CS 5220: Distributed Memory Programming

David Bindel

2017-09-19

Plan for this week

- This week: distributed memory programming
  - Distributed memory HW issues (topologies, cost models)
  - Message-passing programming concepts (and MPI)
  - Some simple examples
- Next week: shared memory programming
  - Shared memory HW issues (cache coherence)
  - Threaded programming concepts (pthreads and OpenMP)
  - A simple example (Monte Carlo)

Basic questions

How much does a message cost?

- *Latency*: time to get between processors
- *Bandwidth*: data transferred per unit time
- How does *contention* affect communication?

This is a combined hardware-software question!

We want to understand just enough for reasonable modeling.

Thinking about interconnects

Several features characterize an interconnect:

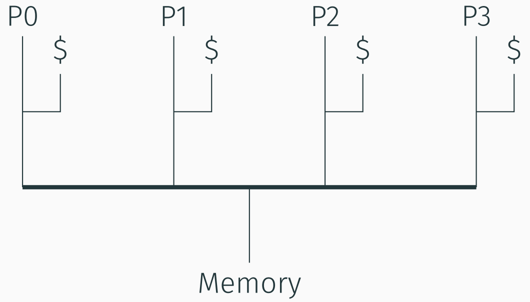
- *Topology*: who do the wires connect?
- *Routing*: how do we get from A to B?
- *Switching*: circuits, store-and-forward?
- *Flow control*: how do we manage limited resources?

Thinking about interconnects

- Links are like streets
- Switches are like intersections
- Hops are like blocks traveled
- Routing algorithm is like a travel plan
- Stop lights are like flow control
- Short packets are like cars, long ones like buses?

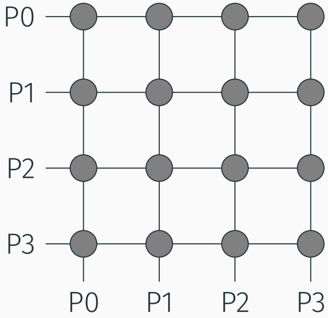
At some point the analogy breaks down...

Bus topology



- One set of wires (the bus)
- Only one processor allowed at any given time
  - *Contention* for the bus is an issue
- Example: basic Ethernet, some SMPs

Crossbar



- Dedicated path from every input to every output
    - Takes $O(p^2)$ switches and wires!
- Example: recent AMD/Intel multicore chips
  (older: front-side bus)

Bus vs. crossbar

- Crossbar: more hardware
- Bus: more contention (less capacity?)
- Generally seek happy medium
    - Less contention than bus
    - Less hardware than crossbar
    - May give up one-hop routing

Network properties

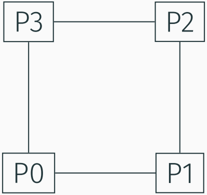
Think about latency and bandwidth via two quantities:

- *Diameter*: max distance between nodes
- *Bisection bandwidth*: smallest bandwidth cut to bisect
    - Particularly important for all-to-all communication

Linear topology

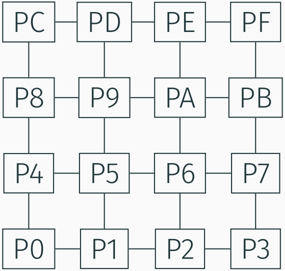
- $p - 1$ links
- Diameter $p - 1$
- Bisection bandwidth 1

Ring topology



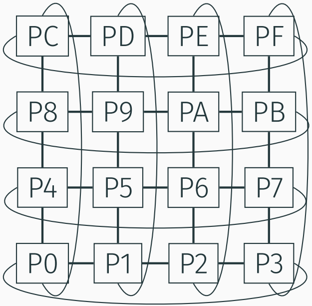
- p links
- Diameter $p/2$
- Bisection bandwidth 2

Mesh



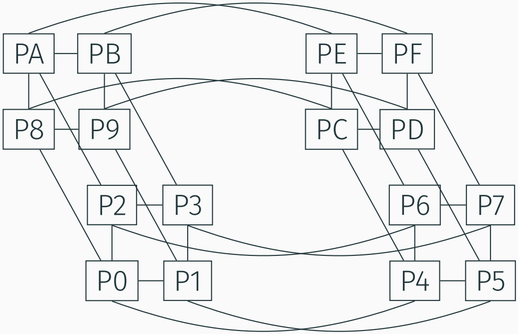
- May be more than two dimensions
- Route along each dimension in turn

Torus



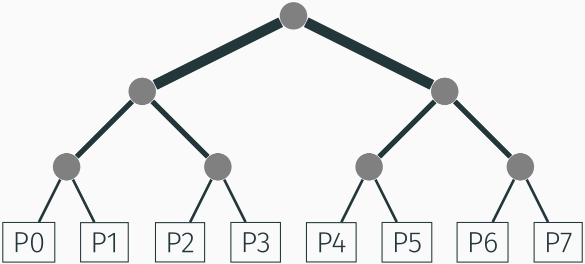
Torus : Mesh :: Ring : Linear

Hypercube



- Label processors with binary numbers
- Connect p_1 to p_2 if labels differ in one bit

Fat tree



- Processors at leaves
- Increase link bandwidth near root

Others...

- Butterfly network
- Omega network
- Cayley graph

Current picture

- Old: latencies = hops
- New: roughly constant latency (?)
    - Wormhole routing (or cut-through) flattens latencies vs store-forward at hardware level
    - Software stack dominates HW latency!
    - Latencies *not* same between networks (in box vs across)
    - May also have store-forward at library level

- Old: mapping algorithms to topologies
- New: avoid topology-specific optimization
    - Want code that runs on next year's machine, too!
    - Bundle topology awareness in vendor MPI libraries?
    - Sometimes specify a *software* topology

α-β model

Crudest model: $t_{\text{comm}} = \alpha + \beta M$

- t_{comm} = communication time
- α = latency
- β = inverse bandwidth
- M = message size

Works pretty well for basic guidance!

Typically $\alpha \gg \beta \gg t_{\text{flop}}$. More money on network, lower α.

LogP model

Like α-β, but includes CPU time on send/recv:

- Latency: the usual
- Overhead: CPU time to send/recv
- Gap: min time between send/recv
- P: number of processors

Assumes small messages (gap $\sim$ bw for fixed message size).

Communication costs

Some basic goals:

- Prefer larger to smaller messages (avoid latency)
- Avoid communication when possible
  - Great speedup for Monte Carlo and other embarrassingly parallel codes!
- Overlap communication with computation
  - Models tell you how much computation is needed to mask communication costs.

Message passing programming

Basic operations:

- Pairwise messaging: send/receive
- Collective messaging: broadcast, scatter/gather
- Collective computation: parallel prefix (sum, max, ...)
- Barriers (no need for locks!)
- Environmental inquiries (who am I? do I have mail?)

(Much of what follows is adapted from Bill Gropp's material.)

MPI

- Message Passing Interface
- An interface spec — many implementations
- Bindings to C, C++, Fortran

Hello world

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello from %d of %d\n", rank, size);
    MPI_Finalize();
    return 0;
}
```

Communicators

- Processes form *groups*
- Messages sent in *contexts*
    - Separate communication for libraries
- Group + context = communicator
- Identify process by rank in group
- Default is `MPI_COMM_WORLD`

Sending and receiving

Need to specify:

- What's the data?
  - Different machines use different encodings (e.g. endian-ness)
  - $\implies$ "bag o' bytes" model is inadequate
- How do we identify processes?
- How does receiver identify messages?
- What does it mean to "complete" a send/recv?

MPI datatypes

Message is (address, count, datatype). Allow:

- Basic types (`MPI_INT`, `MPI_DOUBLE`)
- Contiguous arrays
- Strided arrays
- Indexed arrays
- Arbitrary structures

Complex data types may hurt performance?

MPI tags

Use an integer *tag* to label messages

- Help distinguish different message types
- Can screen messages with wrong tag
- `MPI_ANY_TAG` is a wildcard

MPI Send/Recv

Basic blocking point-to-point communication:

```
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);
```

MPI send/recv semantics

- Send returns when data gets to *system*
    - … might not yet arrive at destination!
- Recv ignores messages that don't match source and tag
    - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` are wildcards
- Recv `status` contains more info (tag, source, size)

Ping-pong pseudocode

Process 0:

```
for i = 1:ntrials
  send b bytes to 1
  recv b bytes from 1
end
```

Process 1:

```
for i = 1:ntrials
  recv b bytes from 0
  send b bytes to 0
end
```

Ping-pong MPI

```
void ping(char* buf, int n, int ntrials, int p)
{
    for (int i = 0; i < ntrials; ++i) {
        MPI_Send(buf, n, MPI_CHAR, p, 0,
                 MPI_COMM_WORLD);
        MPI_Recv(buf, n, MPI_CHAR, p, 0,
                 MPI_COMM_WORLD, NULL);
    }
}
```

(Pong is similar)

Ping-pong MPI

```
for (int sz = 1; sz <= MAX_SZ; sz += 1000) {
    if (rank == 0) {
        clock_t t1, t2;
        t1 = clock();
        ping(buf, sz, NTRIALS, 1);
        t2 = clock();
        printf("%d %g\n", sz,
               (double) (t2-t1)/CLOCKS_PER_SEC);
    } else if (rank == 1) {
        pong(buf, sz, NTRIALS, 0);
    }
}
```

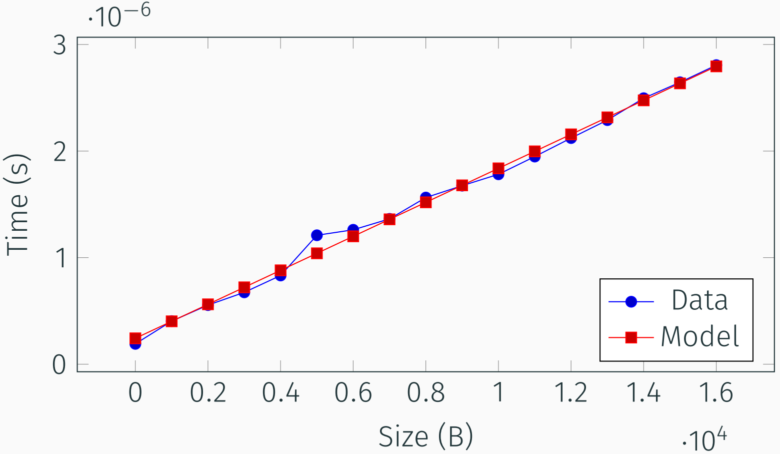
Running the code

On my laptop (OpenMPI)

```
mpicc -std=c99 pingpong.c -o pingpong.x
mpirun -np 2 ./pingpong.x
```

Details vary, but this is pretty normal.

Approximate α-β parameters (2-core laptop)



$\alpha \approx$ `2.43e-07`, $\beta \approx$ `1.59e-10`

Where we are now

Can write a lot of MPI code with 6 operations we've seen:

- `MPI_Init`
- `MPI_Finalize`
- `MPI_Comm_size`
- `MPI_Comm_rank`
- `MPI_Send`
- `MPI_Recv`

… but there are sometimes better ways.

Next time: non-blocking and collective operations!