CS 220: Introduction to Parallel Computing

Measuring Performance

Lecture 16

Today's Agenda

- MPI Collective Communication
- Measuring Performance
- Keeping Time
- Putting it together: estimating pi

Today's Agenda

- **MPI Collective Communication**
- Measuring Performance
- Keeping Time
- Putting it together: estimating pi

MPI Collective Communication

- Thus far we have focused on point-to-point communication
  - **Process A** sends to **Process B**
  - **Process C** waits to hear from **Process D**
- This is fine-grained, and it can be difficult to coordinate when we're working with thousands of cores
- MPI offers some functions that ease this burden: **collective communication**

Broadcasting

```
int MPI_Bcast(
    void *buffer,
    int count,
    MPI_Datatype datatype,
    int root,
    MPI_Comm comm)
```

- Mostly the same as MPI_Send!
- **root** tells us which process will send the message to the rest
  - Nice because we don't need if (p == 0) { … }

Bcast: A Note

- One unintuitive thing about MPI_Bcast: **all** the participating processes call the function
- Process 8 can't call MPI_Bcast and then the rest just MPI_Recv to get the value!
  - Instead, they all just call MPI_Bcast and retrieve the result
- So remember folks: when you use MPI_Bcast, make sure all the processes involved are calling the function!

Barrier Synchronization

```
int MPI_Barrier(
    MPI_Comm comm)
```

- This **blocks** execution until all processes execute it
  - Lets us sync up processes
- Great for situations where we want all processes to check in

Reduction Operations

```
int MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

- MPI can even take care of collecting data for us
- Summing up data from several processes
- Finding the maximum value
- Etc.
- We'll see this today

MPI_Scatter

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

- Somewhat like a broadcast
- Sends data to all the processes
  - From **root**
- Automatically divides the input based on process ranks

MPI_Gather

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

- Picks up the scattered pieces and puts them back together
- Elements are transferred back to **root**
- What would scatter+gather work well for?

Today's Agenda

- A Few Technical Details
- MPI Collective Communication
- **Measuring Performance**
- Keeping Time
- Putting it together: estimating pi

Measuring Parallel Performance

- There are two common metrics for measuring the performance of our parallel algorithms:
  - Speedup
  - Parallel Efficiency
- Evaluating these is crucial: if we're not gaining anything from parallelism, there's no reason to do it
- A closely related concept is **scalability**
  - How our algorithm performs when we give it more resources

Speedup

- The **speedup** of a parallel program is given by:

$$S = \frac{T_{serial}}{T_{parallel}}$$

- How long the serial (original, non-parallel) program takes divided by the parallel run time
- Best speedup possible: ***S = p***
  - Where p is the number of processes

Efficiency

- The **parallel efficiency** of a program is given by:

$$E = \frac{S}{p} = \frac{T_{serial}}{pT_{parallel}}$$

- The speedup divided by the number of processes
- Best efficiency possible: **1**

Scalability (1/2)

- It's possible to write an algorithm that has high efficiency on two, four, eight cores, **but**:
  - Maybe when you try to run it on 16 cores efficiency starts to drop
- There are many reasons this can happen
  - Your algorithm requires a lot of communication
  - Your processes spend a lot of time blocked
  - In some cases, you're only as fast as the **slowest** worker

Scalability (2/2)

- A program that **scales** can use additional resources effectively
  - Doubling the number of processes should halve the execution time!
- We can measure this by calculating parallel efficiency
- If efficiency decreases as we add processes, then the algorithm is **not** scalable

Today's Agenda

- A Few Technical Details
- MPI Collective Communication
- Measuring Performance
- **Keeping Time**
- Putting it together: estimating pi

Keeping Track of Time

- We've done a little bit of timing in our past lectures
  - Using the **time** command
- Not fine-grained
  - We can only test how long it takes to run the entire program
  - What happens when we prompt for a value? What about application startup time (from the OS)?
- We need to be able to track things at a finer level

The **clock()** function

- C includes a clock function that tells us the **number of clock ticks** since the program started
  - Originally intended to be the number of CPU cycles but is implementation-specific
- Different hardware has different clock resolutions
  - Often clock() is a fairly low-resolution timer
- To translate the abstract notion of clock ticks into time, we can use CLOCKS_PER_SEC
  - clock() / CLOCKS_PER_SEC

The **gettimeofday()** function

- On Unix-based systems, this function provides the *wall clock time,* generally with 1 us precision
  - Also hardware dependent
- Wall clock time: the **actual** time taken for something to run
  - As opposed to CPU time
- Usually a better option than clock(), if you have it
- #include <timer.h>

And finally, MPI_Wtime()

- In MPI programs, we have a third option: MPI_Wtime()
- This returns a double with the current wall clock time
  - Uses the best timer available on your platform
- For our MPI programs, we'll use this

Calculating Time Elapsed

- Let's time an operation:

```
double time1 = MPI_Wtime(); /* Start */
solve_worlds_problems(true);
/* Now we wait... */
double time2 = MPI_Wtime(); /* End */
```

- How long did it take?

- `printf("Time: %.10lf\n", time2 - time1);`

Today's Agenda

- A Few Technical Details
- MPI Collective Communication
- Measuring Performance
- Keeping Time
- **Putting it together: estimating pi**

Leibniz Formula for pi

- We can estimate pi with the following formula:

$$1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots=\frac{\pi}{4}.$$

- The more iterations, the more accurate our estimate gets
- We can split these iterations up across multiple processes to speed things up
- MPI to the rescue!