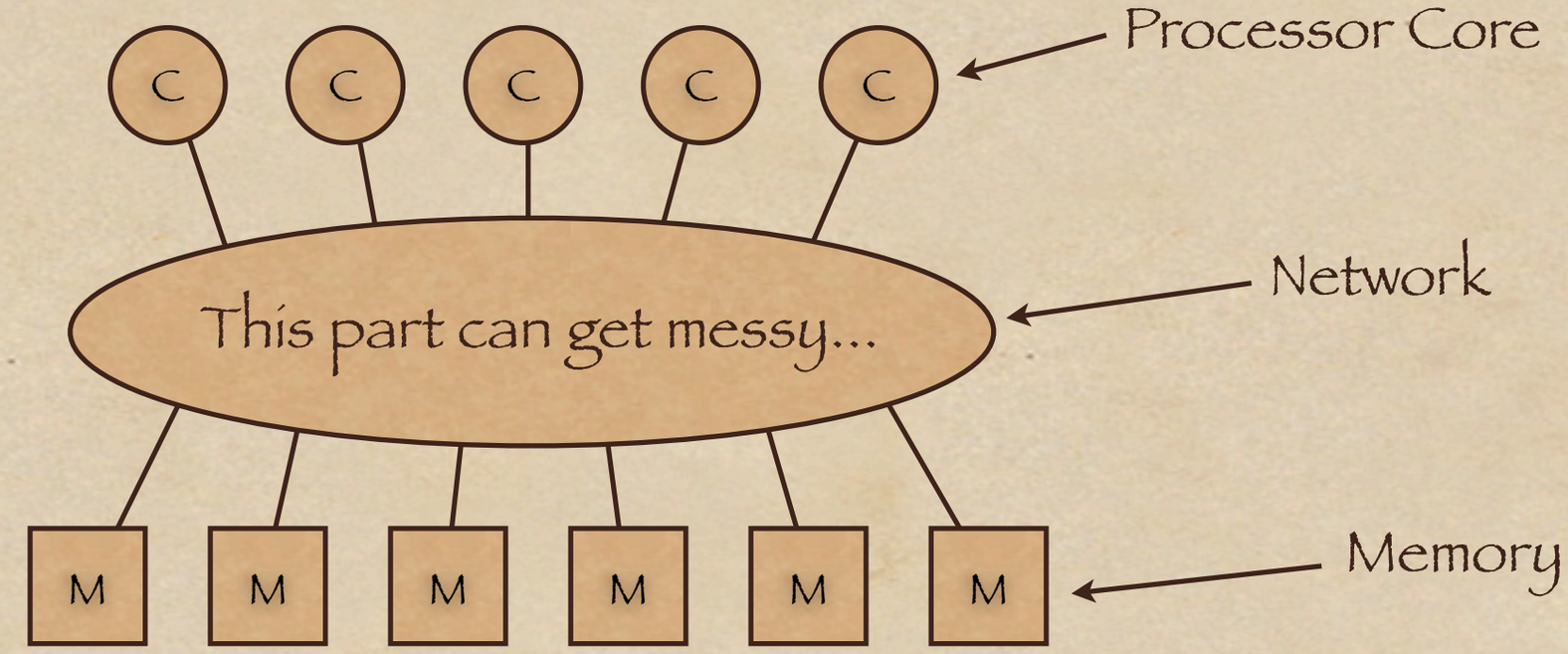
- Non-Uniform Memory Access (the abbreviation is "NUMA").
- Process/Data Positioning.
- Fault Tolerance.
- Using "extra" cores.
- I/O bottleneck.

NUMA:

- Problem: With many cores, you cannot put all memory on one bus, because everybody wants to use it.
- The bus is a limited resource!
- Memory access becomes an Amdahl's-law serial bottleneck!

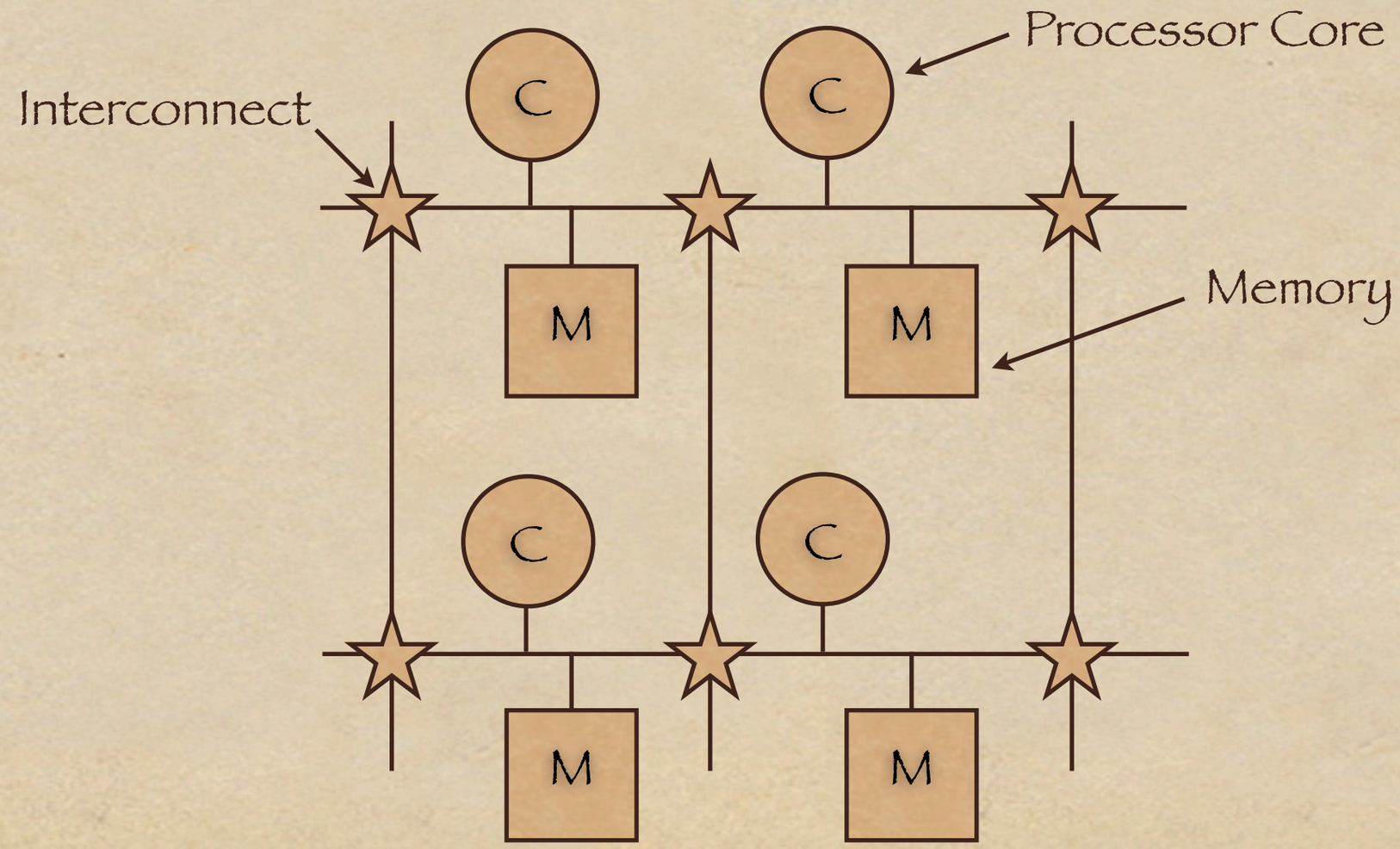
NUMA (2):

- Fix: Use a network.



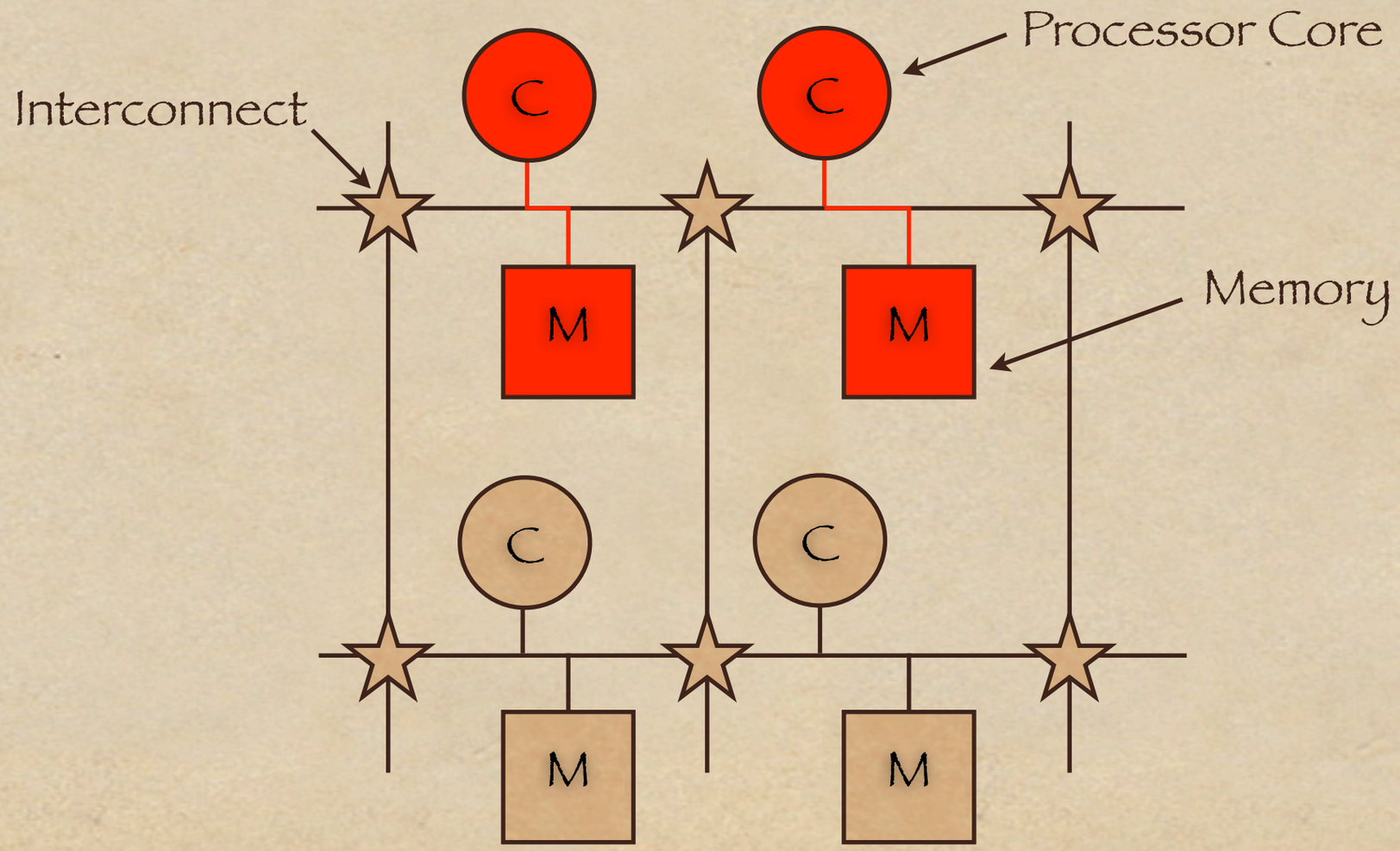
NUMA (3):

- In more detail, for example ...



NUMA (4):

- Two cores talk to local memory.

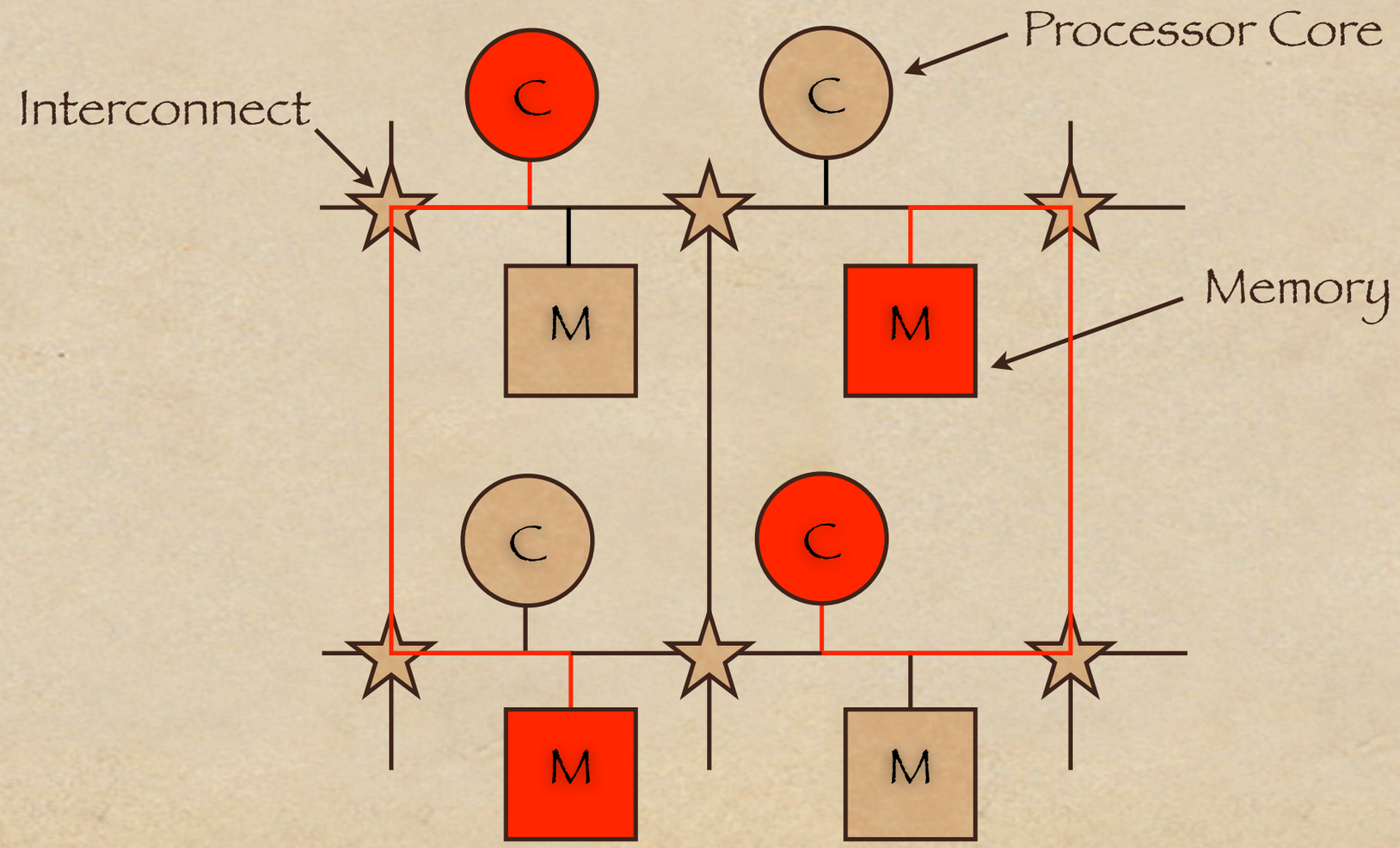


NUMA (5):

- Cache-coherence is a problem.
- A core may not see another core updating memory that the first core already has in cache.
- Therefore, the first core will not know to reload that cache line.

NUMA (6):

- Two cores talk to remote memory.



NUMA (7):

- Latency: How long for a message to get from start to destination.
- Bandwidth: How much data flows per unit time once it gets going.
- Latency and bandwidth both depend on what is talking to what.

NUMA (8):

- If data channels are like a hose...
- Latency has to do with how long the hose is.
- Bandwidth is the rate at which the water finally comes out, after it has made it to the end of the hose.

NUMA (9):

- Or like the post office: Latency is a letter's transit time, bandwidth is how many letters a day you get.
- Or like traffic on the highway: Speed limits, how many lanes, traffic jams, detours, et cetera...

NUMA (IO):

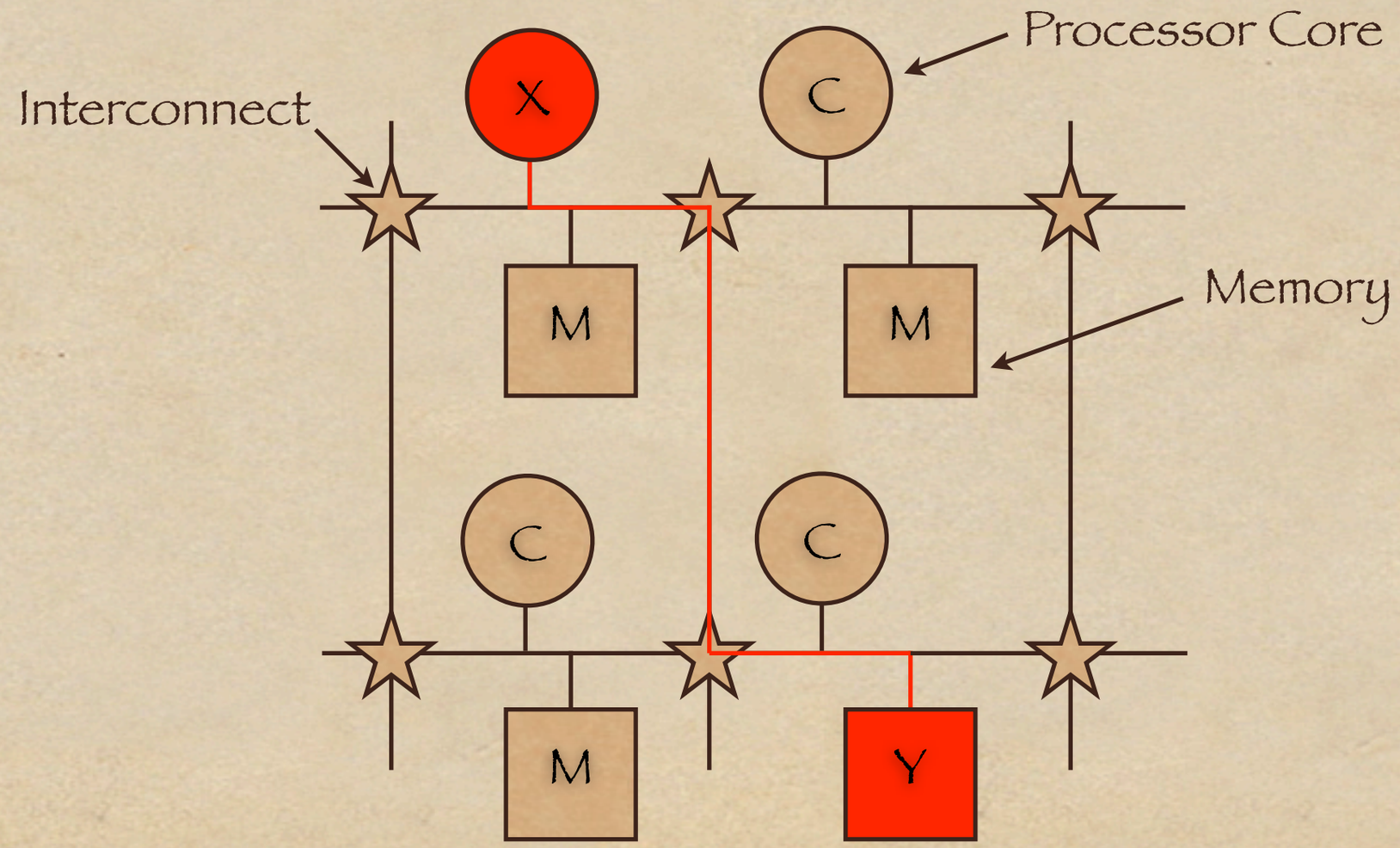
- Latency and bandwidth depend both on hardware design and on contention for use of the network.
- They are important issues for memory access, for core-to-core communication, and for I/O.

NUMA (11):

- NUMA happens even in computers with only one core.
- Latency and bandwidth vary for accessing registers, L1/L2 cache, main memory, memory swapped to disc.
- More cores make things messier.

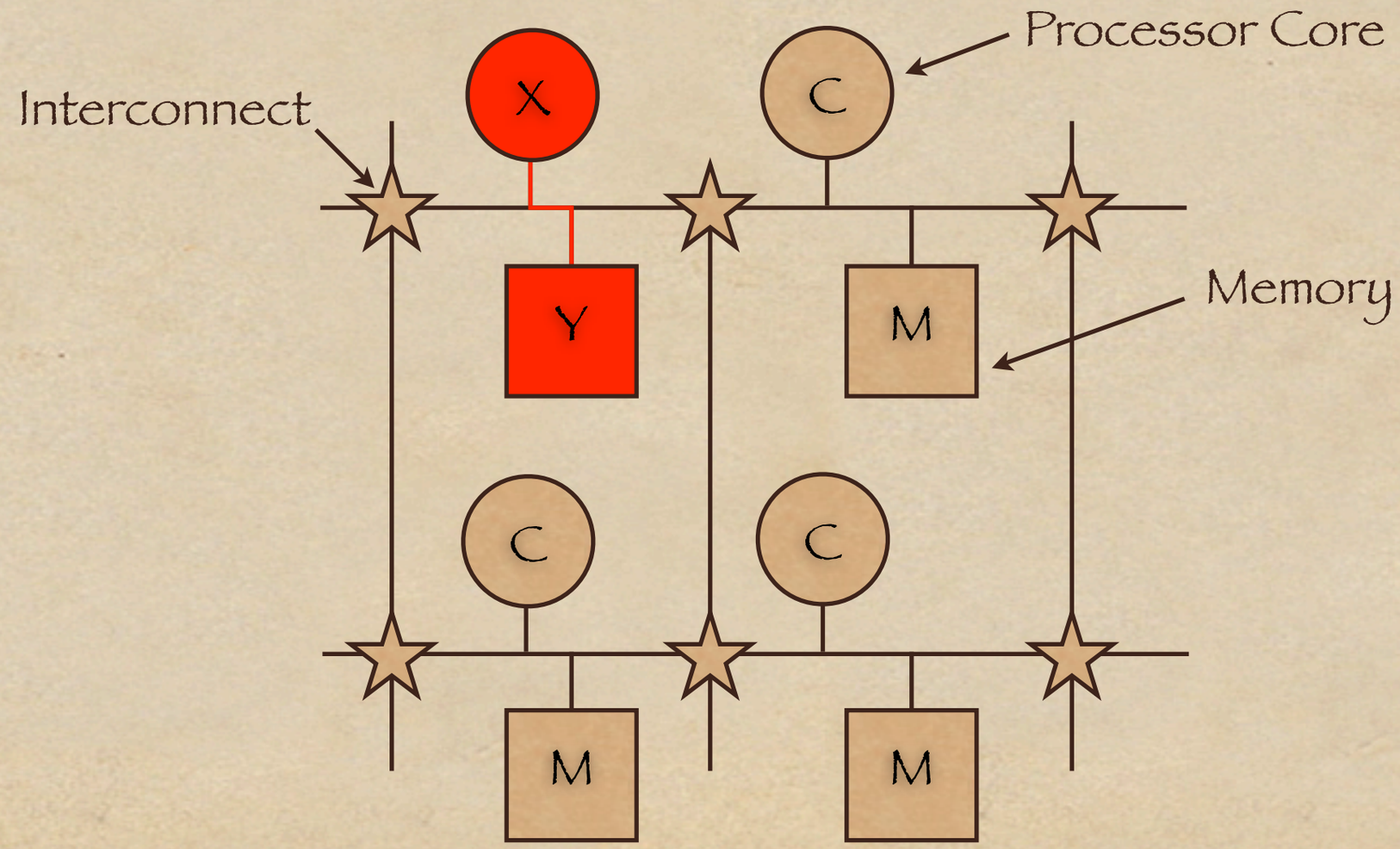
Process/Data Positioning:

- Process X uses data Y: Slow!!



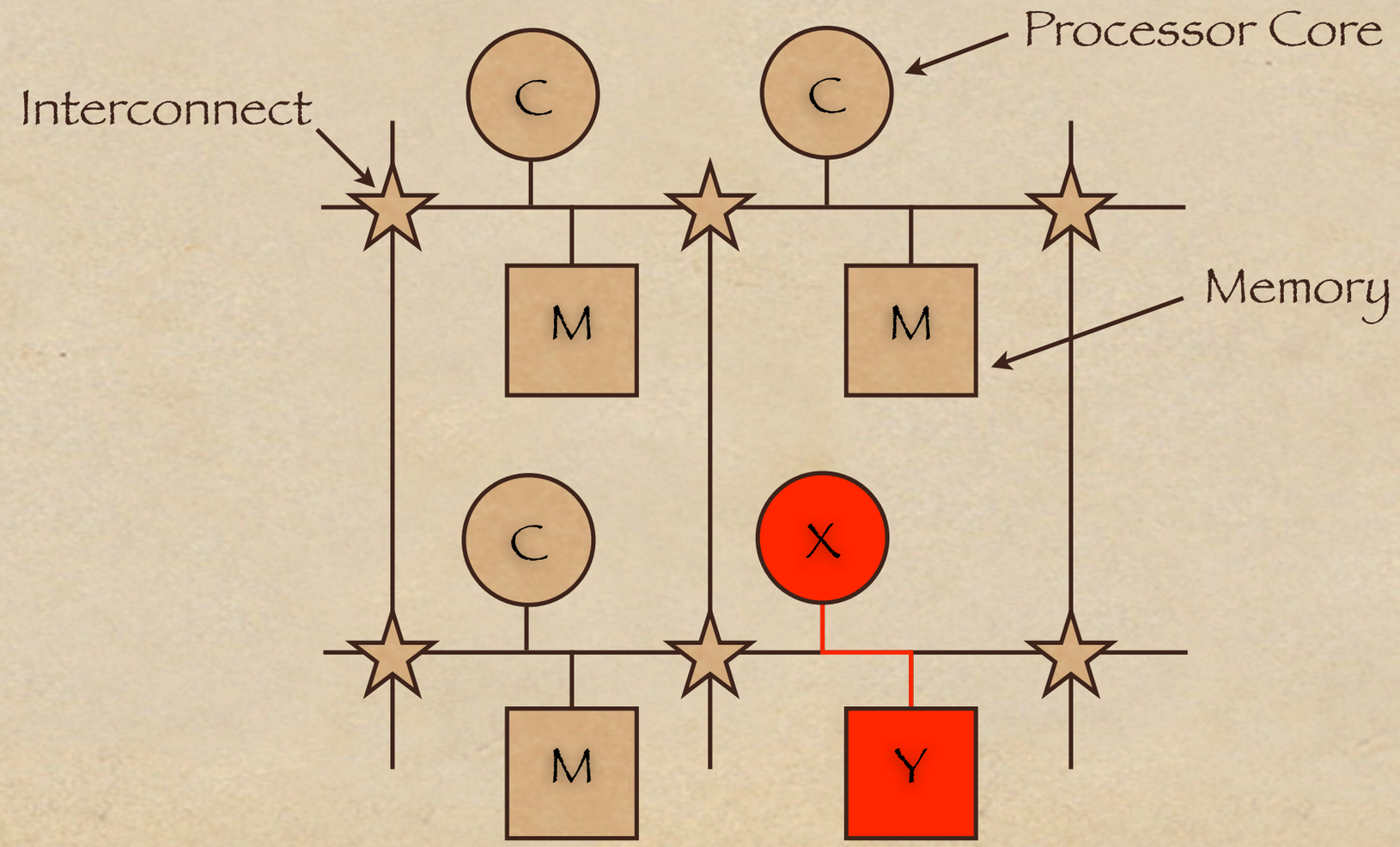
Process/Data Positioning (2):

- So either move the data ...



Process/Data Positioning (3):

- ... or move the process.



Process/Data Positioning (4):

- Static Positioning: Calculate good positions before you start.
- Dynamic Positioning: Watch data-flow, move at run-time. ("Thread Migration" and "Data Migration".)
- Or use both!

Process/Data Positioning (5):

- It takes processor time to move things around: There is a cost.
- You must decide if it is worth doing.
- When many processes each access many data items, this is a real mess.
- And it gets worse ...

Process/Data Positioning (6):

- There are N-factorial ways to distribute N threads over N cores.
- 1000000 factorial is a big number.
- So *you* cannot just *try* them all.
- A brute-force approach to positioning does not work.

Fault Tolerance:

- Big systems have lots of parts, so they fail frequently.
- What do you do if the mean time between failures (MTBF) is small, compared to the expected run time of your job?

Fault Tolerance (2):

- Big scientific calculations on a supercomputer may take months, but the MTBF may only be days.
- It would be nice to be able to predict the next ice age before it actually happens ...

Fault Tolerance (3):

- Using redundant hardware to duplicate processes and data might work, but it is expensive.
- Notwithstanding, you may need multiple copies of key operating system processes and data.

Fault Tolerance (4):

- One solution is "checkpointing".
- Every so often, write everything out to disc, and time-stamp it.
- When the system goes down, restart it using the latest copy.
- Uses lots of discs, but it works.

Using "Extra" Cores:

- You have 10000 cores. Your job does 100 units of serial work and 1000 units that are parallelizable.
- Amdahl's law permits 11 X speedup.
- 100 cores gets most of it: 10 X.
- What to do with 9900 other cores?

Using "Extra" Cores (2):

- Gustavson's Law (I paraphrase):

> **If you build a better mousetrap, nature will just build a better mouse.**

- If you provide more computer power, people will want to use it.

Using "Extra" Cores (3):

- They will say, "Wow, let's use this hot new computer to run 1000000 units of parallel work, not just 1000."
- Amdahl's law now permits speedup of 10001, and with all 10000 cores you actually get 5000.5 - not bad!

Using "Extra" Cores (4):

- Speculative computation: Enter branching calculations before you know which branch will be taken.
- Watch patterns of data use, so as to move data and processes around for more efficiency.

Using "Extra" Cores (5):

- Do checkpointing.
- Run hardware tests to pinpoint failing units before failure. (Need "hot swap" capability.)
- What other support tasks can *you* think up?

Using "Extra" Cores (6):

- It is like in the army: If you haven't got something for all the soldiers to shoot at, put the rest to work peeling potatoes or mowing grass.
- Just make sure they are doing something a little bit useful.

I/O Bottleneck:

- Lots of cores generate lots of I/O.
- So, do I/O in parallel. D'oh?
- Ultimately you probably want serial I/O of some sort. But ...
- Parallel I/O might be particularly useful for checkpointing.

SIMD Machines

- SIMD Machine Basics.
- MasPar.
- Graphics Cards.
- SIMD Languages.

SIMD Machine Basics:

- Single Instruction, Multiple Data.
- Synchronous execution: All processors run in lockstep.
- Programs are easy to understand.
- Bugs are repeatable, because the timing is the same in each run.

MasPar:

- MasPar Computer Corporation.
- Late 1980s to early 1990s.
- Big SIMD machines: 16 K cores.
- At the time, possibly the world's most powerful computers.
- 10**12 logic operations per second.

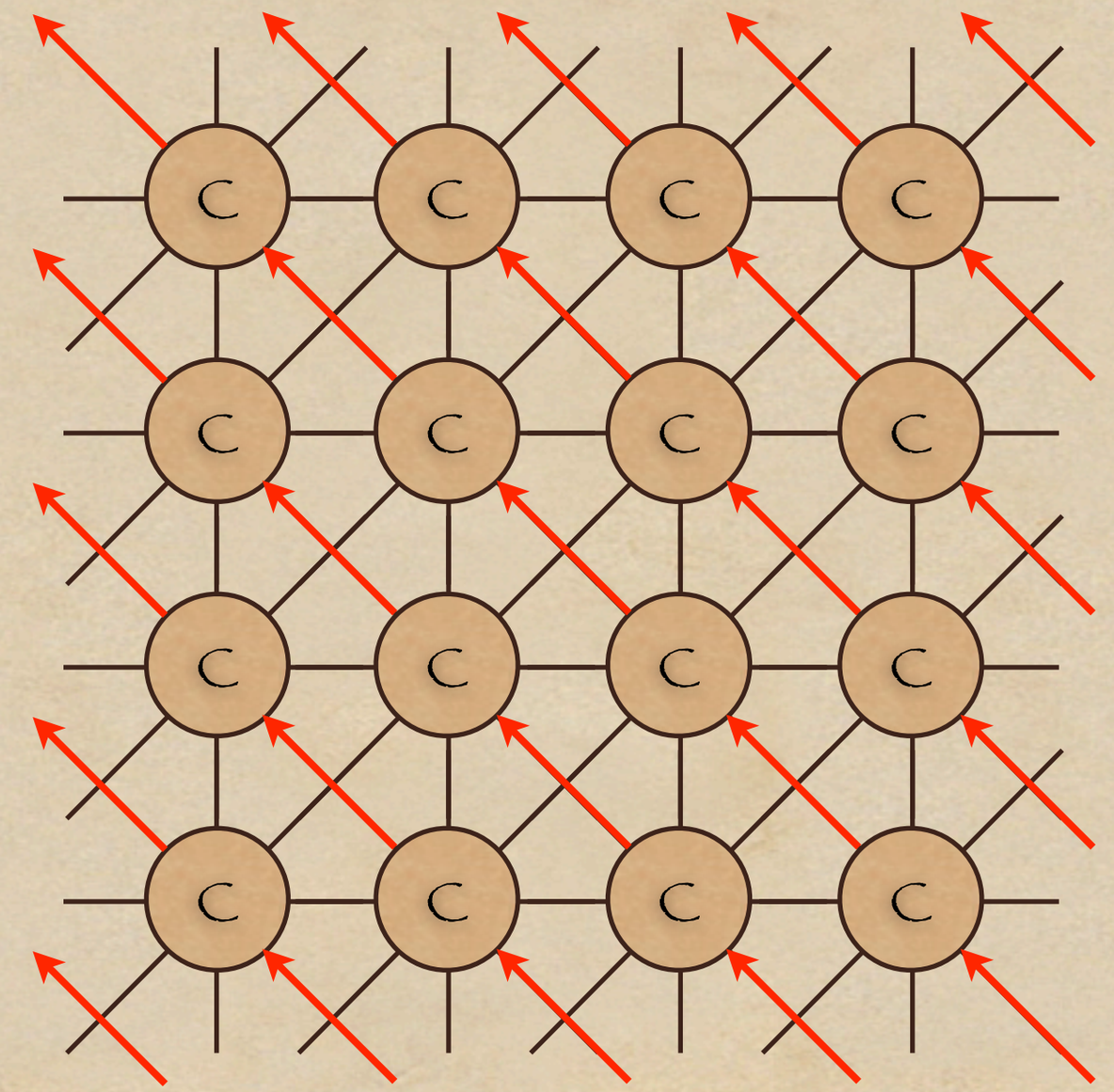
MasPar (2):

- 2D array of cores, as coprocessor to a host, just like a graphics card.
- Host could "broadcast" (send to all SIMD cores).
- Host could "global-or" (get overall or of a particular bit in each core).

MasPar (3):

- SIMD array had 8-way nearest-neighbor communication (with toroidal wrap-around).
- All cores could simultaneously send data to the core to (e. g.) their immediate northwest.

MasPar (4):



Note: Cores have local memory, not shown for brevity.

MasPar (5):

- There is a core-to-core communication network that let distant pairs of cores send data to each other, but not all at once.
- What's this about "not all at once"? That doesn't sound like SIMD ...

MasPar (6):

- Sneaky trick for non-SIMD things:
- Most SIMD instructions are contingent on a local execution bit, the "e-bit": If it is not set in a given core, then most of the instructions executed there do nothing.

MasPar (7):

- So consider this code, that you want to run on the SIMD array:

```
if( test() )
    consequent();
else
    alternative();
```

- The compiler does the following ...

MasPar (8):

(1) All cores run the test, and save the result.

(2) By moving the result to the e-bit, all cores where the test failed turn themselves off.

(3) All cores run code for the consequent. (Turned-off cores of course do nothing.)

(4) All cores turn themselves on (turn the e-bit on). This is an unconditional command.

(5) Cores where the test passed turn themselves off.

(6) All cores run code for the alternative. (Turned-off cores of course do nothing.)

(7) All cores turn themselves on.

MasPar (9):

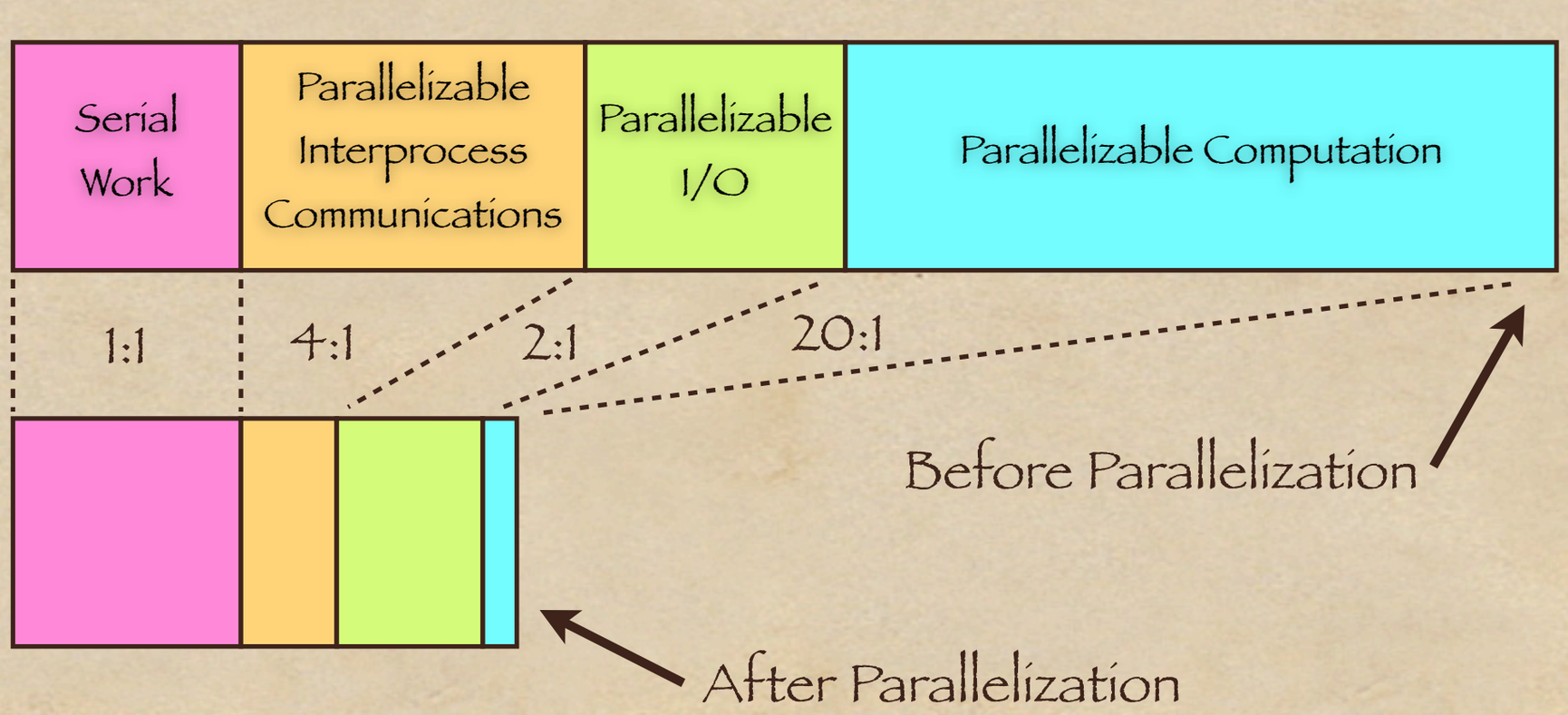
- When you use the e-bit this way:
- You spend more time running code:
  We just had to run code for both consequent and alternative.
- But you get to handle branches in a SIMD machine.

MasPar (10):

- By using the e-bit:
- Distant cores can take turns using the network to send data.
- The host can communicate with individual cores.
- Loops and so on handled similarly.

MasPar (11):

- Amdahl's Law generalizes to many kinds of speed-up combined:



Graphics Cards:

- I am no expert, this may be wrong!
- Graphics cards are pretty good small SIMD arrays: 100s of cores.
- But:
  - No broadcast or global-or!
  - No interprocess communication!

Graphics Cards (2):

- In graphics cards, all out-of-core communication is done on the host.
- You move graphics memory to the host, push it around, and then send it back to the graphics card.
- Therefore ...

Graphics Cards (3):

- In graphics cards:

Communications that go outside the graphics cores are an Amdahl's law bottleneck!

- Now, that is a problem.
- But ...

Graphics Cards (4):

- It might change, because:
- Nearest-neighbor communication doesn't take a lot of silicon.
- GPU manufacturers are trying to enter the supercomputer market.
- Again, I could be wrong.

Graphics Cards (5):

- MasPar failed: We had a small market and so could not afford the best (fastest) silicon technology.
- GPU manufacturers have a huge market, and may be able to add general features and retain speed.

SIMD Languages:

- Good languages make it easy to think about problems.
- Here is how MasPar did it.
- Suppose we want to know if any cores have distance from the (0, 0) corner of the array, equal to 17.

SIMD Languages (2):

```
main() {

    plural int myDistanceIsRight = FALSE; // "plural":  One on each core.
    plural double hypotenuse = 0.0;
    plural int sumSquareSides = 0;

    single int answer = FALSE;            // “single”:  Only on the host.
    single int desiredValue = 17;

// We arranged that each core knew its position in the array.

sumSquareSides = arrayX*arrayX + arrayY*arrayY; // On the SIMD array.
hypotenuse = sqrt( (double)sumSquareSides );    // On the SIMD array.
if( hypotenuse == desiredValue )                // Broadcast.
    myDistanceIsRight = TRUE;                   // On the SIMD array

answer = myDistanceIsRight;           // Implicit cast to single
                                      //   is an automatic global-or.
if( answer )                          // On the host.
    printf("Happy happy joy joy!\n"); // Print on the host.
else
    printf("WAAAHHHH!\n");            // Print on the host.
}
```

Sun's "Phaser"

- Sun Microsystems, Circa 2000.
- Up to 56 K cores in one machine.
- Sort of a SIMD/MIMD hybrid.
- Intended for circuit simulation, but many problems fit that model.
- Messy communication grid.

SIMD/MIMD Hybrid:

- Each core runs a short code loop.
- Separate code for each core.
- At the end of each pass through the loop, all cores synchronize.
- Each loop pass generates data to send to other cores for next pass.

Circuit Simulation:

- Each core models a few parts.
- Each loop pass calculates what those parts do in one clock cycle.
- Data sent are signals sent down wires of the circuit, for use by other parts in the next clock.

Other Problems Like That:

- Weather/climate modeling.
- Neural networks.
- "Relaxation" techniques in math, physics and engineering.
- Physical models with local effects.

Messy Communication Grid:

- Time to send data varied from 1 to ~60 clocks (with no contention).
- Message routing was determined at compile-time; that took some work.
- Latency and bandwidth were both occasionally a problem.

Unix "mmap"

- Reading files from disc is slooow ...
- Unix has a function, "mmap", to load an entire file into memory.
- The OS can still swap out parts of the file to disc, when necessary.
- But the bias is, keep it swapped in.

Reading Mmapped Files:

- Mmap a file to char * address "foo".
- To get an int at offset 1000000 into the file (for example), just do this:

```
myInt = *(int *)(foo + 1000000);
```

- More likely, treat the file image as a big array or a big struct.

Processes May Share a File:

- More than one process may mmap the same file, and share access to the place where it is loaded.
- Processes still have to worry about critical sections, races, and so on.
- Useful for - say - a big database.

Now Do It Backward:

- One process or thread creates a dummy file and fills it full of zeros.
- Many processes or threads mmap that same dummy file.
- They now have a piece of shared memory, for anything they like!

Mmap is Poorly Documented:

- Use the Unix "man" pages.
- Look for on-line tutorials.
- I have no idea how Windows does it.
- "mmap" is not "shmget"; the latter is for memory-mapped hardware.

Catch-All

- The "volatile" keyword.
- Parallel is Not Concurrent.
- Randomness Is Your Friend.

The "volatile" Keyword:

- A good compiler will optimize this loop away to nothing:

```
x = 3;
for( int i = 0; i < 1000000; ++i)
    if( x == 24 )
        doSomething();
```

- But what if x is changed in parallel by some other thread or process?

The "volatile" Keyword (2):

- "volatile" says, "read this every time I tell you to do so":

```
x = 3;
for( int i = 0; i < 1000000; ++i)
    if( ((volatile)x) == 24 )
        doSomething();
```

- Not all implementations work the same: Use caution; read manuals.

The "volatile" Keyword (3):

- DANGER:

```
extern volatile int x;
myFunction( x, x + 3 );  // Two separate reads!!
```

- Another process may change x between the two reads. Oops!

- Better:

```
extern volatile int x;
int y = x;                       // Just one read.
myFunction( y, y + 3 );
```

Parallel is Not Concurrent:

- A "memory barrier" flushes caches, and lets all writes finish up.
- Everything you had been writing will make it out to RAM before the end of a memory barrier operation.
- See "man -k barrier" in Unix.

Parallel is Not Concurrent (2):

- You often need memory barriers for process synchronization. Forgetting one is a serious bug.
- So, suppose you forgot one, and now you test your buggy code on hardware that is only concurrent.

Parallel is Not Concurrent (3):

- Process switch flushes much cache.
- The scheduler may do a memory barrier itself, as a precaution.
- So you may not see your bug till the code runs on real parallel hardware.
- By that time it is way too late!

Parallel is Not Concurrent (4):

- There is a general principle here:

If you are trying to shake stubborn creepy-crawlies out of a big tree, you need a big hammer to hit it with.

- So, use real parallel hardware to test real parallel code.

Randomness Is Your Friend:

- Simple way to share a resource fairly: When you fail to get it, wait for a random time, then try again.
- Ethernet does this; study how.
- The scheduler may be random enough; if so, just sleep a while.

Thanks for listening.

(Have a cookie.)

Jay_Reynolds_Freeman@mac.com

http://web.mac.com/Jay_Reynolds_Freeman