CS 5220

Distributed memory

David Bindel

2024-09-24

Logistics

Matrix multiply

- Deadline extended to Oct 1
- Please use `c4-standard-2` instances
- I also recommend the Intel compilers
- MKL gets 80-100 GFlop/s

Scoring

- Performance: Linearly interpolate
    - 10 GFlop/s = 0 points (baseline)
    - 60 GFlop/s = 5 points (max)
- Writeup (5 points)
    - Describe strategy
    - What you tried
    - What worked or didn’t work
    - Analysis of improvements
    - Performance plots

Advice

- Intel compiler vectorization guidelines
- ICX align: `__declspec(align(16, 8) static double Abuf[BUF_SIZE];`
- Report with `-qopt-report`
- OK with `-fp-model fast=2`
- Recommend `-march=emeraldrapids`

More advice

- Start with a small kernel working on aligned memory
- Get advice from the compiler (`-qopt-report`)
- Build bigger matmul by copying a tile in, doing kernel matmuls, and accumulating result out
- Build timing harnesses for things
- Use Intel Advisor and any other tools you can find!

HW1

- Also due Oct 1
- Main point: get to know Perlmutter!
- Also write just a little MPI (telephone)
- This is an *individual* assignment

Distributed memory

Plan for this week

- This week: distributed memory
    - HW issues (topologies, cost models)
    - Message-passing concepts and MPI
    - Some simple examples
- Next week: shared memory

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:

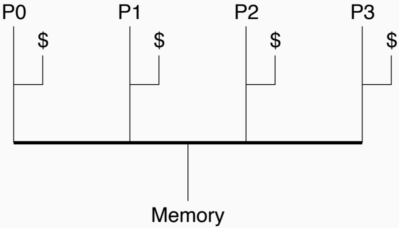
- *Topology*: who do the wires connect?
- *Routing*: how do we get from A to B?
- *Switching*: circuits, store-and-forward?
- *Flow control*: how do we manage limited resources?

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?

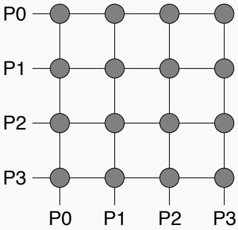
At some point the analogy breaks down...

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

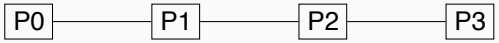


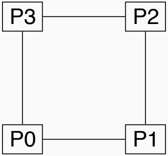
- Dedicated path from every input to every output
    - Takes $O(p^2)$ switches and wires!
- Example: recent AMD/Intel multicore chips
  (older: front-side bus)

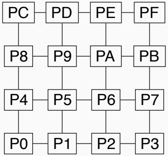
Bus vs. crossbar

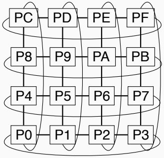
- Crossbar: more hardware
- Bus: more contention (less capacity?)
- Generally seek happy medium
    - Less contention than bus
    - Less hardware than crossbar
    - May give up one-hop routing

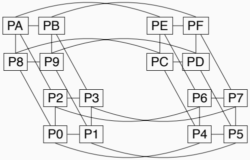
Other topologies

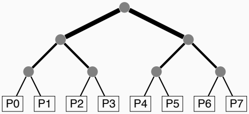












Network properties

Think about latency and bandwidth via two quantities:

- *Diameter*: max distance between nodes
    - Latency depends on distance (weakly?)
- *Bisection bandwidth*: smallest BW cut to bisect
    - Particularly key for all-to-all comm

MANY networks

In a typical cluster

- Ethernet, Infiniband, Myrinet
- Buses within boxes?
- Something between cores?

All with different behaviors.

Modeling picture

- DO distinguish different networks
- Otherwise, want simple perf models
    - Hockney model (α-β)
    - LogP and company
    - And many others

α-β model (Hockney 94)

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

- Communication time t_{comm}
- Latency α
- Inverse bandwidth β
- Message size M

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 $\sim \beta$ for fixed message size).

And many others

Recent survey lists 25 models!

- More complexity, more parameters
- Most miss some things (see Box quote)
- Still useful for guidance!
- Needs to go with experiments

Ping-pong

Process 0:

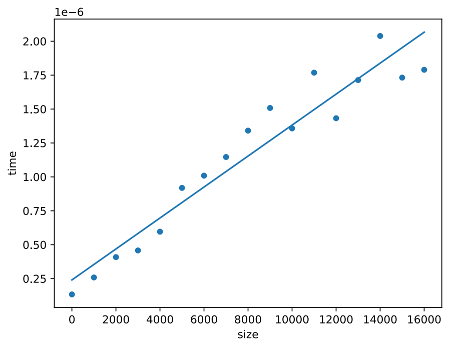
```
for i = 1:ntrials
  send b bytes to 1
  recv b bytes from 1
end
```

Process 1:

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

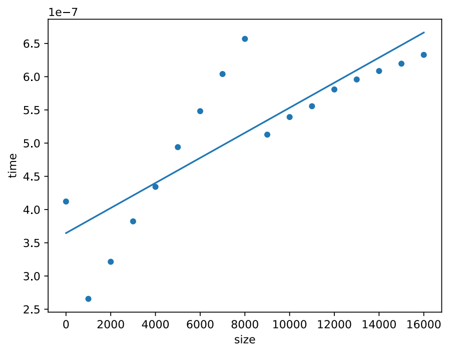
Laptop ping-pong times

$\alpha = 0.240$ microseconds; $\beta = 0.141$ s/GB



Perlmutter ping-pong times

$\alpha = 0.365$ microseconds; $\beta = 0.0189$ s/GB



This is on one node.

Takeaways

- Prefer larger to smaller messages (amortize latency, overhead)
- More care with slower networks
- Avoid communication when possible
    - Great speedup for Monte Carlo and other embarrassingly parallel codes!
- Overlap communication with computation
    - Models tell roughly computation to mask comm
    - Really want to measure, though

MPI programming

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.)

MPI

- Message Passing Interface
- An interface spec — many implementations (OpenMPI, MPICH, MVAPICH, Intel, …)
- Single Program Multiple Data (SPMD) style
- Bindings to C, C++, Fortran

MPI

- Versions
    - 1.0 in 1994
    - 2.2 in 2009
    - 3.0 in 2012
    - 4.1 in 2023
    - 4.2, 5.0 are pending
- Will mostly stick to MPI-2 today

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;
}
```

Building, queueing, running

Several steps to actually run

```
cc -o foo.x foo.c      # Compile the program
sbatch foo.sub         # Submit to queue (SLURM)
# srun -n 2 ./foo.x    # (in foo.sub) Run on 2 procs
```

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 mismatch source/tag
    - `MPI_ANY_SOURCE` and `MPI_ANY_TAG` 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

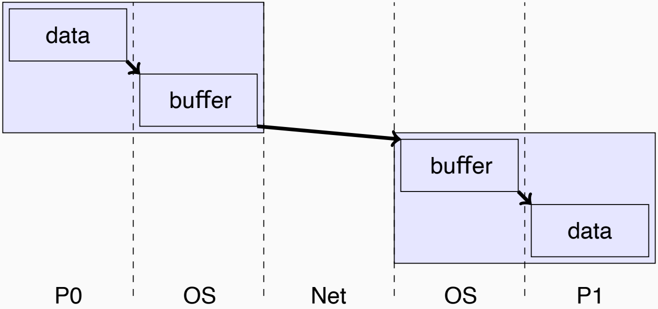
```
void ping(char* buf, int n, int ntrials, int p)
{
    for (int i = 0; i < ntrials; ++i) {
        MPI_Send(buf, n, MPI_CHAR, p, 0,
                 MPI_COMM_WORLD);
        MPI_Recv(buf, n, MPI_CHAR, p, 0,
                 MPI_COMM_WORLD, NULL);
    }
}
```

(Pong is similar)

Ping-pong MPI

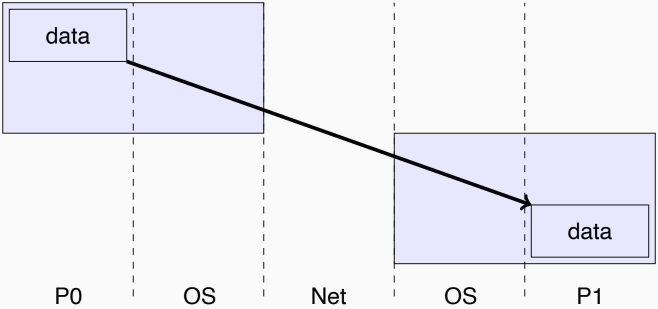
```
for (int sz = 1; sz <= MAX_SZ; sz += 1000) {
    if (rank == 0) {
        double t1 = MPI_Wtime();
        ping(buf, sz, NTRIALS, 1);
        double t2 = MPI_Wtime();
        printf("%d %g\n", sz, t2-t1);
    } else if (rank == 1) {
        pong(buf, sz, NTRIALS, 0);
    }
}
```

Blocking and buffering



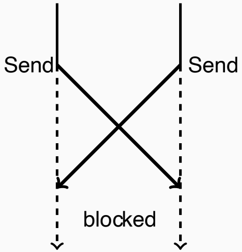
Block until data “in system” — maybe in a buffer?

Blocking and buffering



Alternative: don't copy, block until done.

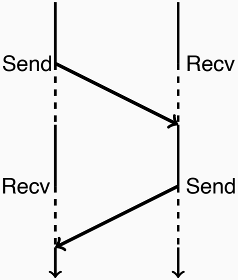
Problem 1: Potential deadlock



Both processors wait to send before they receive!

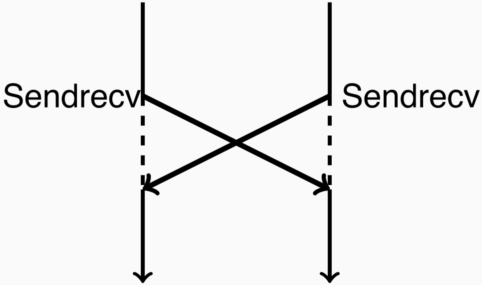
May not happen if lots of buffering on both sides.

Solution 1: Alternating order



Could alternate who sends and who receives.

Solution 2: Combined send/recv



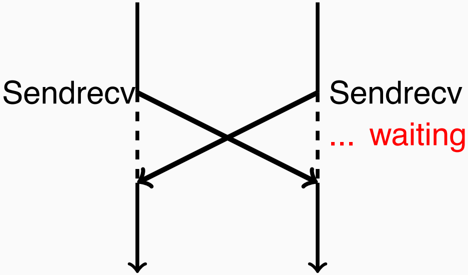
Common operations deserve explicit support!

Combined sendrecv

```
MPI_Sendrecv(sendbuf, sendcount, sendtype,
             dest, sendtag,
             recvbuf, recvcount, recvtype,
             source, recvtag,
             comm, status);
```

Blocking operation, combines send and recv to avoid deadlock.

Problem 2: Communication overhead



Partial solution: nonblocking communication

Blocking vs non-blocking

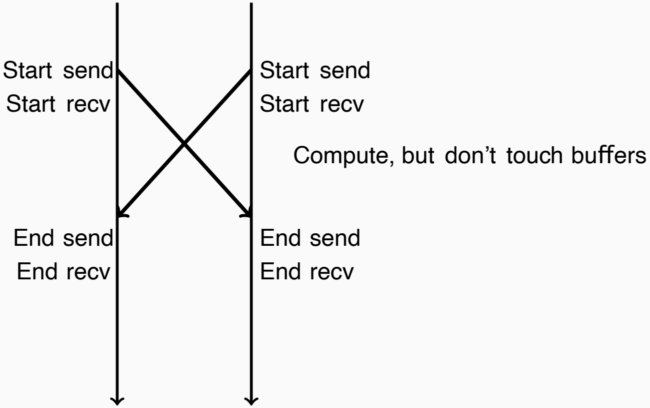
- `MPI_Send` and `MPI_Recv` are *blocking*
  - Send does not return until data is in system
  - Recv does not return until data is ready
  - Cons: possible deadlock, time wasted waiting
- Why blocking?
  - Overwrite buffer during send $\implies$ evil!
  - Read buffer before data ready $\implies$ evil!

Blocking vs non-blocking

Alternative: *nonblocking* communication

- Split into distinct initiation/completion phases
- Initiate send/recv and promise not to touch buffer
- Check later for operation completion

Overlap communication and computation



Start send

Start recv

Start send

Start recv

Compute, but don’t touch buffers

End send

End recv

End send

End recv

Nonblocking operations

Initiate message:

```
MPI_Isend(start, count, datatype, dest
          tag, comm, request);
MPI_Irecv(start, count, datatype, dest
          tag, comm, request);
```

Wait for message completion:

```
MPI_Wait(request, status);
```

Test for message completion:

```
MPI_Test(request, status);
```

Multiple outstanding requests

Sometimes useful to have multiple outstanding messages:

```
MPI_Waitall(count, requests, statuses);
MPI_Waitany(count, requests, index, status);
MPI_Waitsome(count, requests, indices, statuses);
```

Multiple versions of test as well.

Other send/recv variants

Other variants of `MPI_Send`

- `MPI_Ssend` (synchronous) – do not complete until receive has begun
- `MPI_Bsend` (buffered) – user provides buffer (via `MPI_Buffer_attach`)
- `MPI_Rsend` (ready) – user guarantees receive has already been posted
- Can combine modes (e.g. `MPI_Issend`)

`MPI_Recv` receives anything.

Another approach

- Send/recv is one-to-one communication
- An alternative is one-to-many (and vice-versa):
    - *Broadcast* to distribute data from one process
    - *Reduce* to combine data from all processors
    - Operations are called by all processes in communicator

Broadcast and reduce

```
MPI_Bcast(buffer, count, datatype,
          root, comm);
MPI_Reduce(sendbuf, recvbuf, count, datatype,
           op, root, comm);
```

- buffer is copied from root to others
- recvbuf receives result only at root
- op ∈ { `MPI_MAX`, `MPI_SUM`, ...}

Example: basic Monte Carlo

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char** argv) {
    int nproc, myid, ntrials = atoi(argv[1]);
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Bcast(&ntrials, 1, MPI_INT,
              0, MPI_COMM_WORLD);
    run_mc(myid, nproc, ntrials);
    MPI_Finalize();
    return 0;
}
```

Example: basic Monte Carlo

Let `sum[0]` = $\sum_i X_i$ and `sum[1]` = $\sum_i X_i^2$.

```
void run_mc(int myid, int nproc, int ntrials) {
    double sums[2] = {0,0};
    double my_sums[2] = {0,0};
    /* ... run ntrials local experiments ... */
    MPI_Reduce(my_sums, sums, 2, MPI_DOUBLE,
               MPI_SUM, 0, MPI_COMM_WORLD);
    if (myid == 0) {
        int N = nproc*ntrials;
        double EX = sums[0]/N;
        double EX2 = sums[1]/N;
        printf("Mean: %g; err: %g\n",
               EX, sqrt((EX*EX-EX2)/N));
    }
}
```

Collective operations

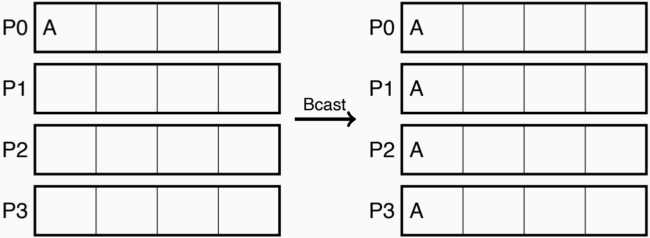
- Involve all processes in communicator
- Basic classes:
    - Synchronization (e.g. barrier)
    - Data movement (e.g. broadcast)
    - Computation (e.g. reduce)

Barrier

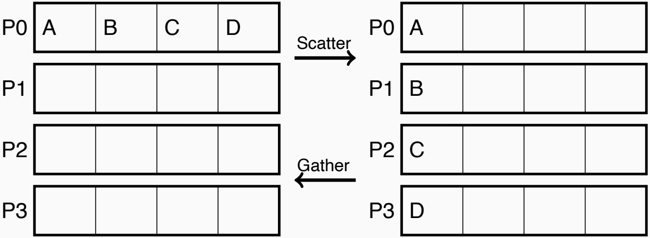
```
MPI_Barrier(comm);
```

Not much more to say. Not needed that often.

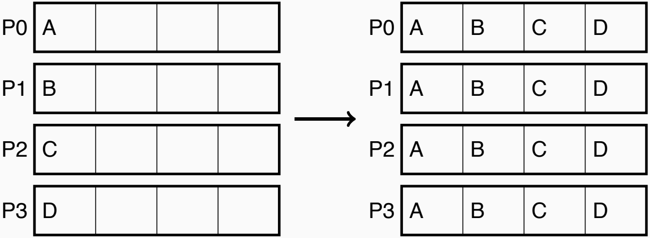
Broadcast



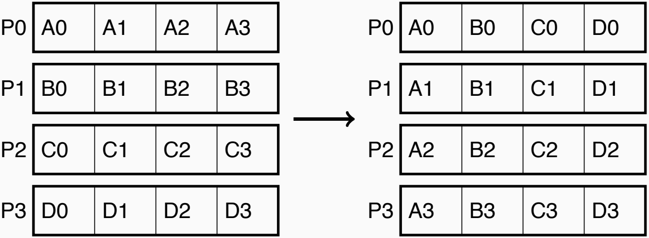
Scatter/gather



Allgather



Alltoall

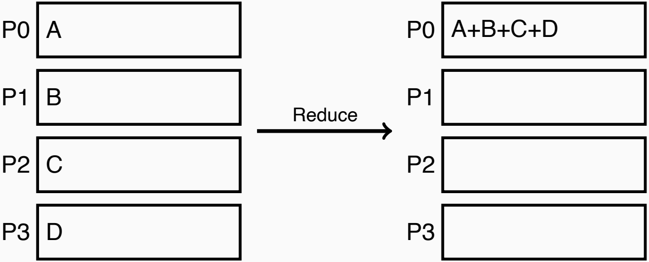


P0	A0	A1	A2	A3
P1	B0	B1	B2	B3
P2	C0	C1	C2	C3
P3	D0	D1	D2	D3

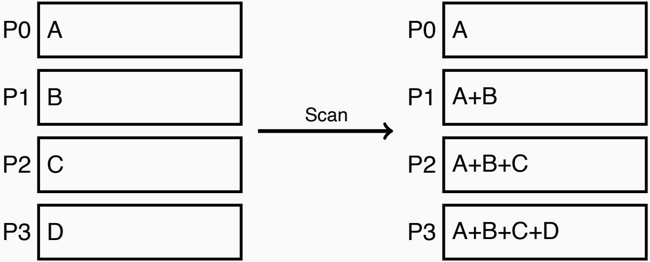
→

P0	A0	B0	C0	D0
P1	A1	B1	C1	D1
P2	A2	B2	C2	D2
P3	A3	B3	C3	D3

Reduce



Scan



P0 A

P1 B

P2 C

P3 D

Scan

P0 A

P1 A+B

P2 A+B+C

P3 A+B+C+D

The kitchen sink

- In addition to above, have vector variants (v suffix), more All variants (`Allreduce`), `Reduce_scatter`, ...
- MPI3 adds one-sided communication (put/get)
- MPI is *not* a small library!
- But a small number of calls goes a long way
    - `Init`/`Finalize`
    - `Get_comm_rank`, `Get_comm_size`
    - `Send`/`Recv` variants and `Wait`
    - `Allreduce`, `Allgather`, `Bcast`