The Parallel Runtime

Though parallel computers run Linux kernels and though compilation is largely routine, there are a few aspects of parallel computers run-time of interest. Communication will remain our main focus.

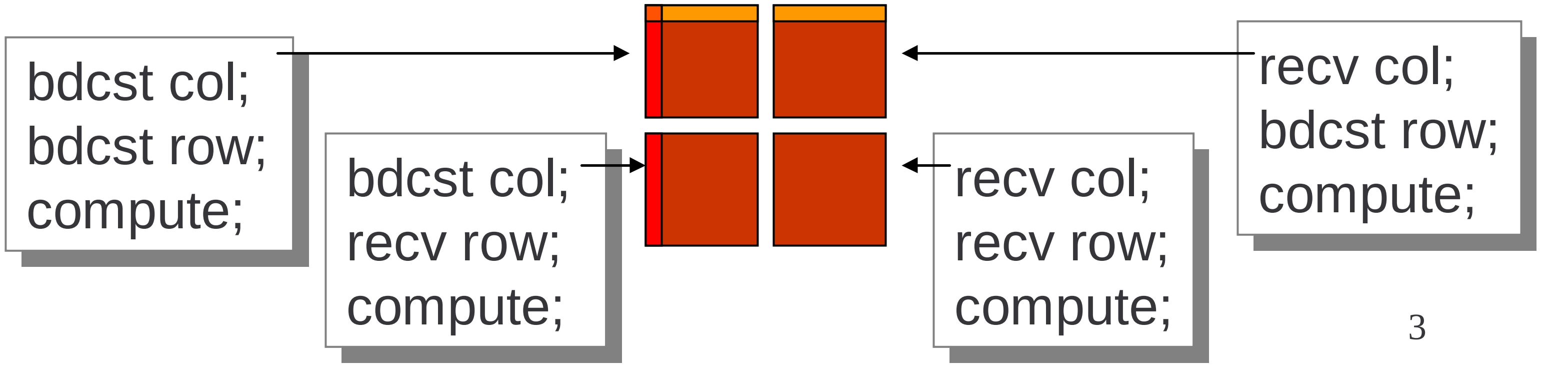
Project Posted

- Due 14 March 2005 at 12:00 Noon PST
- Specification are on the Web
- “Personally interesting” project is preferred
    - Check in with an email outline of plan
- Turn in one sheet in each future lecture of recent progress on project (pass tonight)
- Questions?

Compiling Parallel Programs

- Languages use a “single program, multiple data” (SPMD) view ⇒ the compiler produces 1 program
- Logically, ZPL executes 1 statement at a time, but processors go at their own rates using “data synchronization” -- when data’s present, keep going

```
 for i := 1 to n do
   [1..m,*]  Col := >>[ ,k] A; -- Flood kth col of A
   [*,1..p]  Row := >>[k, ] B; -- Flood kth row of B
[1..m,1..p]    C += Col*Row;   -- Combine elements
 end;
```



All Part Of One Code

The SPMD program form requires that both 'sides' of the communication are coded together if/when 2-sided comm is used

```
if my_col(k) then bdcast(A[mylo1..myhi1,k])
             else recv(Col[mylo1..myhi1. * ]);
if my_row(k) then bdcast(B[k,mylo2..myhi2])
             else recv(Row[ * ,mylo2..myhi2]);
```

The actual form of ZPL's communication is given below

Compiling ZPL Programs

- Because ZPL is high level, most optimizations have a huge payoff
- Examples of important optimizations

```
rightedge  := max<< Pts.x;--Find bounding box
topedge    := max<< Pts.y;--  |
leftedge   := min<< Pts.x;--  |
bottomedge := min<< Pts.y;--  V
```

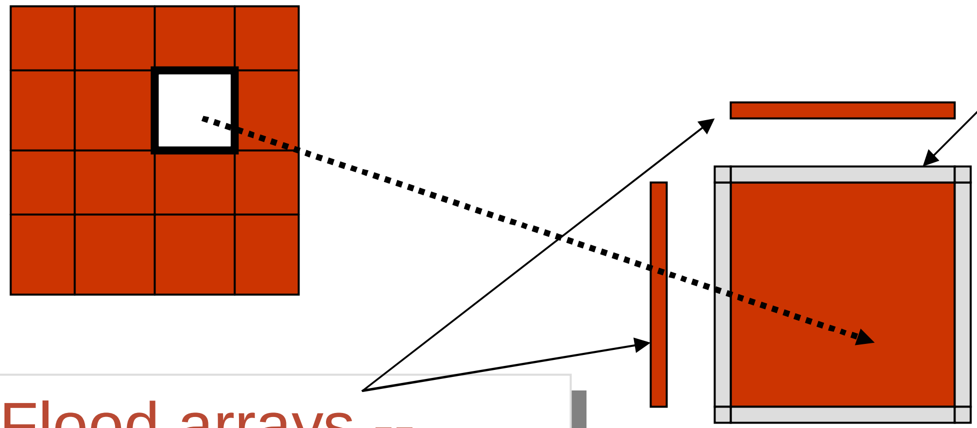
converts to 1 Ladner/Fischer tree on 4-part data; the last 3 communications are “free” [Derrick Weathersby]

```
Above := A@N + B@N + C@N;
```

combines all communication to north (and south) neighbors

What Happens When A Program Runs

- One processor starts, gets the logical arrangement from command line, sends it to others and they start
  - This differs slightly from machine to machine
- Each processor computes which region it owns
- Each processor sets up its scalars, routing tables and data arrays ...



Fluff -- the temporary storage used to hold values transmitted for @-communication -- it is inline to make indexing transparent

Flood arrays -- minimum allocation

The “Problem with Parallel Computers”

Parallel computers have (at least) one of three communication mechanisms …

- Hardware shared memory
- Message passing library
- One-sided communication (shmem)

- Any compiler seeking portability needs to target all three
- The standard solution is to generate message passing code because Shared and Shmem can both implement message passing

Unfortunately … this lowest common denominator solution doesn’t exploit the hardware

Message Passing

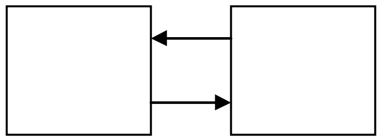
- Message passing is provided by a machine-specific library, but there are standard APIs
  - – MPI -- Message passing interface
  - – PVM -- Parallel Virtual Machine
- Example operations
    - Blocking send … send msg, proceed after it’s received
    - Non-blocking send … send msg, continue execution
    - Wait_for_ACK … wait for ack of non-blocking send
    - Receive … get msg that has arrived, or wait for it
- Message passing is not for compilers; it’s for programmers to use with C or Fortran,

A Compiler's Problem w/ Message Passing

Programmers may use MP deftly, but compilers ?

- Overlapping Communication/Computation
  - When exchanging data with neighbor: do it & wait

```
send_to(p+1, my_data1);
recv_from(p+1, his_data1);
```



  - Programmers can know data not immediately needed

```
nb_send_to(p+1, my_data1);
compute(a, boodle);
recv_from(p+1, his_data1);
wait_for_ACK(p+1, my_data1);
my_data1 = his_data1 + some + other + values;
```

  - If a compiler fails at analysis, it must be conservative, and use blocking sends

Look Under the Covers

To see the sensitivity of high-level operations to the computer's communication mechanism…

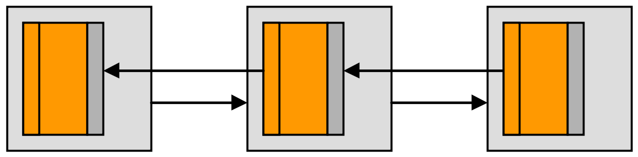
- Consider compiling code for

  ```
  [1..n, 1..n-1] B := A@east;
  ```

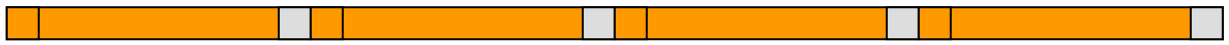
- For three different communications approaches
  - Message Passing
  - Shared Memory
  - One-sided communication

Under the Covers: Message Passing

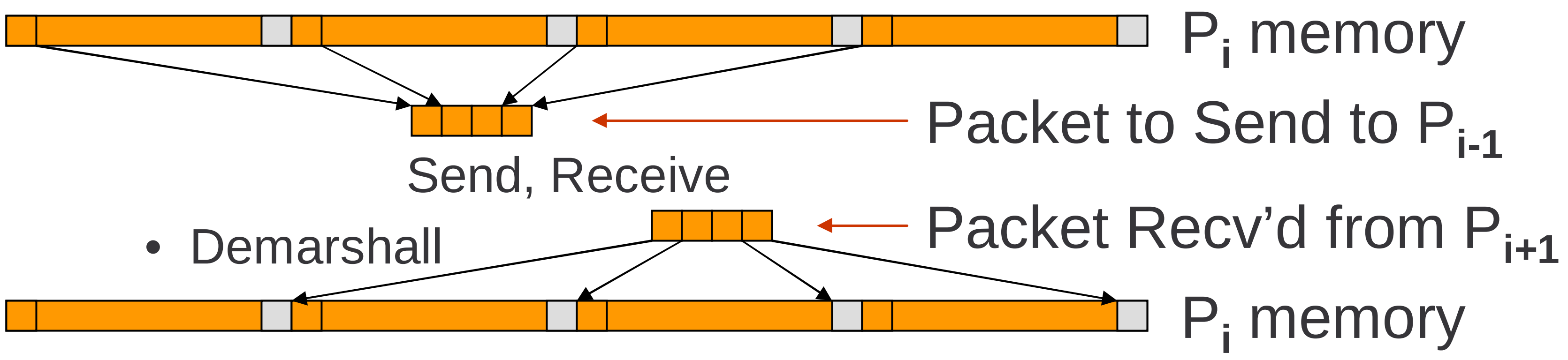
```
B := A@east ...;
```



- Move edge elements of A, then local copy to B



- Message passing …
  - Marshall the elements into a message
  - Demarshall



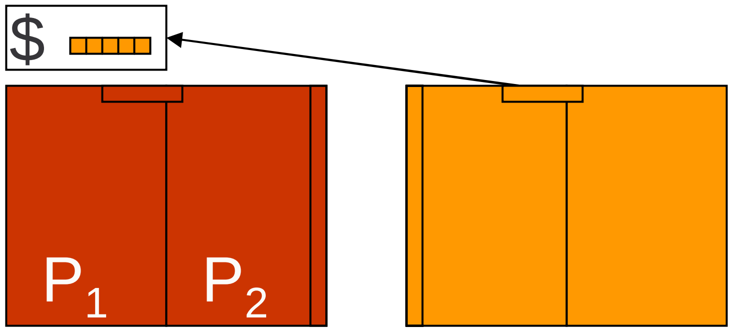
Under the Covers: Shared Memory

One coherent global memory manipulated through “standard” load and store operations (more on this later) with caching

```
B := A@east ...;
```

- In the model each processor writes directly into the portion of B that it ‘owns,’ referencing elements of A
- No explicit ‘fluff’ regions, but synch needed

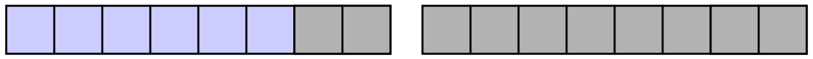
```
barrier_synch();
for (i=mylo1_B;i<myhi1_B;i++){
 for (j=mylo2_B;j<myhi2_B;j++){
        B[i][j]=A[i][j+1];
 }
}
```



Barrier ensures all processors are done w/A, ready for B

Consequences of No Fluff

- A processor's "responsibility" can straddle a cache line



- When reading, get more data than needed, missing benefits of caching
- False sharing -- writes to cache line will invalidate lines in other processors even though there is *no actual collision on words*
- Start on cache line boundaries or use fluff

Alternative Communication (Shmem)

- Message passing is “heavy weight” because it requires both a send and acknowledgement
- A “lighter weight” approach is *one-sided communication*, also known as *shmem*
- Two operations are supported --

  get(P.loc,mine); -- read directly from *loc* of proc. *P* into *mine*

  put(mine,P.loc,); -- store *mine* directly into *loc* of proc *P*
- Not shared memory since there is no memory coherence -- the programmer is responsible for keeping the memory sensible

Under the Covers: Shmem

- To implement "locally coherent" memory, programmers use post() and wait() to set and wait on data present, memory available events

```
B := A@east … ;
   post(P-1, my_data_ready); -- say mine's available
   wait(P+1,his_data_ready); -- wait til neighbor's ready
   for (j=mylo1_B;j<myhi1_B;j++){
        get(Pi+1.A[j][his_col],B[j][fluffCol]);

   } -- directly fetch items and put in fluff column
```

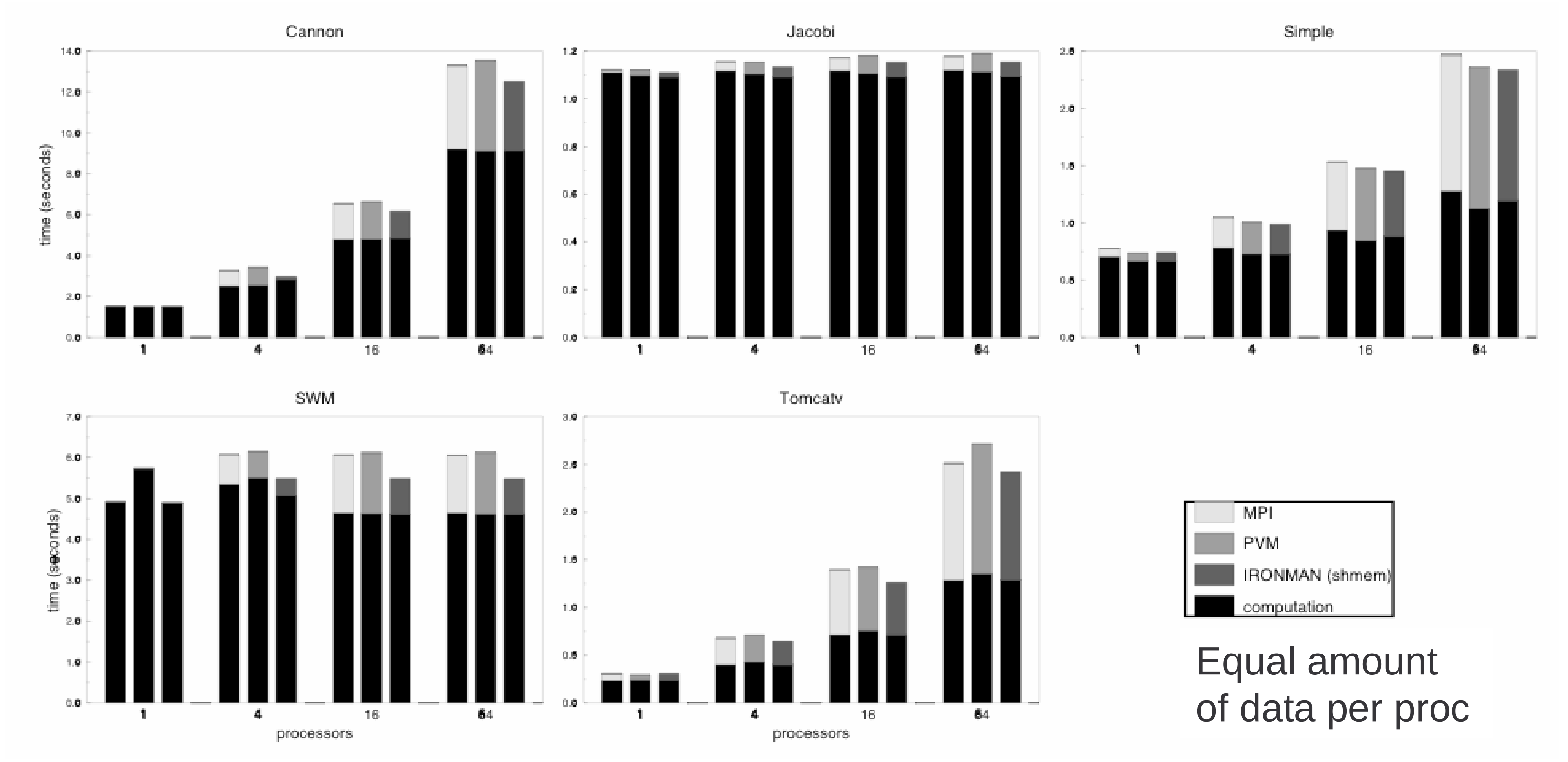
One-sided communication is more efficient because of less waiting and less network traffic

One-sided communication is gaining popularity

Message Passing is the L.C.D.

- Clearly, message passing can be implemented using shared memory and shmem
  – But it must do unnecessary stuff and doesn’t exploit “expensive” part of those machines
- Compilers targeting large computers use message passing to avoid producing 3 back-ends, 3 variations of optimizations, etc.
- How bad can it be?
  + MP still uses fast communication for transmission
  -- MP will go through unnecessary copying, marshalling, demarshalling, copying, handshaking, etc.

Comparison of T3E Communication Types



Equal amount of data per proc

Compilation Challenge for Parallelism

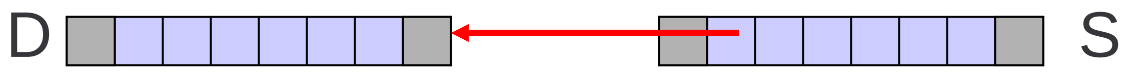
All of these memory models exist on production machines …

- Worried by this problem, the ZPL designers modelled communication by an abstraction called Ironman Communication
  - Ironman abstracts a CTA communication as a load, store
  - Ironman is not biased for/against any comm mechanism

Ironman is designed for compilers, not programmers

Ironman Communication

- The Ironman abstraction says *what* is to be transferred and *when*, but not *how*
- Key idea: 4 procedure calls mark the interval during which communication can occur

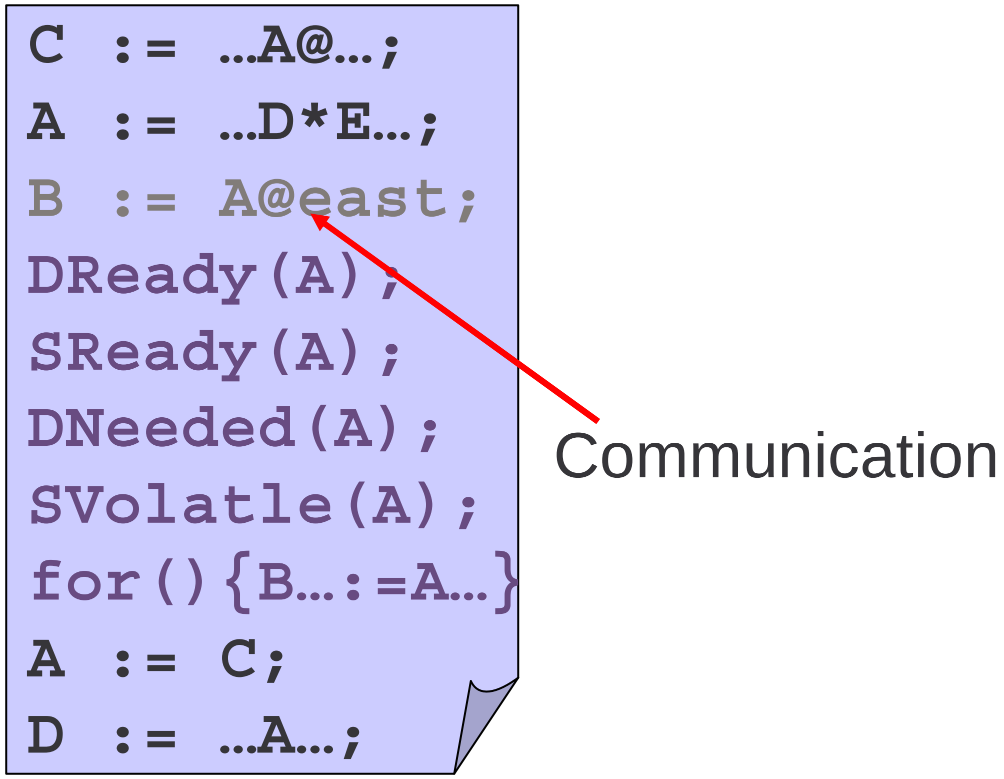


`DR(A)`	= destination location ready to receive data [D side]
`SR(A)`	= source data is ready for transfer [S side]
`DN(A)`	= destination data is now needed [D side]
`SV(A)`	= source location is volatile (to be overwritten) [S side]

- Bound the interval of the source (S) and destination (D) sides of the communication and let the hardware implement the communication

Inserting Communication

- At each communication, the sequence of four procedure calls is inserted in SPMD code



```
C  := …A@…;
A  := …D*E…;
B  := A@east;
DReady(A);
SReady(A);
DNeeded(A);
SVolatle(A);
for(){B…:=A…}
A  := C;
D  := …A…;
```

Ironman Example

Notice the two roles a PE plays in the four calls

P_i

```
A := ...D...;
DR(P_i,A);
SR(P_i-1,A);
   Comm
DN(P_i,A);
SV(P_i-1,A);
A := C;
```

P_{i+1}

```
A := ...D...
DR(P_i+1,A);
SR(P_i,A);
   Comm
DN(P_i+1,A);
SV(P_i,A);
A := C;
```

Destination location ready

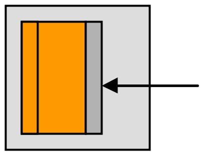
Source data is ready

Communication occurs inside the intervals

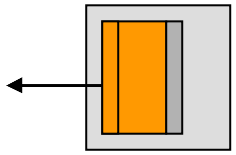
Destination data needed

Source location volatile

Blue indicates one comm



A of P_i



A of P_{i+1}

Ironman Calls

- Every compiled ZPL program uses Ironman calls, but they can have different bindings

	Paragon	MPI Asych	Cray
Destination ready	--	mpi_irecv	post_ready
Source ready	csend	mpi_isend	{wait_ready shmem_put post_done}
Destination needed	crecv	mpi_wait	wait_done
Source volatile	--	mpi_wait	--

One compile, multiple bindings

Optimizing Ironman

Push calls as far apart as possible, expanding the interval for the communication; skew in time

```
C := …A@…;
DReady(A);
A := …D*E…;
SReady(A);
B := A@east;
   Comm
DNeeded(A);
for(){B…:=A…}
SVolatile(A);
A := C;
```

Remember: “Source” is the array itself, “Destination” is the fluff

Ironman Advantages

- Ironman neutralizes different communication models -- avoiding one-size fits all message passing
- Ironman allows the best communication model to be used for the platform
- Extensive optimizations are possible by moving DR, SR calls earlier, and DN, SV calls later … thus reducing wait time and allowing processors to drift in time

Break

- 10 Minutes

Non-shared Memory

Building shared memory in hardware is difficult, as we see next time, and its programming advantages are limited

Leave it out; focus on speed and scaling

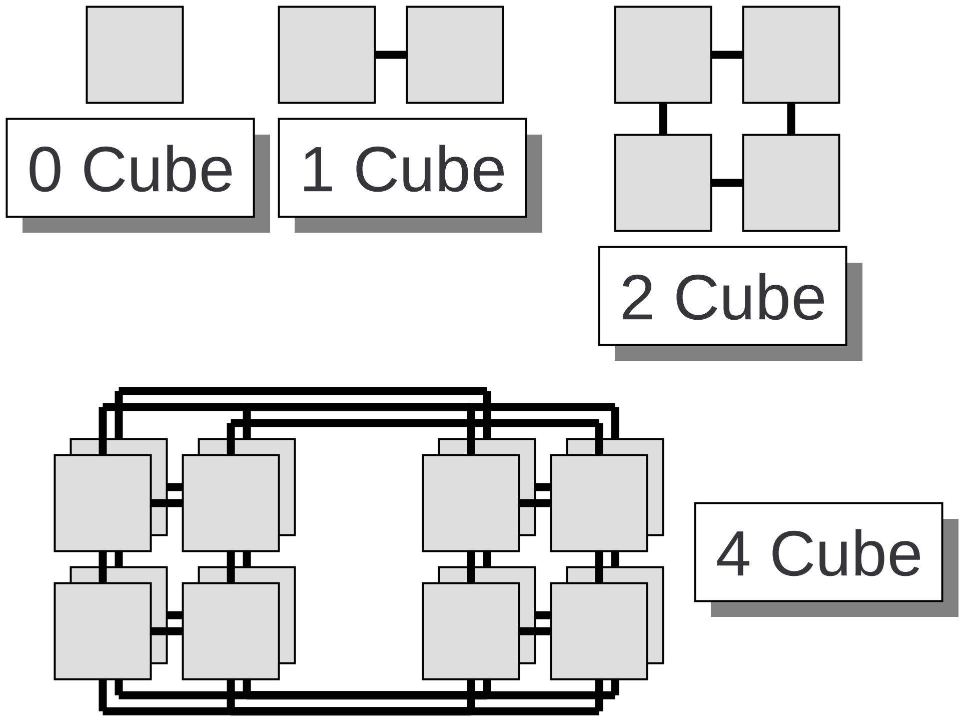
- Three machines
  - nCUBE, an early hypercube architecture -- ‘80s
  - CM-5, architecture that’s “ultimately scalable” -- ’90s
  - XT3 Cray’s and it’s antecedents T3D, T3E -- ‘00s
- Each machine tries to do some aspect of communication well

Focus on 3 Properties of Communication

- Network Interface--how does it connect with the processor element's other parts?
- Network topology--how many hops separate each pair of processors; fewer the better
- Bisection Bandwidth--how many bits can be sent across the bisection of the graph in unit time?
  - Bisection: minimum edge separation of the topology into two equal sets of nodes
  - Generally BB = edges*width*clockrate
  - This is theoretically best

nCUBE/2 A ‘Classic’ Multiprocessor

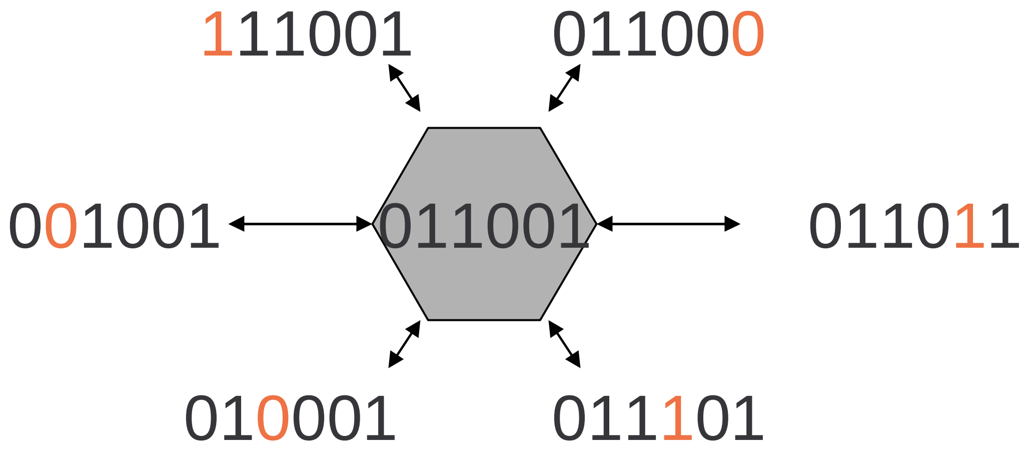
- The nCUBE/2 was a hypercube architecture
- Per node channel capacity grows as $\log_2 P$



Each node has a d bit address To route, “correct” the bits

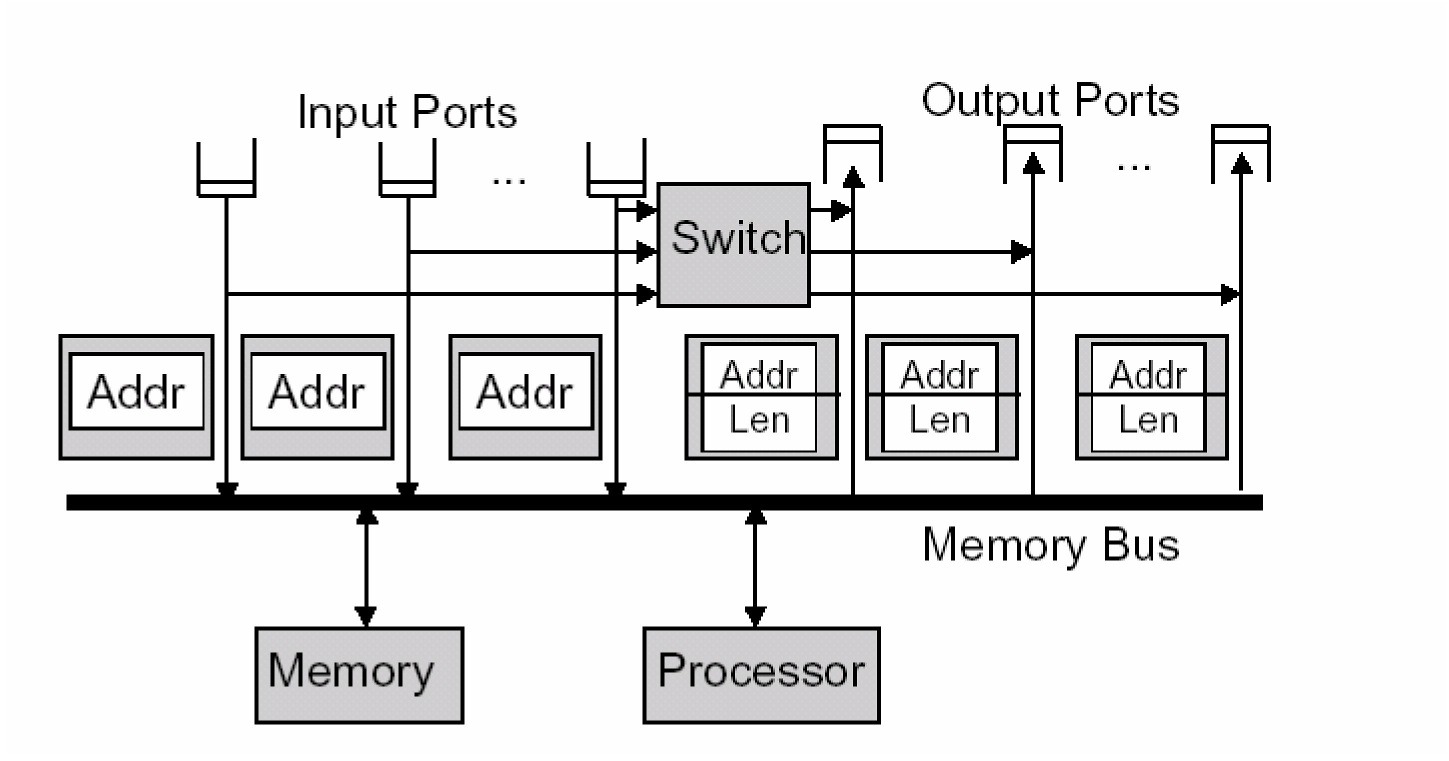
"Correcting The Bit"

- At a node of address x, when packet destined for address y comes in
  - If all bits match … packet has arrived
  - Otherwise, x XOR y, pick a position, i, where there is a mismatch, and send packet out that channel



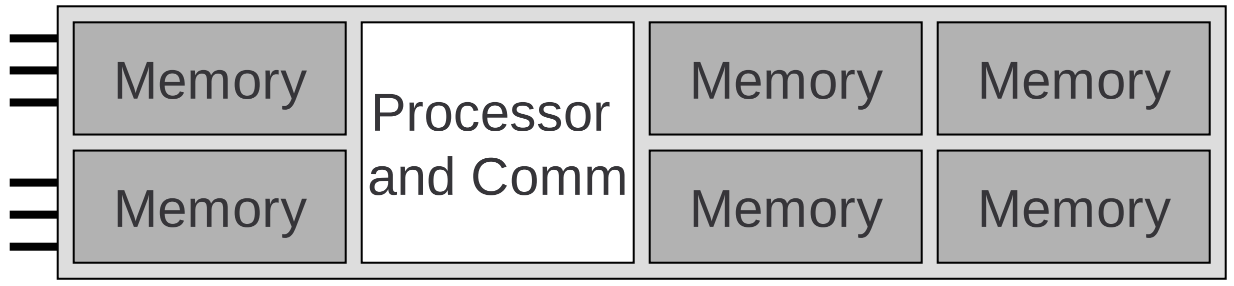
Schematic of nCube Node

Communication Integrated into PE architecture



nCUBE/2 Physical Arrangement

- A single card performed all of the operations, allowing it to be very economical



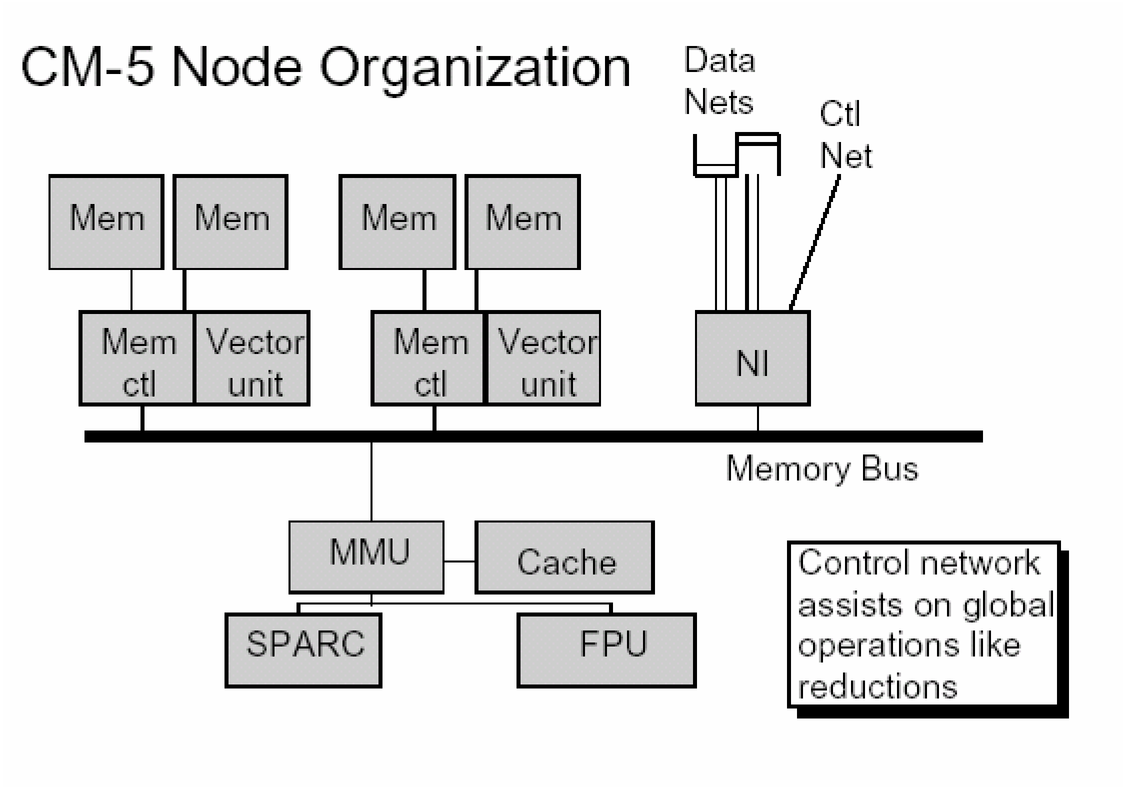
- But adding to the system is impossible … new boards are needed, and new communication -- not so scalable

Connection Machine - 5

- Thinking Machines Inc.’s MIMD machine [Caution: CM-1 and CM-2 are SIMD]
- Goal: Create an architecture that could scale arbitrarily, e.g. 64K processors
- Nodes are standard proc/fpu/mem/NIC
- Scaling came in “powers of 2” using fat tree
- Special hardware performed ‘reductions’
- “Programmed I/O” meant PE was split between comp and comm duties

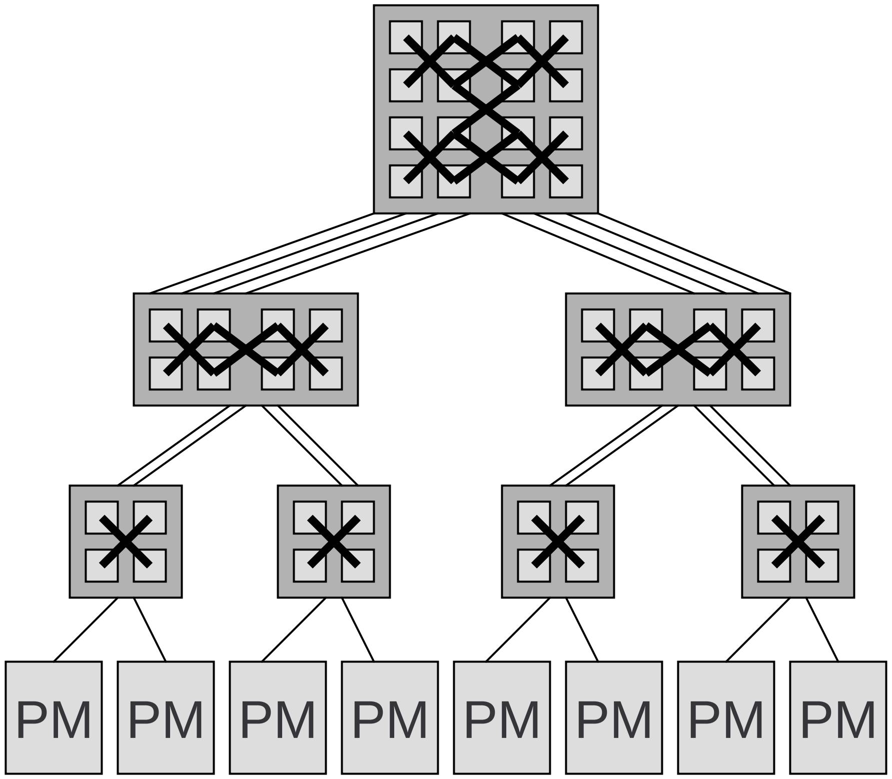
Schematic

- Channel to MMU narrow



CM-5 A “Thinking Machine”

- CM-5 Used a fat tree design

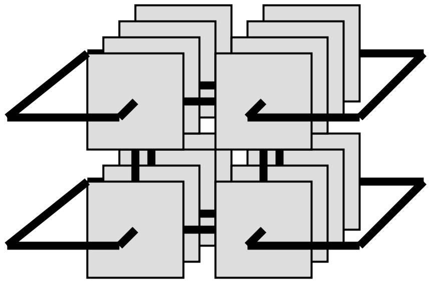


To handle more traffic at higher levels, add more channels and switching capacity

Cray T3D and T3E, Predecessors of XT3

Assume a shared address space -- all processors see the same addresses, *but not the contents*

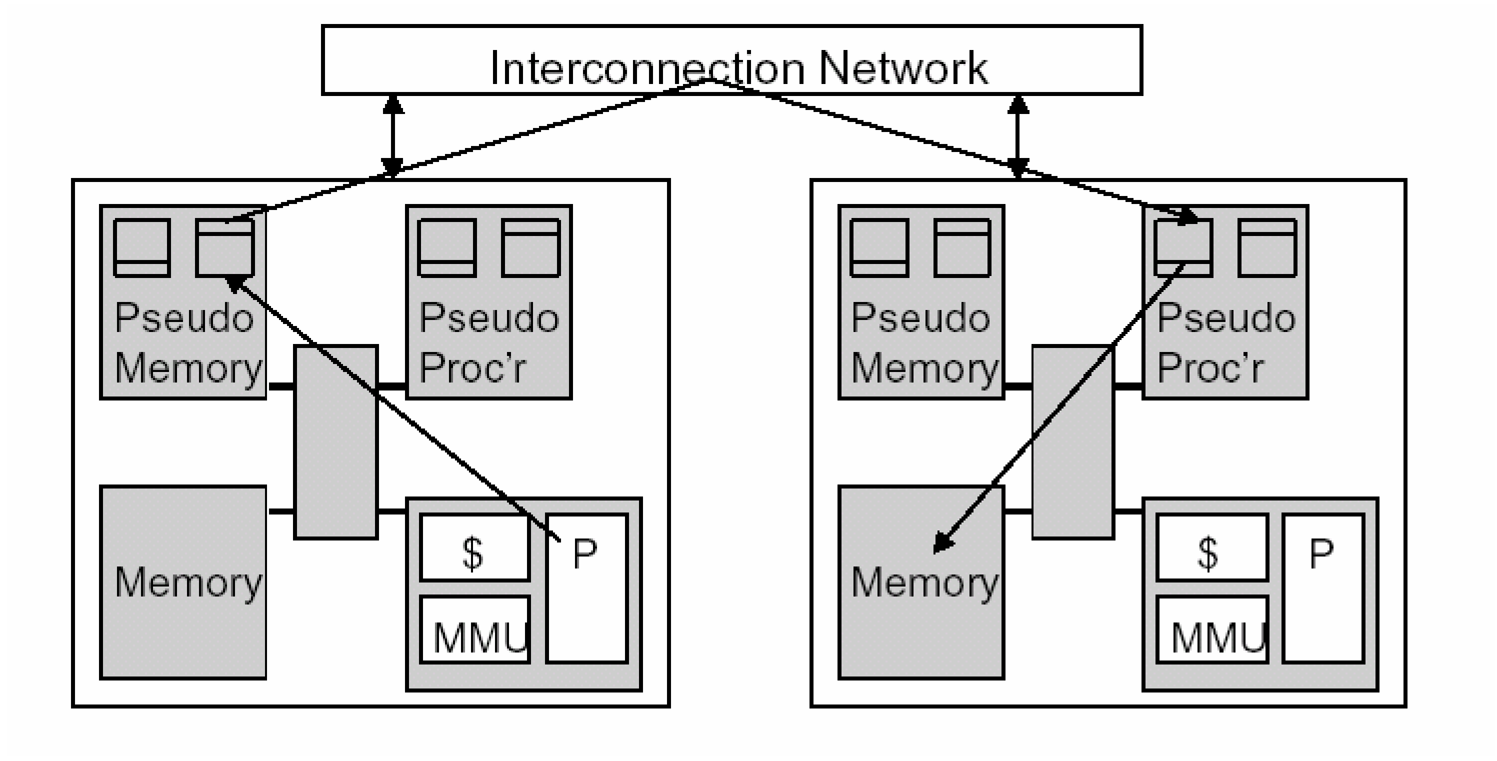
- One-sided communication is implemented using shmem-get and shmem-put
- Result is a non-coherent shared memory



The T3s are three dimensional torus topologies, i.e. a 3D mesh w/wraps

Cray T3

Conceptualize a pseudo-processor and a pseudo-memory implementing get/put



T3D

- Shmem-get and -put eliminate synchronization for the processor, though communication subsystem must
  - Get/Put Are Asymmetric
- There is a short sequence of instructions to initiate a transfer and then ~100 cycles
- A separate network implements global synchronization operations like (eureka), like CTA’s controller network

T3E

- Greater simplification over T3D by using E-registers to abstract pseudo-memory, pseudo-processors
- Gets/Puts instructions move data between global addresses and E-registers
- Read/Modify/Write also possible with E-regs
- Loading Data
    - Put processor address portion in E-register
    - Issue get with a mem-mapped store
    - Actual transfer made from remote processor E-register
    - Load from E-register gets data
- Twice the speed of T3D

(left over from last week)

Recent Language Design

Co-Array Fortran

- Developed within Cray (originally F--) by Numrich&Reed
- Motivated to use T3D/T3E's shmem facilities
- Add's a processor "co-dimension" to the arrays of F95

```
REAL, DIMENSION (N) [*] :: X,Y   !Declare 2 size n vectors
X(:) = Y(:) [PE]      !If PE is same on all images, copy Y to X
```

- Also has a few collective operations, synch. primitives
- CAF provides a clean way to manage (shmem) communication in a "local view" language … machine model is CTA
- Cray supports CAF

(left over from last week)

MM in CoArray Fortran

```
real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
  do q=1,p
     c(i,j)[myP,myQ] = c(i,j)[myP,myQ]
                         + a(i,k)[myP, q]*b(k,j)[q,myQ]
  enddo
enddo
```

Reduction

```
subroutine globalSum(x)
real(kind=8),dimension[0:*] :: x
real(kind=8) :: work
integer n,bit,i, mypal,dim,me, m
dim = log2_images()
if(dim .eq. 0) return
m = 2**dim
bit = 1
me = this_image(x)
do i=1,dim
    mypal=xor(me,bit)
    bit=shiftl(bit,1)
     call sync_all()
     work = x[mypal]
     call sync_all()
     x=x+work
enddo
end subroutine globalSum
```

$$\begin{array}{r} 011001 \\ \text{xor } \underline{000001} \\ 011000 \end{array}$$

XT3 -- Next Generation Scalable Machine

The T3D, T3E are old --most are retired

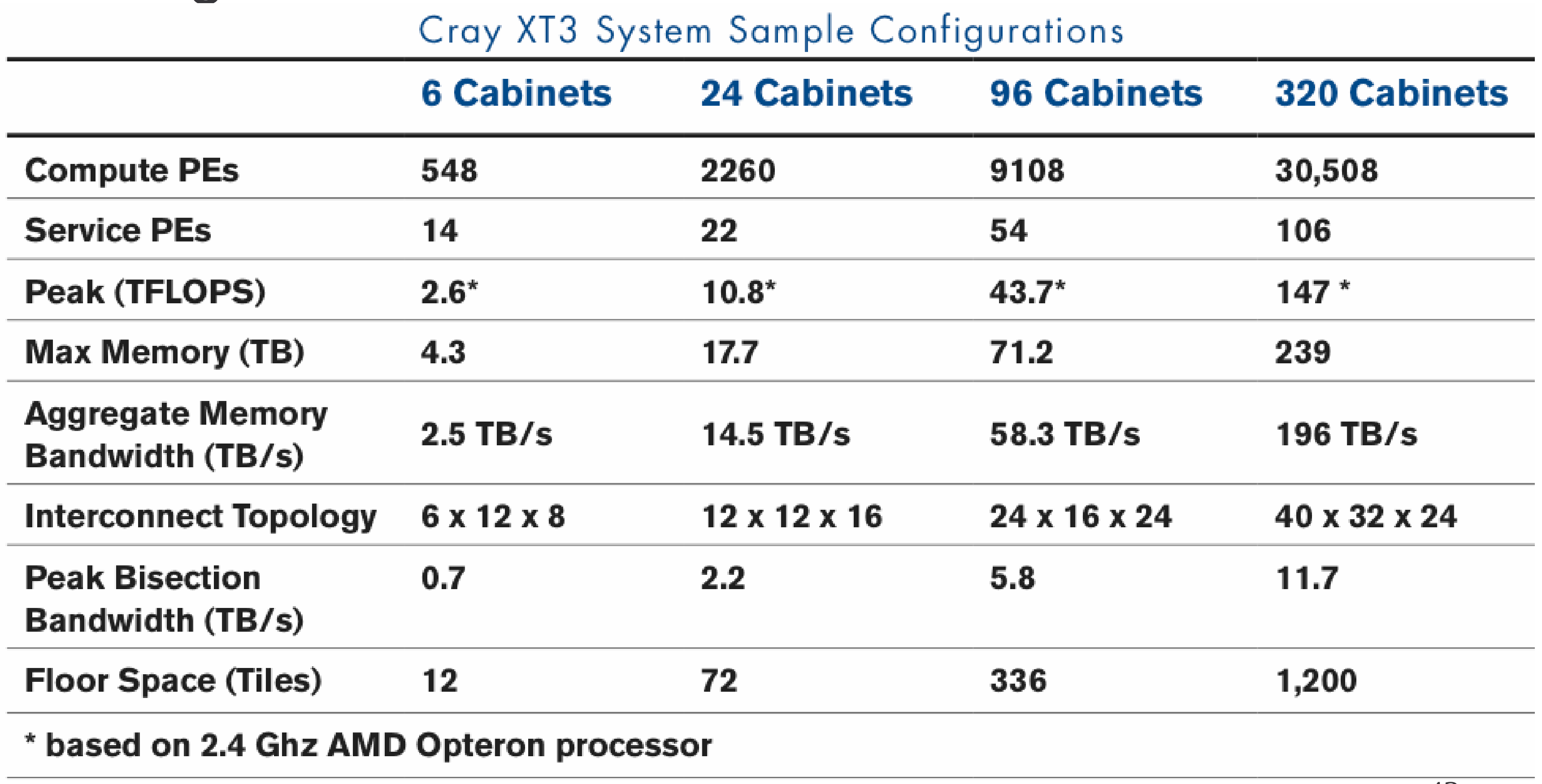
- Cray’s latest entry is the XT3

  Glossy Literature Says It All:

  “Purpose-built to meet the special needs of capability class HPC applications, each feature and function is designed for larger problems, faster solutions, and a greater return on investment.”

XT3 Data Sheet

Configurations

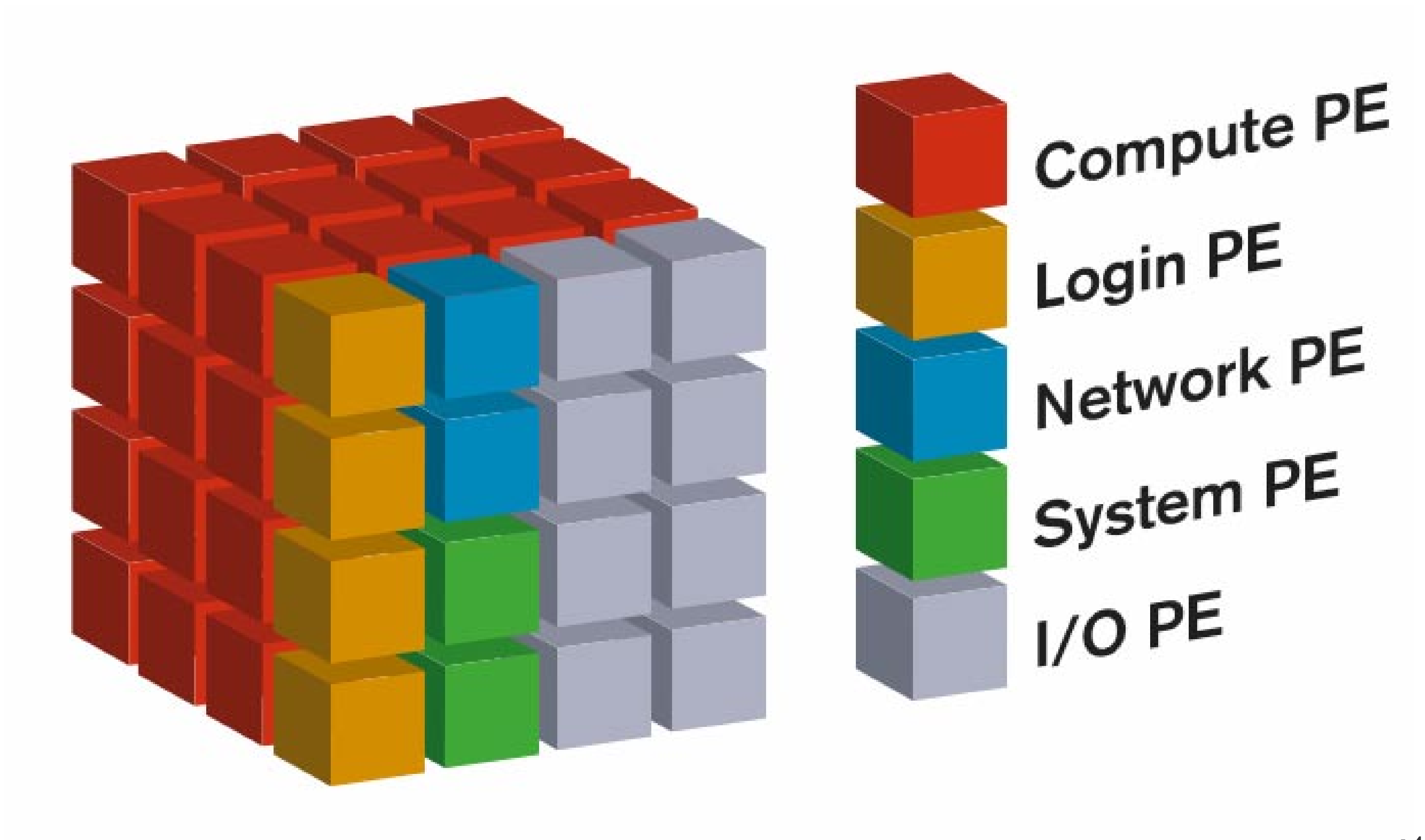


Cray XT3 System Sample Configurations

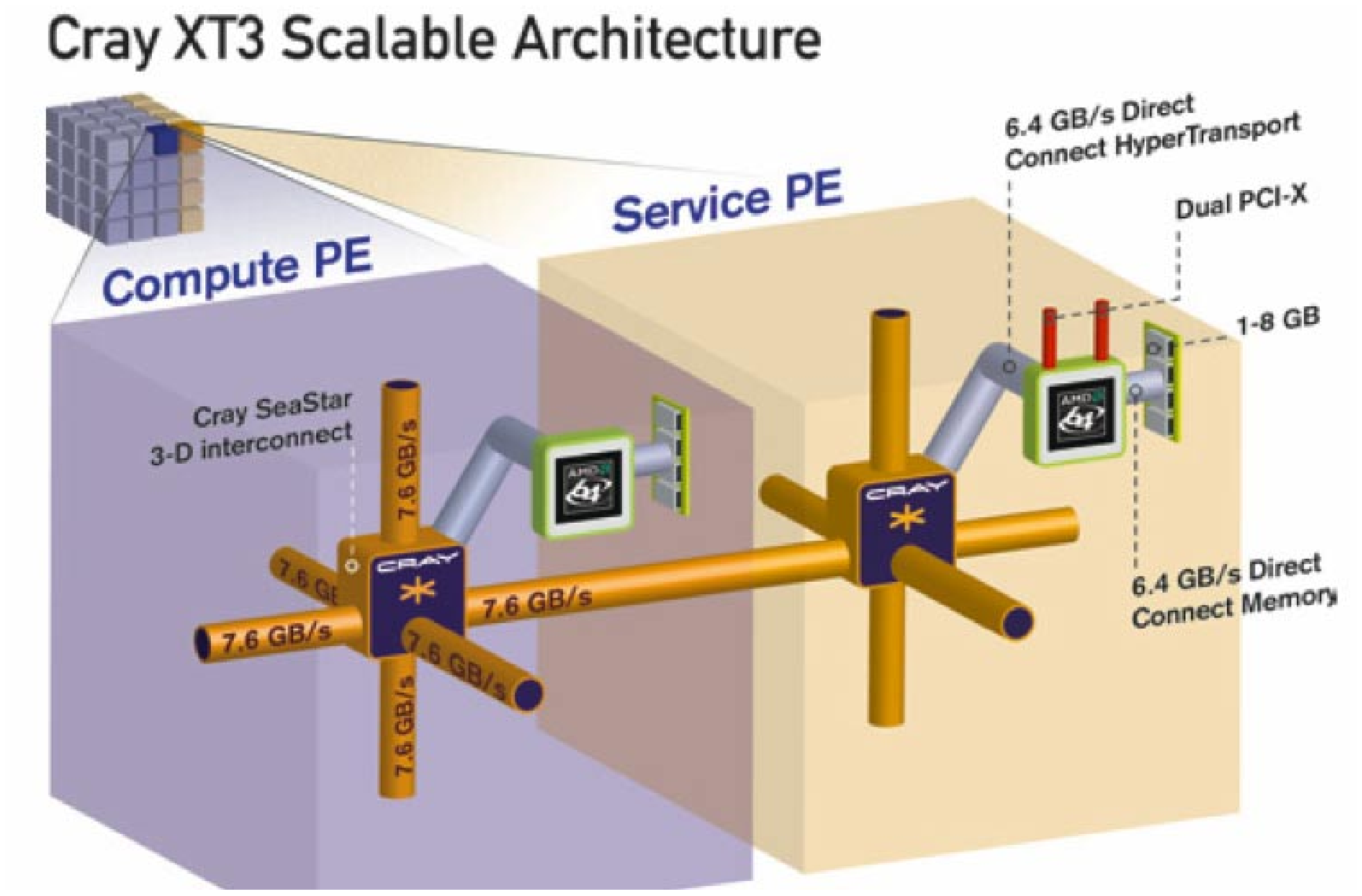
	6 Cabinets	24 Cabinets	96 Cabinets	320 Cabinets
Compute PEs	548	2260	9108	30,508
Service PEs	14	22	54	106
Peak (TFLOPS)	2.6*	10.8*	43.7*	147 *
Max Memory (TB)	4.3	17.7	71.2	239
Aggregate Memory Bandwidth (TB/s)	2.5 TB/s	14.5 TB/s	58.3 TB/s	196 TB/s
Interconnect Topology	6 x 12 x 8	12 x 12 x 16	24 x 16 x 24	40 x 32 x 24
Peak Bisection Bandwidth (TB/s)	0.7	2.2	5.8	11.7
Floor Space (Tiles)	12	72	336	1,200

* based on 2.4 Ghz AMD Opteron processor

XT3 Architecture Cartoon



XT3 Uses 3D Torus Interconnect


Cray XT3 Scalable Architecture
Compute PE
Service PE
6.4 GB/s Direct Connect HyperTransport
Dual PCI-X
1-8 GB
Cray SeaStar 3-D interconnect
7.6 GB/s
7.6 GB/s
7.6 GB/s
7.6 GB/s
7.6 GB/s
7.6 GB/s
6.4 GB/s Direct Connect Memory
CRAY
CRAY

Data Sheet

CPU 64-bit AMD Opteron 100 series processors; up to 96 per cabinet

Cache 64K L1 I cache, 64K L1 D cache, 1 MB L2 cache per processor

FLOPS 460 GFLOPS per cabinet (96 processors @ 2.4 GHz)

Main Memory 1-8 GB Registered ECC SDRAM per processor

Memory Bandwidth 6.4 GB/s per processor

Interconnect 1 Cray SeaStar routing and communications ASIC per Opteron
6 switch ports per Cray SeaStar chip, 7.6 GB/s each
(45.6 GB/s switching capacity per Cray SeaStar chip)
3 dimensional torus interconnect 3 microsecond MPI latency between PEs

External I/O 2 independent 64-bit 133 MHz PCI-X buses per service PE
Gigabit Ethernet PCI-X card (copper and optical)
Dual-Port 2 GB/s Fibre Channel Host Bus Adapter (optical)
10 Gigabit Ethernet card (Optical)

Disk 4 and 8 port Fibre Channel RAID controllers
Configurable Fibre Channel RAID drive sets File System Lustre File System

Data Sheet

CPU 64-bit AMD Opteron 100 series processors; up to 96 per cabinet
Cache 64K L1 I cache, 64K L1 D cache, 1 MB L2 cache per processor
FLOPS 460 GFLOPS per cabinet (96 processors @ 2.4 GHz)
Main Memory 1-8 GB Registered ECC SDRAM per processor
Memory Bandwidth 6.4 GB/s per processor
Interconnect 1 Cray SeaStar routing and communications ASIC per Opteron
6 switch ports per Cray SeaStar chip, 7.6 GB/s each
(45.6 GB/s switching capacity per Cray SeaStar chip)
3 dimensional torus interconnect 3 microsecond MPI latency between PEs
External I/O 2 independent 64-bit 133 MHz PCI-X buses per service PE
Gigabit Ethernet PCI-X card
Dual-Port 2 GB/s Fibre Char
10 Gigabit Ethernet card (Op
Disk 4 and 8 port Fibre Chann
Configurable Fibre Channel

Estimate λ for XT3:

3µ latency, 2.4 GHz clock imply
7.2 KHz per communication x IPC

Clusters (a\k\a Beowulf Clusters)

Clusters are “home made” parallel computers using commodity parts and simple engineering

- Beowulf Hardware
  - Node Hardware
    - Rates, Latencies and Bandwidths
      - Microbenchmarking Tools
      - Lmbench Results
      - Netperf Results
      - CPU Results
    - Conclusions
  - Network Hardware
    - Basic Networking 101
      - Networking Concepts
      - TCP/IP
    - Ethernet
      - 10 Mbps Ethernet
      - 100 Mbps Ethernet
      - 1000 Mbps Ethernet
    - The Dolphin Serial Channel Interconnect
    - Myrinet

Robert G Brown
Engineering a Beowulf Style Cluster

http://www.phy.duke.edu/~rgb/Beowulf/beowulf_book/beowulf_book/index.html

Northshore Cluster

- Northshore is
  - 32 Pentium 4 PEs (some may be down)
  - 2.4 GHz
  - 1GB memory
  - GigaBit Ethernet Interconnect
- 60 more processors to be added soon

Summary

- Non-Shared memory computers scale
- Communication design is important
    - NIC’s relationship to other parts
    - Network topology
    - Bisection bandwidth
- The “rap” on Non-Shared memory machines is that the work of orchestrating communication is explicit in software

The response is, “Yes, but it doesn’t have to be explicit in the code the programmer writes”

∴Spend design $ on network and processor performance not on memory sharing

Citations

Bradford L. Chamberlain, Sung-Eun Choi, and Lawrence Snyder. **A compiler abstraction for machine independent parallel communication generation.** In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 1997

R.W. Numrich and J.K. Reid, “Co-Array Fortran for Parallel Programming”, ACM Fortran Forum, 17(2):1-31, 1998