NTNU – Trondheim
Norwegian University of Science and Technology

Timing, scatter, and gather

Jan.Christian.Meyer@ntnu.no

Today’s topic

- We’ve talked a lot about processors, networks, and operations, and called them “fast” or “slow”
- How long do they actually take?
- The only way to find out precisely is, sadly, to run them and see
- We can make some educated guesstimates, though

The precise way: run it and see

- Unsurprisingly, MPI has a clock
- It’s one of the very few functions that responds with a return value that isn’t an error code:

```
double MPI_Wtime( void );
```

- The answer is some number of seconds, represented as a double-precision floating point value
- The ‘W’ is short for *walltime*, which means it measures how much real time passes, regardless of
  - Whether it’s spent on your program or not,
  - Whether it’s spent in system calls, libraries, or your own expressions
  - Whether it’s spent by 1 or 1000 ranks
  - *Etc.*
- It’s meant to be like a clock on the wall that everyone can see

Timing in a single rank

- There's no MPI requirement for what calendar year, time zone, country, or parallel universe the clock is relative to
  - It's just some number of seconds
- That's ok, because we mainly want to measure differences in it:

```
double t_start = MPI_Wtime();
do_something_useful();
double t_end = MPI_Wtime();
printf ( “Something useful took %ld seconds!\n”, t_end – t_start );
```

- Hey, presto!

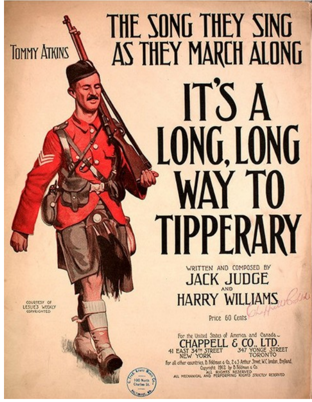
Timing with many ranks

- Program stages with communication in them are subject to ranks waiting somewhat unpredictably long for each other
  - Some may have been held up previously, and arrive late to the stage you're timing
- In order to isolate that your timings are only affected by the operations in the section you want to time, synchronize the ranks first:

```
MPI_Barrier ( MPI_COMM_WORLD );
double t_start = MPI_Wtime();
do_something_useful();
double t_end = MPI_Wtime();
printf (
    "Something useful took %ld seconds on rank %d!\n", t_end – t_start, rank
);
```

- You get *P* different timings still, but you can collect them, find the average, variance, median, *etc. etc.* and figure out how long things take.

Theoretical guesstimates



- Suppose we are posting letters in the mail instead of sending bytes across wires
- A tiny postcard will take some amount of time to get from here to Tipperary (or wherever)
- A large box will take a similar amount of time, even if you can put more stuff in it
- This interval is connected to the distance from A to B, rather than the message
- Let's call it *latency*, and write α

Postcards vs. boxes

- The difference between the postcard and the box is how much stuff gets moved
- Packing and unpacking the box takes additional time, and it's additional labor for whoever is transporting it
- Network capacity is usually measured in some multiple of [bytes / second], we call it *bandwidth* and write β
- Equally interesting from a message passing perspective, is the *inverse bandwidth* β^{-1}, measured in [seconds/byte]
- That is, how much transfer time do we add by sending additional bytes

Approximate communication time

- When we know the size *n* of our message, we can estimate the transmission time as the sum of latency and n times the inverse bandwidth:

  $$T_{comm}(n) = \alpha + n\,\beta^{-1}$$

- Because of the analogy with the mail system, this estimate is sometimes called the *"postal model"*
- I call it the *Hockney model*, because it was first published by one Roger W. Hockney
- Still others call it the *pingpong* model, for reasons that will imminently be made clear

Hockney's equipment

- Roger developed his model in order to estimate message costs on the Intel Paragon machine
  - The computer museum here at NTNU still has one
  - It doesn't run any more
- Communication links were equally fast throughout the entire machine
- Therefore, the α and β^{-1} could be measured between any pair of processors, and characterize the whole contraption

Hockney's experiment

- The *ping-pong* test of communication speed goes as follows:
  - Start the clock
  - Repeat "a lot of" times:
    - Send message from A to B (ping)
    - Send message from B to A (pong)
  - Stop the clock
  - Divide the time difference by 2 (for both directions), and the number of messages
- The "lot of" times have to be adjusted to whatever makes the procedure last long enough that you can reliably time it
  - That depends on the speed of the equipment you're using

Extracting α and β^{-1}

- In order to find the latency, we can do the ping-pong test with a massive number of either empty or 1-byte messages
  - This way, latency will dominate the time taken
  - 1-byte messages are only necessary if your machine skips empty messages
- In order to find the inverse bandwidth, we can do the ping-pong test with a smaller number of huge messages
  - This way, bandwidth requirements will dominate the time taken
  - Your choice of “huge” should reflect how many layers of the memory hierarchy you want the procedure to account for

In modern times

- The days of uniform latency and bandwidth are long gone
  - The cost of sending messages between adjacent cores on a chip is wildly different from the cost of sending them to another computer across the room
- If you want to make sense of ping-pong results nowadays, you have to measure as many different α/β pairs as you have types of links in your platform
- It can still be useful, though, if you are careful about where your ranks are running

(There are also a couple of statistical techniques to make the measurements more stable and reliable, but I won't bother you with them in TDT4200)

Latency lags bandwidth

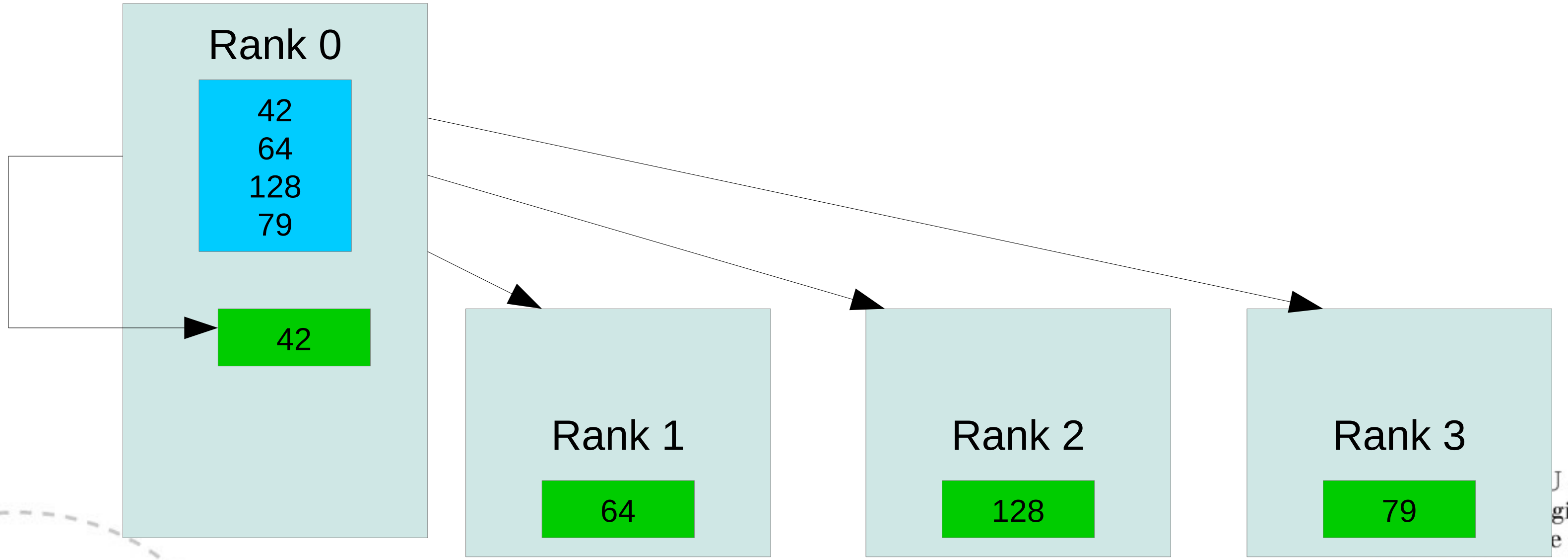
- Latency is often the smaller part of transmission time
- It is, however, very difficult to improve upon:
  - Bandwidth can be expanded by adding extra lanes to the interconnect fabric
  - Latency is ultimately restricted by the speed of light, nothing can go faster from A to B
- Research in parallel computing is eagerly investigating *latency-masking techniques*
  - We can’t get rid of it, but we can do something useful in the meantime
  - Overlapping computation with MPI_Isend is one such technique

Back to the MPI stuff

- Out of the collective operations, we only looked at barrier, broadcast and reduction
- I won’t go through all of them (they’re in the documentation), but two more are in common use:
- MPI_Scatter takes a huge lump of data on one rank and distributes parts of it around
- MPI_Gather collects distributed parts into a huge lump of data on one rank

MPI_Scatter

- This is another rooted collective, like Bcast and Reduce
- I've illustrated it with 0 as the root
- Note that the root also gets a rank-sized piece of the data, even though it already has a copy



Scatter arguments

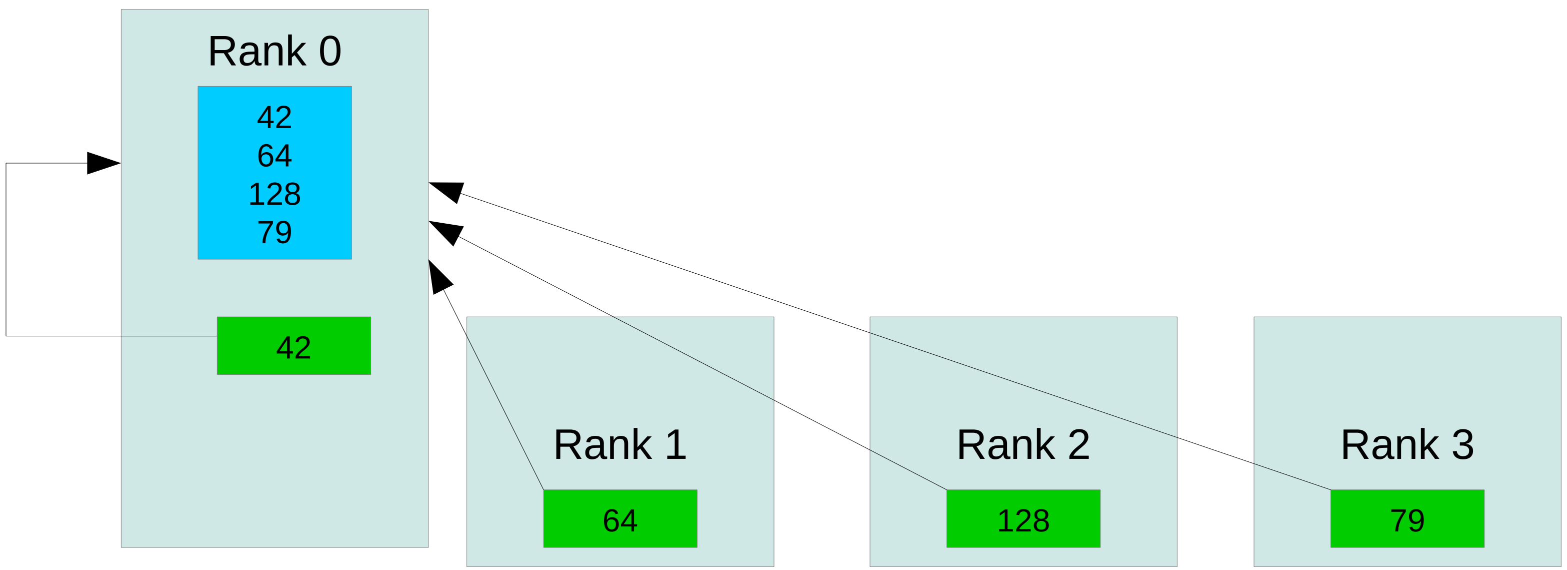
- They *look* pretty much the same as Sendrecv

```
int MPI_Scatter(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

- The send-{buf,count,type} are only relevant on the root rank
- Mind that the root's send buffer must contain *p* times as many elements as the sendcount, for *p* participants
  - *e.g.* if you're scattering to 4 ranks, with a sendcount of 1, there has to be 4 elements in the buffer

MPI_Gather

- This is the same thing, just in the opposite direction



NTNU – Trondheim
Norwegian University of
Science and Technology

Gather arguments

The list is the same as before:

```
int MPI_Gather(
    const void *sendbuf, int sendcount, MPI_Datatype sendtype,
    void *recvbuf, int recvcount, MPI_Datatype recvtype,
    int root, MPI_Comm comm
);
```

- This time it's the recv-{buffer,count,type} that are only relevant to the root
- Mind the size of the receive-buffer

NTNU – Trondheim
Norwegian University of
Science and Technology

Analyzing a collective operation

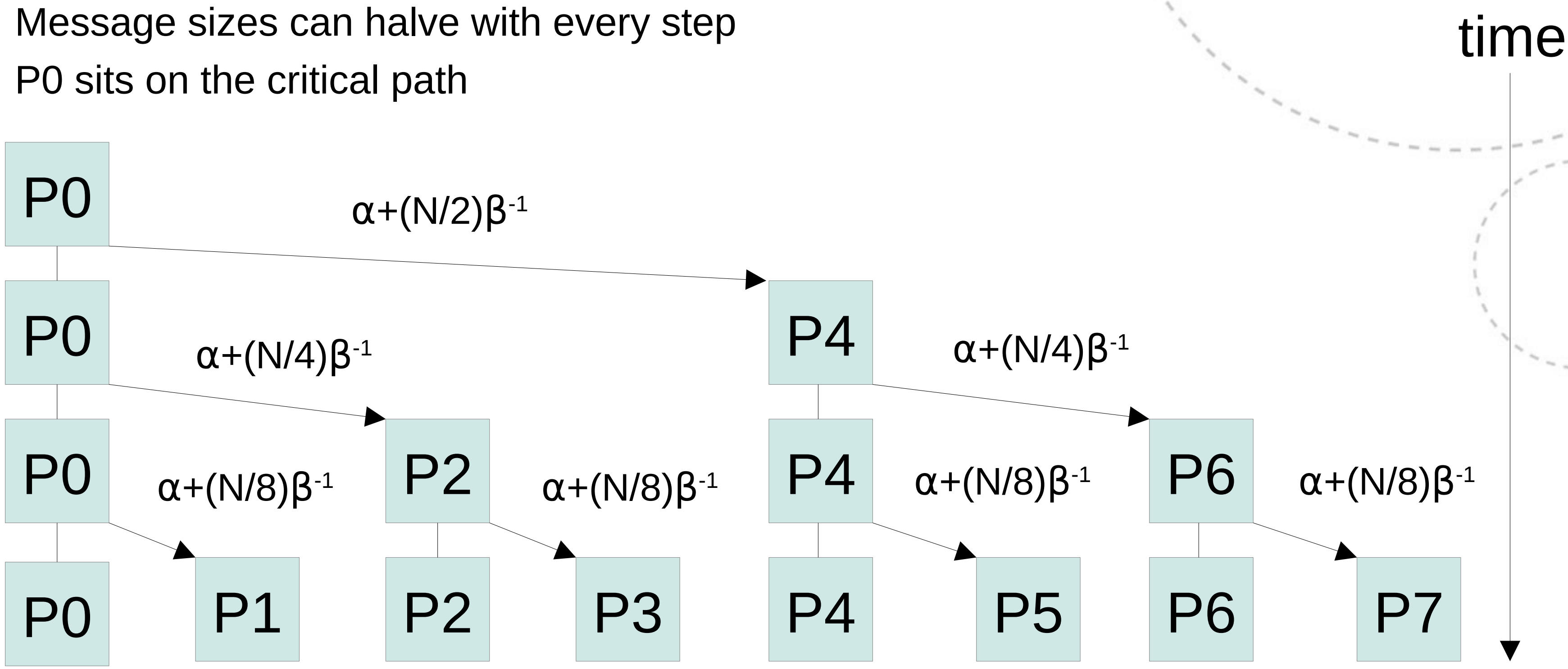
- There can be multiple ways to implement collective operations
- Suppose we use a linear approach to scatter N elements from rank 0 in a collective of *p* ranks
  - Just let rank 0 send all the messages, one after the other
- There will be (p-1) latencies
- Each send requires (N/p)β^{-1} of the bandwidth, so

$$T_{scatter}(N,p) = (p-1)\alpha + \frac{p-1}{p} N \beta^{-1}$$

NTNU – Trondheim
Norwegian University of Science and Technology

Scatter using a binary tree

- Message sizes can halve with every step
- P0 sits on the critical path



$$T_{scatter}(N,p) = log_2(p)\alpha + \beta^{-1} \sum_{i=1}^{log_2(p)} \frac{N}{2^i} = log_2(p)\alpha + \frac{p-1}{p} N\beta^{-1}$$

Conclusions from the comparison

- For scatter, we can save some latency by choosing communication patterns cleverly
- It doesn't make any difference to the bandwidth requirement
- That stands to reason, because rank 0 has to push the same amount of data out the door either way

In reality

- We glossed over the fact that not all links are equal
- Still, we figured out something about the two communication patterns, independent of platform details
- Dissecting communication patterns like this is a handy skill
- You can try it with reductions at home