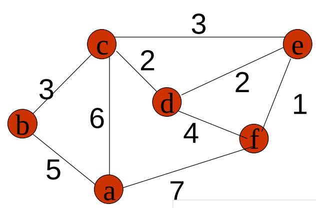
The Parallel Runtime

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 be our main focus.

1

Floyd-Warshall Alg (Homework not assigned)

- Accept an nxn (symmetric) array **E** of edge weights with 0 on diagonals, ℜ for no edge
- Compute the all pairs shortest path: min(d[i,j],d[i,k]+d[k,j])



E	a	b	c	d	e	f
a	0	5	6	ℜ	ℜ	7
b	5	0	3	ℜ	ℜ	ℜ
c	6	3	0	2	3	ℜ
d	ℜ	ℜ	2	0	2	4
e	ℜ	ℜ	3	2	0	1
f	7	ℜ	ℜ	4	1	0

Assume all edge weights are less than 1000

2

A Homework Solution

```
program FW;
config var n : integer = 10;
region R = [1..n,1..n];
       H = [*,1..n];
       V = [1..n,*];
var    E : [R] integer;
      Hk : [H] integer;
      Vk : [V] integer;
procedure FW();
var k : integer;
[R] begin
    -- Read E here, infinity is 10K
     for k := 1 to n do
[H]      Hk := >> [k, ]E;
[V]      Vk := >> [ ,k]E;
           E := min(E, Hk + Vk);
     end;
    -- Write E here
end;
```

3

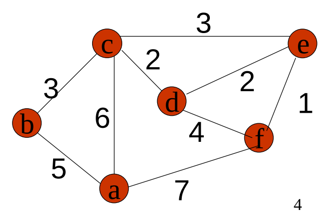
Example, Connecting Through (a)

E	a	b	c	d	e	f
a	0	5	6	ℜ	ℜ	7
b	5	0	3	ℜ	ℜ	ℜ
c	6	3	0	2	3	ℜ
d	ℜ	ℜ	2	0	2	4
e	ℜ	ℜ	3	2	0	1
f	7	ℜ	ℜ	4	1	0

Hk	-	-	-	-	-	-
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7

min(ℜ,7+6)=13

Vk	-	-	-	-	-	-
-	0	0	0	0	0	0
-	5	5	5	5	5	5
-	6	6	6	6	6	6
-	ℜ	ℜ	ℜ	ℜ	ℜ	ℜ
-	ℜ	ℜ	ℜ	ℜ	ℜ	ℜ
-	7	7	7	7	7	7



4

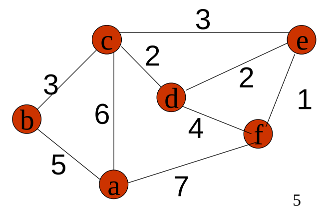
Example, Connecting Through (a)

E	a	b	c	d	e	f
a	0	5	6	ℜ	ℜ	7
b	5	0	3	ℜ	ℜ	ℜ
c	6	3	0	2	3	ℜ
d	ℜ	ℜ	2	0	2	4
e	ℜ	ℜ	3	2	0	1
f	7	ℜ	ℜ	4	1	0

Hk	-	-	-	-	-	-
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7
-	0	5	6	ℜ	ℜ	7

`min(3,5+6)=3`

Vk	-	-	-	-	-	-
-	0	0	0	0	0	0
-	5	5	5	5	5	5
-	6	6	6	6	6	6
-	ℜ	ℜ	ℜ	ℜ	ℜ	ℜ
-	ℜ	ℜ	ℜ	ℜ	ℜ	ℜ
-	7	7	7	7	7	7



5

Homework: Performance Model & UDRs

The PSP paper gives two mode computations

(a) Use WYSIWYG analysis to say which is better

(b) Create custom “maxmode” to improve last line

```
-- "Standard" mode code
[1..n] begin
  S := 0;
  for i := 1 to n do
[i..n] S += ((>>[i] V) = V);
  end;
  count := max<< S; -- largest freq count
  mode := max<<((count = S) * V); -- get mode
end;
```

6

PSP Mode

```
-- PSP mode code
[1,1..n] begin -- assume R = [1..n,1..n]
             -- assume row 1 of V is input
[1..n,1]    Vt := V#[Index2,Index1]; -- transp
             -- Replicate, compute and collapse
           S := +<<[R] (>>[1,]V = >>[,1]Vt);
           count := max<<  S;
           mode := max<< ((count = S)*V);
       end;
```

Hints: Reasoning is what counts in (a); in (b) use global data reference

7

Non-shared Memory

Building shared memory in hardware is difficult, and its programming advantages are limited

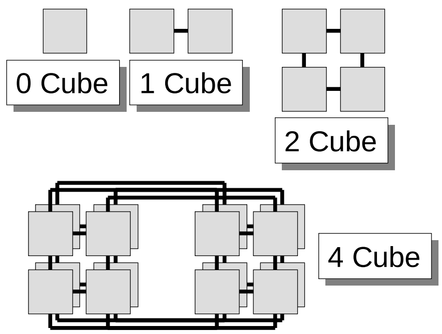
Leave it out; focus on speed and scaling

- Three machines
  - nCUBE, an early hypercube architecture
  - CM-5, connection machine's "ultimately scalable"
  - T3D/T3E Cray's first foray into shared address sp
- Each machine tries to do some aspect of communication well

8

nCUBE/2 A ‘Classic’ Multiprocessor

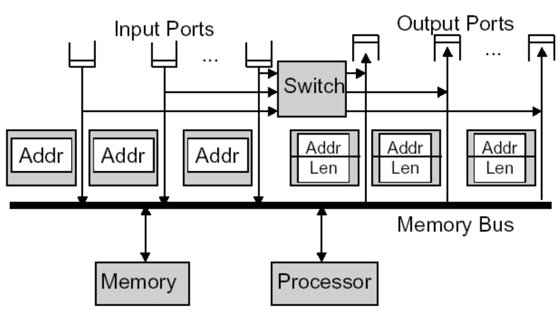
- The nCUBE/2 was a hypercube architecture
- Per node channel capacity grows as $\log_2 P$



Each node has a d bit address To go from d to d^d, “correct” bits

Schematic of Node

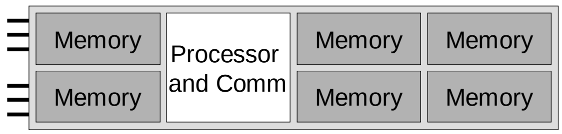
Communication Integrated into PE architecture



10

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

11

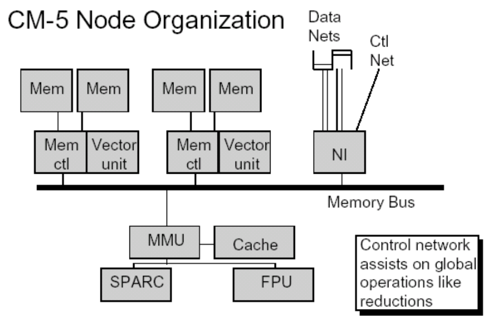
Connection Machine - 5

- Thinking Machines Inc.'s MIMD machine [Caution: CM-1 and CM-2 are SIMD]
- Goal: Create an architecture that could scale arbitrarily
- 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

12

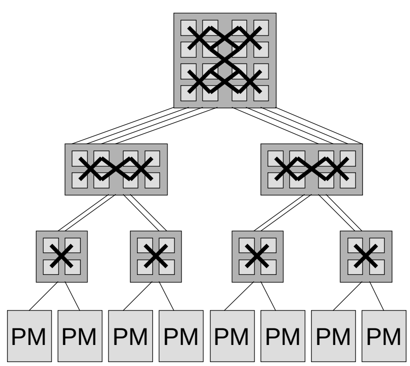
Schematic

- Channel to MMU narrow



CM-5 A "Thinking Machine"

- CM-5 Used a fat tree design

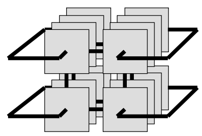


To handle more traffic at higher levels, add more channels and switching capacity

Cray T3D and T3E

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

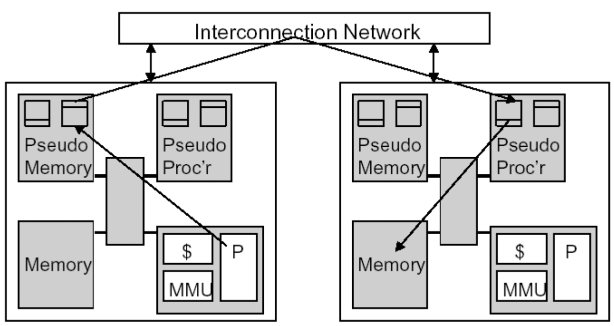


The T3s are three dimensional torus topologies, i.e. a 3D mesh w/wraps

15

Cray T3

Conceptualize a pseudo-processor and a pseudo-memory



16

T3D

- Shmem-get and -put eliminate synchronization for the processor, though communication subsystem must
  - 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)

17

T3E

- Greater simplification over T3D by using 512 E-registers for loading/storing
- 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

18

Moving Data In Parallel Computation

Two views of data motion in parallel computation

- It should be transparent -- shared memory
  - Data movement is complex … simplify by eliminating it
  - Analogous mechanisms (VM, paging, caching) have proved their worth and show that amortizing costs works
- It is the programmer’s responsibility to move data to wherever it is needed -- message passing
  - Data movement is complex ... rely on programmer to do it well
  - Message passing is universal -- it works on any machine while shared memory needs special hardware

Many furious battles have taken place over this issue … at the moment message passing is the state-of-the-art

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, wait until it is acked
  - Non-blocking send … send msg, continue execution
  - Wait_for_ACK … wait for ack of non-blocking send
  - Receive … get msg that has arrived
- Programmers insert the library calls in-line in C or Fortran programs

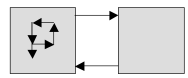
20

Message Passing Example

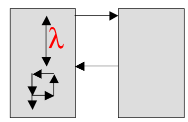
In message passing, there is no abstraction … the programmer does everything

- Consider overlapping comm/comp
  - When exchanging data in middle of a computation

```
nb_send(to_right, data1);
much computing;
recv(from_left, loc1);
```



  - The programmer knows that **`data1`** will not be used, and so there can be no error in delaying the **`recv`**
  - … but compilers usually must be much more conservative



Compiler uses blocking send

Alternative Middle Communication

- A “lighter weight” approach is the *one-sided communication*, *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

Alternative Implementation

- Message passing is "heavy weight" because it needs send, acknowledgement, marshalling
- Using one-sided communication is easy

```
my_temp := data1;      -- store where neighbor can get it
post(P-1,my_data_ready);  -- say that it's available
much computing;        -- overlap
wait(P+1,his_data_ready); -- wait if neighbor not ready
get(P+1.his_temp, loc1); -- get it now
```

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

Most computers do not implement shmem

23

Msg Pass Lowest Common Denominator

- Most programmers write direct message passing code
  - With explicit message passing statements in code it is difficult to adapt to new computer

Msg passing, shmem, shared are all different conceptions

- All compilers targeting large parallel machines (except ZPL) use message passing
  - Unable to exploit other communication models

Message passing, shmem, shared require different compiler formulation

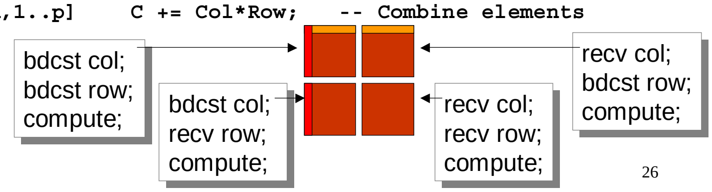
24

Break

Compiling ZPL Programs

- ZPL uses a “single program, multiple data” (SPMD) view ⇒ compiler produces 1 program
- Logically, ZPL executes 1 statement at a time, but processors go at their own rate using “data synchronization”

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

The actual form of communication is given below

27

Compiling ZPL Programs

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

```
rightedge  := max<< Pts.x;
topedge    := max<< Pts.y;
leftedge   := min<< Pts.x;
bottomedge := min<< Pts.y;
```

converts to 1 Ladner/Fischer tree on 4-part data

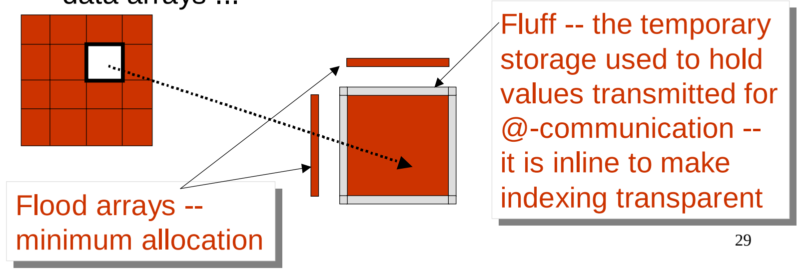
```
North := A@N + B@N + C@N;
```

combines all communication to north (and south) neighbors

28

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 ...



29

There are lots of different machines

- Programmers will generate code for different code for different machines
    - Shared memory
    - Message Passing
    - Shmem
- What should a compiler do???
- Begin with some examples

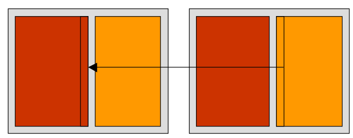
30

Example -- shared memory

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

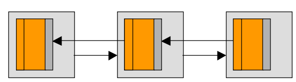
- In the shared memory model each processor writes directly into the portion of B that it ‘owns,’ referencing elements of A as needed
- No explicit ‘fluff’ regions, but synch needed

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

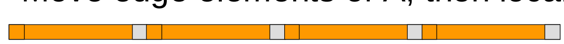


Example -- message passing

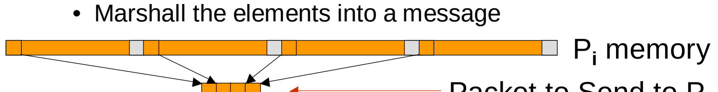
```
B := A@east ...;
```



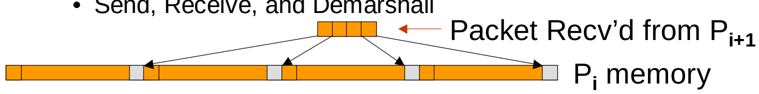
- Move edge elements of A, then local copy to B



- Message passing …
  - Marshall the elements into a message



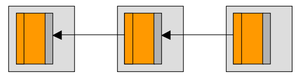
  - Send, Receive, and Demarshall



32

Example -- one-sided communication

B := A@east ...;



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

```
post(my_data_ready);      -- say it's available
wait(P+1,his_data_ready); --wait if neighbor ~ ready
get(P+1.my_low1,his_col);--addr of P+1 1st col
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
```

33

Compilation Challenge for Parallelism

- All of these memory models exist on production machines …
- How can a single compiler target all models?
- Worried by this problem, the ZPL designers modeled communication by an abstraction called Ironman Communication
    - Ironman abstracts a CTA communication as a time-dependent load, store
    - Ironman is not biased for/against any comm mechanism

Ironman is designed for compilers, not programmers

34

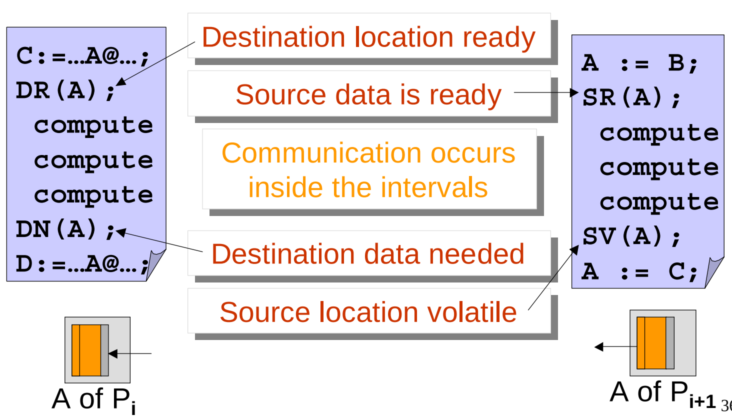
Ironman Communication

- The Ironman abstraction says *what* is to be transferred and *when*, but not *how*
- Key idea: 4 procedure calls mark the intervals during which communication can occur
  - `DR(A)` = destination location ready to receive data [R side]
  - `SR(A)` = source data is ready for transfer [S side]
  - `DN(A)` = destination data is now needed [R side]
  - `SV(A)` = source location is volatile (to be overwritten) [S side]
- Bound the interval on the sending (S) and receiving (R) sides of the communication and let the hardware implement the communication

35

Ironman Example

Placement of the Ironman procedure calls



36

Ironman Calls

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

	nCube	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	--

37

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

38

Summary

- There are three basic techniques for memory reference and communication
  - Coherent shared memory w/ transparent communication
  - Local memory access with message passing -- everything is left to the programmer
  - One-sided communication, a variation on message passing in which `get` and `put` are used
- Message passing is state-of-the-art for both programmers and compilers (except ZPL)
- Ironman is ZPL's communication abstraction that neutralizes differences & enables optimizations

39