CISC 372: Parallel Computing

Wildcards and Nondeterminism

Stephen F. Siegel

Department of Computer and Information Sciences
University of Delaware

“Wildcard” receives: `MPI_ANY_SOURCE`

`MPI_Recv(buf, count, datatype, source, tag, comm, status)`

- `source` argument can be `MPI_ANY_SOURCE`
  - special constant defined by MPI
  - means “receive message from any source”
  - use with care
  - introduce nondeterminism into the parallel program
  - program can produce different results on different executions
  - sometimes this is necessary (dynamic load-balancing)
  - do not use unless necessary for algorithm
- can use in combination with `MPI_ANY_TAG`

Wildcard receive: example using `MPI_ANY_SOURCE`: `anysource.c`

```c
#include<stdio.h>
#include<mpi.h>
int main() {
  int message, rank;
  MPI_Init(NULL, NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank == 0) {
    message = 0; MPI_Send(&message, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  } else if (rank == 1)  {
    message = 1; MPI_Send(&message, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
  } else if (rank == 2) {
    for (int i=0; i<2; i++) {
      MPI_Recv(&message, 1, MPI_INT, MPI_ANY_SOURCE, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      printf("Proc 2 received: %d\n", message);
    }
  }
  MPI_Finalize();
}
```

```
> mpiexec -n 3 ./anysource.exec
Proc 2 received: 0
Proc 2 received: 1
```

```
> mpiexec -n 3 ./anysource.exec
Proc 2 received: 1
Proc 2 received: 0
```

Semantics: matching

A send operation and a receive operation match if all of the following hold:

1. the communicators are the same
2. the rank of the receiver equals the `dest` argument in the send
3. the rank of the sender equals the `source` argument in the receive
   - OR the `source` argument is `MPI_ANY_SOURCE`
4. the tag in the send equals the `tag` argument in the receive
   - OR the `tag` argument is `MPI_ANY_TAG`

Note:

- the receiver can determine if an incoming message matches
  - by examining only the message envelope
- the message data plays no role in determining a match
- the message datatype plays no role in determining a match

Semantics: matching, cont.

- messages within a channel are ordered
  - a receive can only be matched with the oldest matching message in the channel
- messages in different channels are not ordered
  - a wildcard (`MPI_ANY_SOURCE`) receive can choose any incoming channel with a matching message
  - and select the oldest matching message from that channel

Message Ordering Example 1

```
Rank 0:
  MPI_Recv(MPI_ANY_SOURCE);
  MPI_Recv(MPI_ANY_SOURCE);

Rank 1:
  MPI_Send(to process 0);

Rank 2:
  MPI_Send(to process 0);
```

Which message gets matched with which receive?

Answer: either way — no order on messages in different channels

Message Ordering Example 2

```
Rank 0:
  MPI_Recv(MPI_ANY_SOURCE);
  MPI_Recv(MPI_ANY_SOURCE);

Rank 1:
  MPI_Send(to process 0);
  MPI_Send(to process 2);

Rank 2:
  MPI_Recv(from process 1);
  MPI_Send(to process 0);
```

Now process 1 sends its message to 0 before process 2 does.
Which message gets matched with which receive?
Answer: either way — no order on messages in different channels
It doesn't matter that proc 1 sent before proc 2.

Message Ordering Example 3

```
Rank 0:
  MPI_Recv(MPI_ANY_SOURCE);
  MPI_Recv(MPI_ANY_SOURCE);

Rank 1:
  MPI_Send(to process 0);
  MPI_Send(to process 0);
```

Which message gets matched with which receive?

Answer: first send is matched with first receive, second send is matched with second receive

Messages within a channel are ordered

Determinism and nondeterminism

- Programs restricted to
  - deterministic sequential operations
  - only one process performs I/O
  - `MPI_Send`, `MPI_Recv`, `MPI_Sendrecv`, `MPI_Init`, `MPI_Finalize`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_ANY_TAG`
  - collective operations other than reductions on floating-point numbers
  - but not `MPI_ANY_SOURCE`
- are guaranteed to be deterministic
  - given the same input twice, same output will be produced
  - even though the paths from input to output may be differ
- But if you add `MPI_ANY_SOURCE`
  - the program may be nondeterministic
  - given same input twice, two different outputs possible
  - see `simplend.c`

The need for barrier synchronization

- programs that use MPI_ANY_SOURCE sometimes require barrier synchronization!
- this is one of the few times barriers are absolutely needed

Wildcard deadlock example: function f

```
/* Each non-root process sends a message to root.
   Root receives using MPI_ANY_SOURCE.
   This is a perfectly fine deadlock-free function.
 */
void f() {
  if (myrank == 0) {
    MPI_Status status;
    int x;
    for (int i = 1; i < nprocs; i++) {
      MPI_Recv(&x, 1, MPI_INT, MPI_ANY_SOURCE, 0, comm, &status);
      printf("Proc 0: received %d from proc %d\n", x, status.MPI_SOURCE);
    }
  } else {
    MPI_Send(&myrank, 1, MPI_INT, 0, 0, comm);
  }
}
```

Wildcard deadlock example: function g

```
/* Each non-root process sends a message to root.
   Root receives in order of increasing rank.
   This is a perfectly fine deadlock-free function.
 */
void g() {
  if (myrank == 0) {
    MPI_Status status;
    int x;
    for (int i = 1; i < nprocs; i++) {
      MPI_Recv(&x, 1, MPI_INT, i, 0, comm, &status);
      printf("Proc 0: received %d from proc %d\n", x, status.MPI_SOURCE);
    }
  } else {
    MPI_Send(&myrank, 1, MPI_INT, 0, 0, comm);
  }
}
```

Wildcard deadlock example: function `main`

What happens when I call the two deadlock-free functions in sequence?

```
int main() {
  MPI_Init(NULL, NULL);
  MPI_Comm_size(comm, &nprocs);
  MPI_Comm_rank(comm, &myrank);
  f();
  g();
  MPI_Finalize();
}
```

```
mpiexec -n 4 ./wcdl.exec
Proc 0: received 1 from proc 1
Proc 0: received 1 from proc 1
Proc 0: received 2 from proc 2
^C[mpiexec@basie.local]
```

Deadlock!

Wildcard deadlock example: what happened?

1. proc 1 in function `f` sent message to proc 0
2. proc 1 in function `g` sent message to proc 0
3. proc 1 terminates
4. proc 2 in function `f` sent message to proc 0
5. proc 0 in function `f` received message from proc 1 at `MPI_ANY_SOURCE`
6. proc 0 in function `f` received message from proc 1 at `MPI_ANY_SOURCE`
7. proc 0 in function `f` received message from proc 2 at `MPI_ANY_SOURCE`
8. proc 0 in function `g` waits for message from proc 1

A message from proc 1 in `g` was received by proc 0 in `f`.

- not what the programmer intended

Wildcard deadlock example: a solution

- place a `barrier` between `f` and `g`
- no one will be able to enter `g` until everyone has completed `f`

```
int main() {
  MPI_Init(NULL, NULL);
  MPI_Comm_size(comm, &nprocs);
  MPI_Comm_rank(comm, &myrank);
  f();
  MPI_Barrier(comm);
  g();
  MPI_Finalize();
}
```

```
mpiexec -n 4 ./wcdl.exec
Proc 0: received 3 from proc 3
Proc 0: received 1 from proc 1
Proc 0: received 2 from proc 2
Proc 0: received 1 from proc 1
Proc 0: received 2 from proc 2
Proc 0: received 3 from proc 3
```

Load Balancing

- load balancing crucial to performance
- some problems can be broken up in a predictable way
  - diffusion, π, sat
  - each process does (roughly) same amount of work
- for other problems this is difficult
  - no way to predict how long a task will take
  - many numerical algorithms require iterating until convergence
  - example: numerical integration
  - heterogenous hardware: processors running at different speeds
  - cyclic distributions are not always going to solve this problem

The Manager-Worker pattern

- break up problem into finite set of tasks
- there should be many more tasks than processes
- one process plays role of manager
- remaining processes are workers
- manager
  1. distributes one task to each worker
  2. waits for any worker to send back result
  3. processes result and sends new task to that worker
  4. if no tasks remain, sends termination signal to worker instead
  5. when all results have been returned and termination signals sent, finished
- worker
  1. waits for task from manager
  2. solves the task and sends result to manager
  3. repeat until termination signal received

Non-determinism

- algorithm is inherently non-deterministic
- everything depends on the order in which workers send back results
- it is possible for one worker to do all but `nprocs-2` tasks
- it is possible for workers to do same number of tasks
- manager must use some nondeterminisitic MPI construct, e.g.
  - `MPI_ANY_SOURCE`
  - `MPI_Waitany`
  - `MPI_Waitsome`
  - `MPI_Test`
  - `MPI_Testany`
  - `MPI_Testsome`
  - `MPI_Probe`
  - `MPI_Iprobe`
- a correct program should return same result independent of these choices

Manager-worker example: matrix-matrix multiplication

A: $N \times L$ B: $L \times M$ C: $N \times M$

Example: $N = 4$, $L = 3$, $M = 2$

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \\ a_{30} & a_{31} & a_{32} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \\ b_{20} & b_{21} \end{bmatrix} = \begin{bmatrix} a_{00}b_{00} + a_{01}b_{10} + a_{02}b_{20} & a_{00}b_{01} + a_{01}b_{11} + a_{02}b_{21} \\ a_{10}b_{00} + a_{11}b_{10} + a_{12}b_{20} & a_{10}b_{01} + a_{11}b_{11} + a_{12}b_{21} \\ a_{20}b_{00} + a_{21}b_{10} + a_{22}b_{20} & a_{20}b_{01} + a_{21}b_{11} + a_{22}b_{21} \\ a_{30}b_{00} + a_{31}b_{10} + a_{32}b_{20} & a_{30}b_{01} + a_{31}b_{11} + a_{32}b_{21} \end{bmatrix}$$

- problem can be viewed as a series of matrix-vector multiplications
  - multiply row i of A by B to get row i of C ($i = 0, \ldots N-1$)

$$\begin{bmatrix} A[0] \\ A[1] \\ A[2] \\ A[3] \end{bmatrix} \times B = \begin{bmatrix} A[0] \times B \\ A[1] \times B \\ A[2] \times B \\ A[3] \times B \end{bmatrix}$$

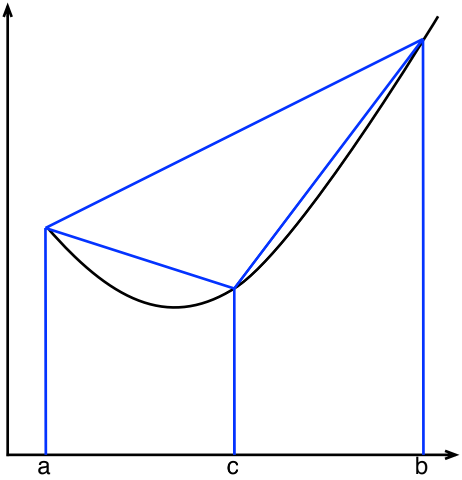
Matrix-matrix multiplication: `matmat.c` `matmat_mpi.c`

- sequential solution: `matmat.c`
- parallel solution: `matmat_mpi.c`
  - uses manager-worker pattern
  - a task: one row of A times B to get one row of C

Other sources of nondeterminism: floating-point

- addition and multiplication of real numbers are associative operations
- this is not true for floating-point numbers!
  - `a+(b+c)` does not necessarily equal `(a+b)+c`
  - why?
  - round-off error
  - $\texttt{ROUND}(a + \texttt{ROUND}(b + c))$ does not necessarily equal $\texttt{ROUND}(\texttt{ROUND}(a + b) + c)$
- this is a problem with `MPI_Reduce` used with floating-point numbers and `MPI_SUM`
  - could run the code twice and get two different answers
  - for most applications, differences are "small"
  - but not always
  - in any case: makes testing hard
- to parallelize sequential programs with floating point operations, for greatest assurance ...
  - floating point operations should be identical in both programs
  - if one computes `a+(b+c)` the other must compute `a+(b+c)`, not `(a+b)+c`
  - then they will get exact same result in all cases

Example: Numerical Integration with Trapezoid Rule



- compute area A of large trapezoid
- divide interval in half
- compute areas A_l and A_r of left and right trapezoids
- compare $A_l + A_r$ with A
- if difference is "sufficiently small" return $A_l + A_r$
- else call recursively on left and right subintervals and return sum

See `integral.c`

Characteristics of Algorithm

- it is not known beforehand how many subdivisions will be required to achieve convergence
- the number of subdivisions may differ at different points on the x-axis
  - where curve is close to a straight line, fast converence
  - where higher derivatives are high, slower convergence
- balanced static partitioning of work not possible
- manager-worker pattern called for
  - break up $[a, b]$ into subintervals in exact same way as sequential
  - many more subintervals than processes (e.g., $100\times$)
  - task: compute integral over one subinterval
  - give each worker one task, ...
  - manager sums results in exact same order as sequential

See `integral_mpi.c`