Message Passing and Network Fundamentals
ASD Distributed Memory HPC Workshop

Computer Systems Group

Research School of Computer Science
Australian National University
Canberra, Australia

October 30, 2017
Day 1: Message Passing and Network Fundamentals

<table>
<thead>
<tr>
<th>Time</th>
<th>Lecture Topics</th>
<th>Hands-On Exercise</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:00</td>
<td>Introduction and Course Overview Elements of Message Passing</td>
<td>Introduction to the NCI Raijin System and MPI</td>
<td>Peter Strazdins</td>
</tr>
<tr>
<td>10:30</td>
<td>COFFEE BREAK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:00</td>
<td>Message Passing Semantics and Protocols</td>
<td>MPI Message Semantics</td>
<td></td>
</tr>
<tr>
<td>12:30</td>
<td>LUNCH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13:30</td>
<td>System Area Networks</td>
<td>MPI Message Performance</td>
<td></td>
</tr>
<tr>
<td>15:00</td>
<td>AFTERNOON TEA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15:30</td>
<td>Routing and Communication Costs</td>
<td>Simulating Routing Protocols</td>
<td></td>
</tr>
</tbody>
</table>

Message Passing and Networks lecture slides (pdf)
Outline

1. Introduction and Course Overview
2. Elements of Message Passing
3. Message Passing Semantics and Protocols
4. System Area Networks
5. Routing and Communication Costs
Introduction and Course Overview

Introduction to Distributed Memory HPC

- hardware:
  - system area networks, routing, communication costs
  - parallel I/O (filesystems)
  - fault tolerance

- parallelization strategies
  - embarrassingly parallel, data partitioning, synchronous computation
  - algorithm case studies

- message passing programming model:
  - algorithms: collective communications
  - programming: MPI basics, collectives, extensions, I/O
  - hybrid parallelism
  - system support: OS concerns, runtimes, low-level implementation

- alternate programming models (PGAS) and their support
Motivations:

- shared memory systems may not scale sufficiently (for processing speed, memory and I/O)
- main programming model (message passing) has many advantages
  - well-defined semantics; works on shared memory (safer? faster??)
  - used also for distributed computing (networking)
  - increasingly used in non-cache coherent processors

But it brings its own challenges:

- more complex algorithm design, reducing overheads (communication costs, load imbalance, serial portions), debugging, sheer scale
NCI’s Raijin: A Petascale Supercomputer

- 57,472 cores (dual socket, 8 core Intel Xeon Sandy Bridge, 2.6 GHz) in 3592 nodes
- 160 TBytes (approx.) of main memory
- Mellanox Infiniband FDR interconnect (52 km cable),
- interconnects: ring (cores), full (sockets), fat tree (nodes)
- 10 PBytes (approx.) of usable fast filesystem (for shortterm scratch space apps, home directories)
- power: 1.5 MW max. load
- 24th fastest in the world in debut (November 2012); first petaflop system in Australia
  - fastest (||) filesystem in the s. hemisphere
NCI’s Integrated High Performance Environment

- NCI data movers
- VMware
- Cloud
- Raijin Login + Data movers
- Raijin HPC Compute
- To Huxley DC

10 GigE
/g/data 56Gb FDR IB Fabric
Raijin 56Gb FDR IB Fabric

Massdata (tape)
Persistent global parallel filesystem
Raijin high-speed filesystem

Cache 1.0PB
/g/data1
/g/data2
/short
/home
Outline

1. Introduction and Course Overview
2. Elements of Message Passing
3. Message Passing Semantics and Protocols
4. System Area Networks
5. Routing and Communication Costs
The Options for Message Passing

1. design a special parallel programming language
   - Occam, Go

2. extend the syntax of an existing sequential language
   - CC++ (extension to C++), Fortran M, Linda (C or Fortran based)

3. use a standard sequential language with special library
   - most common, e.g. MPI, PVM, P4, TCGMSG

*We will take option 3 and use MPI: see [www.mpi-forum.org](http://www.mpi-forum.org)*
MPI-1: Message Passing Interface

- parallel computer vendors initially developed own message-passing APIs
  - e.g. Fujitsu’s APLib for the AP1000 series (1991–1998)
  - issue: portability across machines difficult (especially with subtle differences in semantics)
- early work on a standard started in 1992 at Oak Ridge National and Rice Uni
- over 40 academic and government participants
- at that stage, there was a plethora of different message passing environments
- some deliberate exclusions from MPI-1:
  - IO and dynamic process creation
  - debugging and profiling tools were outside scope
- target was C and Fortran applications, although not strongly
- MPI-1 released in May 94
  - contained: point-to-point communications, collective operations, processor topology
- minor clarifications: MPI 1.1 (June 95), MPI 1.2 (July 97)
Basic Requirement #1: Process Creation

Require a method for creating separate processes for execution on different CPUs: `spawn(name_of_executable, where_to_run_it)`

- **options:**
  - static: number of processes fixed throughout execution
  - dynamic: number of processes fluctuates during execution
- both require a means of identifying each process uniquely

The initial MPI standard (MPI-1) did not permit dynamic spawning:

- the number of processes is defined by runtime environment, e.g.
  
  ```
  mpirun -np 4 mpi_job
  ```

- MPI provides a function to identify the number of processes and a unique identifier for each process (the rank of the process within the parallel group of processes):

```
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```
Basic Requirement #2: Data Transmission

Require a method for sending/receiving messages between processes:

\[ \text{send(data, to\_where)} \text{ and receive(data, from\_where)} \]

- **data**
  - usually a pointer and number of bytes
  - non-contiguous data must be packed
    (note however that we can create datatypes for strided vectors and sub-arrays)
  - heterogeneous systems may require type conversion (e.g. big/little endian)

- MPI send and receive calls:

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```
Consider a two-process MPI program, attempting send each other’s rank:

```c
const int TAG = 99; int rank, otherRank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if (rank == 0) {
    MPI_Send(&rank, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
    MPI_Recv(&otherRank, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
} else {
    MPI_Recv(&otherRank, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Send(&rank, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
}
assert(otherRank == 1 - rank);
```
Rolling Your Own

UNIX provides you with all you need to build your own message passing library:

- `fork` - spawns an identical task to parent
- `ssh` - starts process on a remote machine
- `exec` - overwrites a process with a new process
- `sockets` - provide communication between machines
- `shmget` - provides communication within shared memory
- `xdr` - provides data conversion between machines

MPI implementations (MPICH, OpenMPI etc) use these utilities, e.g. on NCI Raijin system, CSIT labs
Hands-on Exercise: Intro. to Raijin and MPI
Outline

1. Introduction and Course Overview
2. Elements of Message Passing
3. Message Passing Semantics and Protocols
4. System Area Networks
5. Routing and Communication Costs
Message Transfer Semantics: Definitions

- **synchronous**: the send only returns when message has been received, e.g. typical 3 message protocol:
  - request to send
  - receive the OK to send
  - send message
- **blocking**: the send returns when it is safe to re-use the sending buffer
  - **locally blocking**: returns after MPI copies the message into a local buffer
  - **globally blocking**: returns when receiver has collected the message (and hence has posted its receive call)

The receiver returns when message has been received.

- **non-blocking**: the send returns *immediately* even though the data may still be in the original buffer
  - another function call is used to check that the buffer is free to use again

The receiver also returns *immediately*; another call is used to check for arrival.
Message Transfer

- Process 1 sends a message to Process 2.
- Process 2 receives the message.
- Time stamps and message transfer details are shown.

Message Passing Semantics and Protocols
MPI Eager Message Protocol

(for **locally blocking messages**)

- if message arrives before receive call is posted, receiver will buffer message in a system area
  - generally limited to small messages (max. message size may decrease with number of processes)
  - a (small) number of pre-allocated buffers will be kept for this purpose
    - it is generally not possible to allocate a buffer as the message arrives
    - e.g. Infiniband requires buffers to be **registered** (and pinned in memory)

- advantages: minimizes latency and synchronization delay
- disadvantages: memory wastage, if not used; not scalable; extra copy
MPI Rendezvous Message Protocol

(for **globally blocking messages**)

- consists of the following steps:
  1. sender sends the **message envelope** (tag, communicator, size etc) to receiver (which then stores it)
  2. when a matching receive call is posted, receiver notifies the sender that the data can be sent
  3. sender then transfers data

- advantages: only have to buffer meta-data (much more scalable), zero-copy possible (with RDMA)

- disadvantages: extra latency and complexity

**Short protocol**: data is small enough to be sent with the envelope (like eager)
Message Selection

- the sending process calls

```c
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
```

- `dest`: where to send message, e.g. process 0, ..., p – 1
- A programmer-determined message tag can be used to create classes of messages

- the receiving process calls

```c
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Status *status);
```

- receive a message from given process (including `MPI_ANY_SOURCE`)
- receive a message with given tag (including `MPI_ANY_TAG`)
- Buffer must be long enough for the incoming message (!)

- both have **blocking** semantics (send is either local or global)
consider a two-process MPI program, attempting send each other’s a array:

```c
char a[N]; int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// initialize a, using rank
MPI_Send(a, N, MPI_CHAR, 1-rank, 99, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_CHAR, 1-rank, 99, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE);
```

What would happen, if the send was \textbf{locally blocking}, \textbf{globally-blocking}, or \textbf{non-blocking}?

- in the sockets API, what semantics does `send()` have? (similarly for Unix pipes, with `write()`)

the previous two-process MPI program could be written as:

```c
char a[N]; int rank, msize; MPI_Status status;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// initialize a, using rank
MPI_Send(a, N, MPI_CHAR, 1-rank, 99, MPI_COMM_WORLD);
MPI_Recv(a, N, MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Get_count(&status, MPI_INT, &msize);
printf("received message from %d tag=%d size=%d\n",
    status(MPI_SOURCE), status(MPI_TAG), &msize);
```

- MPI matches received messages on the source rank and tag only
- actual received message size (msize, above) may be less than the specified count (N, above)
- if it is greater, an error will be raised
to enforce locally blocking sends, can use buffered send MPI_Bsend()

- the message is copied to an MPI buffer area
- by default this area is quite small; MPI_Buffer_attach()
- in general, problematic to use for large messages

non-blocking sends can be enforced by using:

MPI_Isend(..., MPI_Request *req) to initiate a send request
followed by an MPI_Wait(MPI_Request *req, MPI_Status *status) (wait till send completes)

both of the above can be used in conjunction with normal MPI_Recv()

also, we can specify a non-blocking receives,

MPI_Irecv(..., MPI_Request *req), similarly followed by an MPI_Wait()
Non-blocking Send/Recv Example

2 process example:

```c
int rank, otherRank;
const int TAG = 101;
MPI_Request reqSend, reqRecv;
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
// initiate communication
MPI_Isend(&rank, 1, MPI_INT, 1-rank, TAG, MPI_COMM_WORLD, &reqSend);
MPI_Irecv(&otherRank, 1, MPI_INT, 1-rank, TAG, MPI_COMM_WORLD, &reqRecv);
// do something useful here
...
// complete communication
MPI_Wait(&reqSend, MPI_STATUS_IGNORE);
MPI_Wait(&reqRecv, MPI_STATUS_IGNORE);
```
Hands-on Exercise: MPI Message Semantics
Outline

1. Introduction and Course Overview
2. Elements of Message Passing
3. Message Passing Semantics and Protocols
4. System Area Networks
5. Routing and Communication Costs
Overview: System Area Networks

- inevitably the performance of a single processor is limited by the clock speed
- improved manufacturing increases clock but ultimately limited by speed of light
- Instruction-Level Parallelism allows multiple ops at once, but is limited

It’s time to go parallel!

Overview:

- review: architectural classifications
- review: shared/distributed memory
- static/dynamic processor connectivity
- evaluating static networks
Architecture Classification: Flynn’s Taxonomy

- **why classify:**
  - what kind of parallelism is employed? Which architecture has the best prospect for the future? What has already been achieved by current architecture types? Reveal configurations that have not yet considered by system architect. Enable building of performance models.

Flynn’s **taxonomy** is based on the degree of parallelism, with 4 categories determined according to the number of instruction and data streams.

### Data Stream

<table>
<thead>
<tr>
<th>Instr’n Stream</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td></td>
<td>1CPU</td>
<td>Array/Vector Processor</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD</td>
<td>MIMD</td>
</tr>
<tr>
<td></td>
<td>(Pipelined?)</td>
<td>Multiple Processor</td>
</tr>
</tbody>
</table>
SIMD and MIMD

**SIMD: Single Instruction Multiple Data**
- also known as data parallel processors or array processors
- vector processors (to some extent)
- current examples include SSE instructions, SPEs on CellBE, GPUs
- NVIDIA’s SIMT ($T = \text{Threads}$) is a slight variation

**MIMD: Multiple Instruction Multiple Data**
- examples include quad-core PC, octa-core Xeons on Raijin
MIMD

Most successful parallel model
- more general purpose than SIMD (e.g. CM5 could emulate CM2)
- harder to program, as processors are not synchronized at the instruction level

Design issues for MIMD machines
- scheduling: efficient allocation of processors to tasks in a dynamic fashion
- synchronization: prevent processors accessing the same data simultaneously
- interconnect design: processor to memory and processor to processor interconnects. Also I/O network,
- overhead: e.g. from coordinating activities between processors
- partitioning: identifying parallelism in algorithms is non-trivial

(aside: **SPMD Single Program Multiple Data**: all processors run the same executable: more restrictive than MIMD)
each processor has local or private memory
interact solely by message passing
commonly known as **distributed memory parallel computers**
memory bandwidth scales with number of processors
example: between “nodes” on the NCI Raijin system
processors interact by modifying data objects stored in a shared address space

simplest solution is a flat or uniform memory access (UMA)

scalability of memory bandwidth and processor-processor communications (arising from cache line transfers) are problems

so is synchronizing access to shared data objects

example: dual/quad core PC (ignoring cache)
Dynamic Processor Connectivity: Crossbar

- is a non-blocking network, in that the connection of two processors does not block connections between other processors
- complexity grows as $O(p^2)$
- may be used to connect processors, each with their own local memory (typically encapsulated in a switch)

- may also be used to connect processors with various memory banks in shared memory systems
Dynamic Connectivity: Multi-staged Networks

- consist of $\log p$ stages, thus $p \log p$ total cost, where $p$ is the number of processors ($\log p = \log_2(p)$)
- for message source $s$ and destination $t$, at stage 1:
  - route through if most significant bits of $s$ and $t$ are the same
  - otherwise, crossover
- process repeats for next stage using the next most significant bit etc
Static Connectivity: Complete, Mesh, Tree

Completely connected (becomes very complex!)

Linear array/ring, mesh/2d torus

Tree (static if nodes are processors)
a multidimensional mesh with exactly two processors in each
dimension, i.e $p = 2^d$ where $d$ is the dimension of the hypercube

- advantages: $\leq d$ ‘hops’ between any processors
- disadvantage: the number of connections per processor is $d$
- can be generalized to $k$-ary $d$-cubes, e.g. a $4 \times 4$ torus is a 4-ary 2-cube
- examples: NCube, SGI Origin, Cray T3D, TOFU
two processors connected directly only if binary labels differ by one bit in a $d$-dim. hypercube, each processor directly connects to $d$ others 
a $d$-dimensional hypercube can be partitioned into two $d - 1$ sub-cubes etc 
the number of links in the shortest path between two processors is the Hamming distance between their labels 
the **Hamming distance** between two processors labeled $s$ and $t$ is the number of bits that are on in the binary representation of $s \oplus t$ where $\oplus$ is **bitwise exclusive or** (e.g. 3 for $101 \oplus 010$ and 2 for $011 \oplus 101$)
Diameter

- the maximum distance between any two processors in the network
- directly determines communication time (latency)

Connectivity

- the multiplicity of paths between any two processors
- a high connectivity is desirable as it minimizes contention (also enhances fault-tolerance)
- **arc connectivity** of the network: the minimum number of arcs that must be removed for the network to break it into two disconnected networks
  - 1 for linear arrays and binary trees
  - 2 for rings and 2D meshes
  - 4 for a 2D torus
  - $d$ for $d$-dimensional hypercubes
Channel width
- the number of bits that can be communicated simultaneously over a link connecting two processors

Bisection width and bandwidth
- **bisection width** is the minimum number of communication links that have to be removed to partition the network into two equal halves
- **bisection bandwidth** is the minimum volume of communication allowed between two halves of the network with equal numbers of processors

Cost
- many criteria can be used; we will use the number of communication links or wires required by the network
## Summary: Static Interconnection Metrics

<table>
<thead>
<tr>
<th>Network</th>
<th>Diameter</th>
<th>Bisection width</th>
<th>Arc connectivity</th>
<th>Cost (no. of links)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Completely-connected</td>
<td>1</td>
<td>$p^2/4$</td>
<td>$p - 1$</td>
<td>$p(p - 1)/2$</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>$2 \log((p + 1)/2)$</td>
<td>1</td>
<td>1</td>
<td>$p - 1$</td>
</tr>
<tr>
<td>Linear array</td>
<td>$p - 1$</td>
<td>1</td>
<td>1</td>
<td>$p - 1$</td>
</tr>
<tr>
<td>Ring</td>
<td>$\lfloor p/2 \rfloor$</td>
<td>2</td>
<td>2</td>
<td>$p$</td>
</tr>
<tr>
<td>2D Mesh</td>
<td>$2(\sqrt{p} - 1)$</td>
<td>$\sqrt{p}$</td>
<td>2</td>
<td>$2(p - \sqrt{p})$</td>
</tr>
<tr>
<td>2D Torus</td>
<td>$2\lfloor \sqrt{p}/2 \rfloor$</td>
<td>$2\sqrt{p}$</td>
<td>4</td>
<td>$2p$</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$\log p$</td>
<td>$p/2$</td>
<td>$\log p$</td>
<td>$(p \log p)/2$</td>
</tr>
</tbody>
</table>

**Note:** the Binary Tree suffers from a bottleneck: all traffic between the left and right sub-trees must pass through the root. The fat tree interconnect alleviates this:

- usually, only the leaf nodes contain a processor
- it can be easily ‘partitioned’ into sub-networks without degraded performance (useful for supercomputers)

**Discussion point:** which would you use for small, medium and large $p$? What values of $p$ would (roughly) be the cuttoff points?
Further Reading: Parallel Hardware

- The Free Lunch Is Over!
- Ch 1, 2.1-2.4 of Introduction to Parallel Computing
- Ch 1, 2 of Principles of Parallel Programming
- Lecture notes from Calvin Lin:
  A Success Story: ISA
  Parallel Architectures
Hands-on Exercise: MPI Message Performance
Routing and Communication Costs

Outline

1. Introduction and Course Overview
2. Elements of Message Passing
3. Message Passing Semantics and Protocols
4. System Area Networks
5. Routing and Communication Costs
Overview: Routing and Communication Costs

Optimizing communications is non-trivial! (Introduction to Parallel Computing, Grama et al)

- routing mechanisms and communication costs
- routing strategies: store-and-forward, cut-through
- communication patterns: algorithms and cost models
  - using store-and-forward and cut-through for:
    - one-all and all-all broadcasts on rings, meshes, trees and hypercubes
- effects of special hardware

Refs: Grama et al, Ch 2.5-,4
Routing mechanism: what path a message takes through the network

- **minimal, non-minimal**: minimal takes shortest path. Non-minimal may be used to avoid network congestion
- **deterministic routing**: unique path determined based solely on the source and destination processor (ignores state of network)
- **adaptive routing**: use information on the state of the network

Communication cost is made of:

- **start-up time** \((t_s)\): time required to handle a message at the sending processor (add header and trailer and execute the routing algorithm)
- **per-hop time** \((t_h)\): time taken for the header of the message to travel between two directly connected processors. Related to the latency in routing switch
- **per-word transfer time** \((t_w)\): dependent on channel bandwidth
Routing and Communication Costs

Store-and-Forward

- common on early parallel systems
- each intermediate processor in the path from the sending to receiving processor forwards the message AFTER entire message has been received and stored
- assuming a message of size $m$ traversing $l$ links

$$t_{\text{comm}} = t_s + (m \times t_w + t_h) \times l$$

- usually per-hop time $t_h$ is substantially less than $m \times t_w$ (typical switch latency is in the nsec range) so:

$$t_{\text{comm}} \approx t_s + m \times t_w \times l$$
Cut-Through Routing/Wormhole Routing

- message is sent in fixed-length segments called **flow-control digits** or **flits**
- **wormhole routing** **pipelines** the **flits** through the network
- uses less memory at the intermediate processor
- message header takes \( l \times t_h \) to arrive
- if message is \( m \) words long it will arrive in time \( m \times t_w \) after the arrival of the header

\[
t_{\text{comm}} = t_s + l \times t_h + m \times t_w
\]

- ignoring start-up, the cost to send a message is only \( O(m + l) \)
  compared to \( O(m \times l) \) for **store-and-forward routing**
- pioneered (mostly) by Bill Dally for the Torus Routing chip (1986),
  with ‘milestone papers’ on **deadlock avoidance** (1987) and **virtual channels** (1992)
Pipelining of Flow Control Digits (flits)

Store and Forward Network

Two Flits

Four Flits
Deadlock

- **deadlock** arises when no message can progress in the network
- flit routing techniques are designed to prevent deadlock
  - note: this does not prevent user programmed deadlocks!
Communication Costs on Static Networks

- for a single message transfer:
  - **store-and-forward**: $t_{\text{comm}} = t_s + (m \times t_w + t_h) \times l \approx t_s + m \times t_w \times l$
  - **cut-through**: $t_{\text{comm}} = t_s + l \times t_h + m \times t_w$

Discussion point: how valid is network diameter a measure of performance under CT? Does this affect which you would network for small, medium and large $p$? Also, what would you expect the relative size of $t_s$ be over $t_h$? (consider a TCP/IP message).

- how can we extend this to derive cost models for communication patterns?, e.g.

  ![One-All Broadcast](image1)

  ![All-All Broadcast](image2)

- a one-to-all broadcast is closely related to a **all-to-one reduction**
Routing and Communication Costs

SF/One-All/Ring

- assuming that each processor can only send a single message:

- case 1 ‘costs’ 7 sends
- case 2 ‘costs’ 4 sends

\[ t_{\text{one-all}} = (t_s + m \times t_w) \left\lceil \frac{p}{2} \right\rceil \]
Routing and Communication Costs

SF/One-All/Hypercube

- start by sending along highest dimension of hypercube (dimension specified by the most significant bit in the binary representation of the processor label)
- then continue along successively lower dimensions

\[ t_{\text{one-all}} = \log p \times (t_s + m \times t_w) \]
CT/One-All/Ring

- try mapping the hypercube algorithm to a ring
  - algorithm sends message to non-nearest neighbours
  - mapping useful as complexity of passing a message of size $m$ between processors separated by $l$ lines is at most $O(m + l)$

- all messages are sent in the same direction
- at each step, the distance of communication halves while the number of processors communicating doubles

- cost is:

$$t_{\text{one-all}} = \sum_{i=1}^{\lfloor p/2 \rfloor} (t_s + m \times t_w + t_h \times p/2^i)$$

$$= (t_s + m \times t_w) \log p + t_h (p - 1)$$

- compared to SF: $(t_s + m \times t_w) \lfloor p/2 \rfloor$, i.e. CT routing reduces time by $p/(2 \log p)$
as for ring except in 2D. Recall the cost along a row (or column) is:

\[ t_{\text{row}} = (t_s + m \times t_w) \log \sqrt{p} + t_h (\sqrt{p} - 1) \]

hence, for the entire broadcast:

\[ t_{\text{one-all}} = (t_s + m \times t_w) \log p + 2t_h (\sqrt{p} - 1) \]
Routing and Communication Costs

CT/One-All/Tree and Hypercube

Balanced binary tree (assuming same $t_h$ between switches and processors):

$$t_{\text{one-all}} = (t_s + m \times t_w + t_h(\lg p + 1)) \lg p$$

Hypercube: no merit from CT routing since communication is always to nearest neighbour
**Routing and Communication Costs**

### SF/All-All/Ring

- perform $p$ one-all broadcasts
  - *but* total time will scale as $p \times t_{\text{one-all}}$
- use links more efficiently to perform all $p$ one-all broadcasts at the same time

![Diagram showing the network for SF/All-All/Ring with numbers and arrows indicating the routes of one-all broadcasts.]

- total cost:
  
  $$t_{\text{all-all}} = (t_s + m \times t_w) \times (p - 1)$$

**etc**
Routing and Communication Costs

SF/All-All/Mesh

- each row performs an all-all broadcast within the row (or column)
  \[ t_{\text{row}}^{\text{all-all}} = (t_s + m \times t_w)(\sqrt{p} - 1) \]
- each processor collects the \( \sqrt{p} \) messages they have received from the other processors in that row, and consolidates them into a single message of length \( \sqrt{p} \times m \)
- each column performs an all-all broadcast within the column

\[
 t_{\text{all-all}} = (t_s + \sqrt{p} \times m \times t_w)(\sqrt{p} - 1) + (t_s + m \times t_w)(\sqrt{p} - 1) = 2t_s(\sqrt{p} - 1) + m \times t_w(p - 1)
\]
SF/All-All/Hypercube

- extension of the mesh algorithm
- in each step, pairs exchange data and the message length doubles

\[
\begin{align*}
t_{\text{all-all}} &= \sum_{i=1}^{\lfloor \log p \rfloor} (t_s + 2^{i-1} m \times t_w) \\
&= t_s \lfloor \log p \rfloor + m \times t_w(p - 1)
\end{align*}
\]
Routing and Communication Costs

**CT/All-All/Ring**

- previously we saw an advantage from CT routing as the number of messages sent doubles at each stage
- not so for all-all broadcast as it gives rise to network contention

![Diagram of network contention](image URL)
Routing and Communication Costs

Routing Messages in Parts: SF on Hypercube

- the longest time to send any message is (noting the max. number of hops is $\lg p$)
  \[ t_{\text{comm}} = t_s + t_w \times m \times \lg p \]
- there are $\lg p$ distinct paths between any pair of processors (i.e. $\lg p$ wires)
- split message into $\lg p$ parts and send along the longest path first
- process will complete in $2 \lg p$ steps

\[
  t_{\text{comm}} = 2 \lg p (t_s + t_w \times m / \lg p) \\
  = 2 (t_s \times \lg p + t_w \times m)
\]

- $t_s$ time has increased by $2 \lg p$
- $t_w$ time has been reduced by $2 / \lg p$

Method of choice will depend on the message length
Special Hardware

**All-port communication:**
- some machines have hardware that will send messages in multiple directions at the same time

**Challenge:** is there some hardware extension to CT that would enable (say) a one-all broadcast on a ring to complete in $t_s + mt_w + (p - 1)t_h$ (as opposed to $(\lg p(t_s + mt_w) + (p - 1)t_h)$?

**Special networks:**
- some machines had special hardware to enable synchronization, broadcast operations to be performed very fast, e.g. IBM BlueGene Systems
Summary of Broadcast Communication Times

- assuming:
  - CT routing
  - one-port communications

<table>
<thead>
<tr>
<th>operation</th>
<th>ring</th>
<th>2D mesh (wraparround, square)</th>
<th>hypercube</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-to-all</td>
<td>((t_s + t_w m) \lg p)</td>
<td>((t_s + t_w m) \lg p)</td>
<td>((t_s + t_w m) \lg p)</td>
</tr>
<tr>
<td></td>
<td>(+ t_h(p - 1))</td>
<td>(+2t_h(\sqrt{p} - 1))</td>
<td></td>
</tr>
<tr>
<td>All-to-all</td>
<td>((t_s + t_w m)(p - 1))</td>
<td>(2t_s(\sqrt{p} - 1))</td>
<td>(t_s \lg p)</td>
</tr>
<tr>
<td></td>
<td>(+ t_w m(p - 1))</td>
<td>(+t_w m(p - 1))</td>
<td></td>
</tr>
</tbody>
</table>
Summary

Topics covered today:

- what is the distributed memory HPC paradigm and why we use it
- basics of message passing: process creation and elementary send/receive operations
- various messaging semantics, their underlying protocols and MPI APIs
- system area networks: topologies and their performance
- routing strategies, and costs of communication patterns on various topologies

Tomorrow - a deeper look at message passing!
Hands-on Exercise: Simulating Routing Protocols