Announcements

- Programming Assignment #1 is slightly delayed.
- See class web page for paper assignments
  - Everyone sends questions for 3 papers during the term

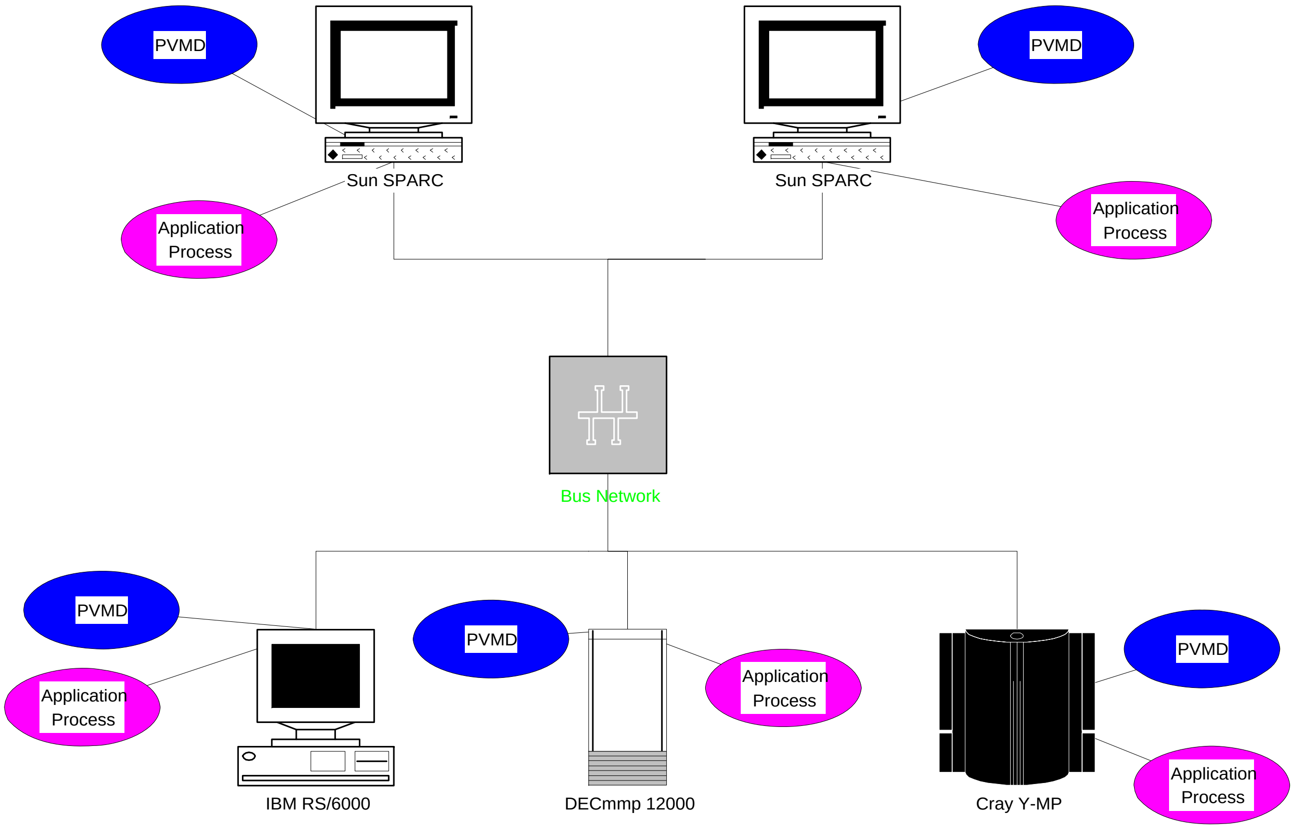
MPI Communication Calls

- Parameters
  - var – a variable
  - num – number of elements in the variable to use
  - type {MPI_INT, MPI_REAL, MPI_BYTE}
  - root – rank of processor at root of collective operation
  - dest – rank of destination processor
  - status - variable of type MPI_Status;
- Calls (all return a code – check for MPI_Success)
  - MPI_Send(var, num, type, dest, tag, MPI_COMM_WORLD)
  - MPI_Recv(var, num, type, dest, MPI_ANY_TAG, MPI_COMM_WORLD, &status)
  - MPI_Bcast(var, num, type, root, MPI_COMM_WORLD)
  - MPI_Barrier(MPI_COMM_WORLD)

PVM

- Provide a simple, free, portable parallel environment
- Run on everything
  - Parallel Hardware: SMP, MPPs, Vector Machines
  - Network of Workstations: ATM, Ethernet,
    - UNIX machines and PCs running Win*
  - Works on a heterogenous collection of machines
    - handles type conversion as needed
- Provides two things
  - message passing library
    - point-to-point messages
    - synchronization: barriers, reductions
  - OS support
    - process creation (pvm_spawn)

PVM Environment (UNIX)



- One PVMD per machine
  - all processes communicate through pvmd (by default)
- Any number of application processes per node

PVM Message Passing

- All messages have tags
  - an integer to identify the message
  - defined by the user
- Messages are constructed, then sent
  - pvm_pk{int,char,float}(*var, count, stride)
  - pvm_unpk{int,char,float} to unpack
- All proccess are named based on task ids (tids)
  - local/remote processes are the same
- Primary message passing functions
  - pvm_send(tid, tag)
  - pvm_recv(tid, tag)

PVM Process Control

- Creating a process
  - pvm_spawn(task, argv, flag, where, ntask, tids)
  - flag and where provide control of where tasks are started
  - ntask controls how many copies are started
  - program must be installed on target machine
- Ending a task
  - pvm_exit
  - does not exit the process, just the PVM machine
- Info functions
  - pvm_mytid() - get the process task id

PVM Group Operations

- Group is the unit of communication
  - a collection of one or more processes
  - processes join group with pvm_joingroup(“<group name>“)
  - each process in the group has a unique id
    - pvm_gettid(“<group name>“)
- Barrier
  - can involve a subset of the processes in the group
  - pvm_barrier(“<group name>“, count)
- Reduction Operations
  - pvm_reduce(void (*func)(), void *data, int count, int datatype, int msgtag, char *group, int rootinst)
    - result is returned to rootinst node
    - does not block
  - pre-defined funcs: PvmMin, PvmMax,PvmSum,PvmProduct

PVM Performance Issues

- Messages have to go through PVMD
  - can use direct route option to prevent this problem
- Packing messages
  - semantics imply a copy
  - extra function call to pack messages
- Heterogenous Support
  - information is sent in machine independent format
  - has a short circuit option for known homogenous comm.
    - passes data in native format then

Sample PVM Program

```
int main(int argc, char **argv) {
    int myGroupNum;
    int friendTid;
    int mytid;
    int tids[2];
    int message[MESSAGESIZE];
    int c,i,okSpawn;

    /* Initialize process and spawn if necessary */
    myGroupNum=pvm_joingroup("ping-pong");
    mytid=pvm_mytid();
    if (myGroupNum==0)  { /* I am the first process */
        pvm_catchout(stdout);
        okSpawn=pvm_spawn(MYNAME,argv,0,"",1,&friendTid);
        if (okSpawn!=1) {
            printf("Can't spawn a copy of myself!\n");
            pvm_exit();
            exit(1);
        }
        tids[0]=mytid;
        tids[1]=friendTid;
    } else { /*I am the second process */
        friendTid=pvm_parent();
        tids[0]=friendTid;
        tids[1]=mytid;
    }
    pvm_barrier("ping-pong",2);

    /* Main Loop Body */
    if (myGroupNum==0) {
        /* Initialize the message */
        for (i=0 ; i<MESSAGESIZE ; i++) {
            message[i]='1';
        }

        /* Now start passing the message back and forth */
        for (i=0 ; i<ITERATIONS ; i++) {
            pvm_initsend(PvmDataDefault);
            pvm_pkint(message,MESSAGESIZE,1);
            pvm_send(tid,msgid);

            pvm_recv(tid,msgid);
            pvm_upkint(message,MESSAGESIZE,1);
        }
    } else {
            pvm_recv(tid,msgid);
            pvm_upkint(message,MESSAGESIZE,1);
            pvm_initsend(PvmDataDefault);
            pvm_pkint(message,MESSAGESIZE,1);
            pvm_send(tid,msgid);
    }
    pvm_exit();
    exit(0);
}
```

Defect Patterns in High Performance Computing

Based on Materials Developed by
Taiga Nakamura

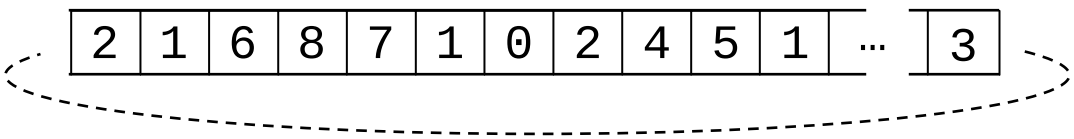
What is This Lecture?

- Debugging and testing parallel code is hard
  - What kinds of software defects (bugs) are common?
  - How can they be prevented or found/fixed effectively?
- Hypothesis: Knowing common defects (bugs) will reduce the time spent debugging
  - … during programming assignments, course projects
- Here: Common defect types in parallel programming
  - "Defect patterns" in HPC
  - Based on the empirical data we collected in past studies
  - Examples are in C/MPI (suspect similar defect types in Fortran/MPI, OpenMP, UPC, CAF, …)

Example Problem

- Consider the following problem:

A sequence of *N* cells



1. N cells, each of which holds an integer [0..9]
   - `E.g., cell[0]=2, cell[1]=1, …, cell[N-1]=3`
2. In each step, cells are updated using the values of neighboring cells
   - $\texttt{cell}_{\texttt{next}}\texttt{[x] = (cell[x-1] + cell[x+1]) mod 10}$
   - $\texttt{cell}_{\texttt{next}}\texttt{[0]=(3+1), cell}_{\texttt{next}}\texttt{[1]=(2+6), …}$
   - (Assume the last cell is adjacent to the first cell)
3. Repeat 2 for *steps* times

What defects can appear when implementing a parallel solution in MPI?

First, Sequential Solution

- Approach to implementation
  - Use an integer array `buffer[]` to represent the cell values
  - Use a second array `nextbuffer[]` to store the values in the next step, and swap the buffers
  - Straightforward implementation!

Sequential C Code

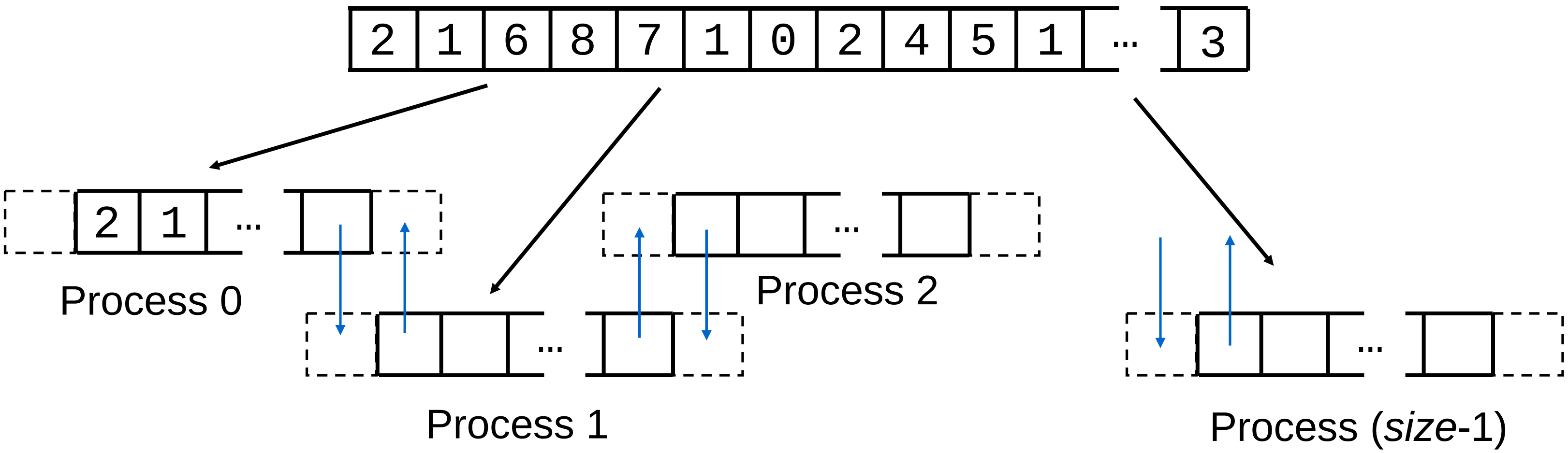
```
/* Initialize cells */
int x, n, *tmp;
int *buffer     = (int*)malloc(N * sizeof(int));
int *nextbuffer = (int*)malloc(N * sizeof(int));
FILE *fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 0; x < N; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}

/* Final output */
...
free(nextbuffer); free(buffer);
```

Approach to a Parallel Version

- Each process keeps (1/size) of the cells
  - *size*:number of processes



- Each process needs to:
  - update the locally-stored cells
  - exchange boundary cell values between neighboring processes (nearest-neighbor communication)

Recurring HPC Defects

- Now, we will simulate the process of writing parallel code and discuss what kinds of defects can appear.

- Defect types are shown as:
  - Pattern descriptions
  - Concrete examples in MPI implementation

Pattern: Erroneous use of language features

- Simple mistakes in understanding that are common for novices
  - E.g., inconsistent parameter types between send and recv,
  - E.g., forgotten mandatory function calls
  - E.g., inappropriate choice of functions

Symptoms:

- Compile-type error (easy to fix)
- Some defects may surface only under specific conditions
  - (number of processors, value of input, hardware/software environment…)

Causes:

- Lack of experience with the syntax and semantics of new language features

Cures & preventions:

- Check unfamiliar language features carefully

Adding basic MPI functions

```
/* Initialize MPI */
MPI_Status status;
status = MPI_Init(NULL, NULL);
if (status != MPI_SUCCESS) { exit(-1); }

/* Initialize cells */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
...

/* Final output */
...

/* Finalize MPI */
MPI_Finalize();
```

What are the bugs?

What are the defects?

```
/* Initialize MPI */
MPI_Status status;      MPI_Init(&argc, &argv);
status = MPI_Init(NULL, NULL);
if (status != MPI_SUCCESS) { exit(-1); }

/* Initialize cells */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }    MPI_Finalize();
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* Main loop */
...
```

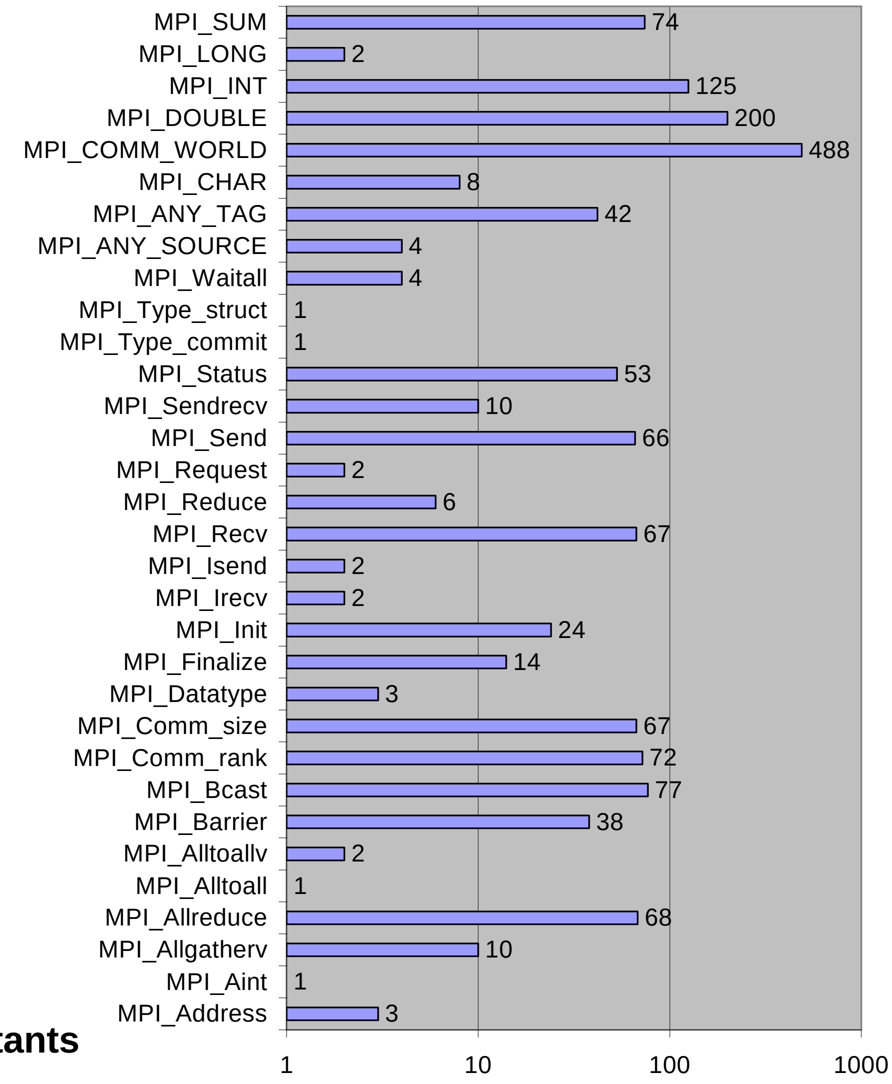
- Passing NULL to MPI_Init is invalid in MPI-1 (ok in MPI-2)
- MPI_Finalize must be called by all processors in every execution path

Does MPI Have Too Many Functions To Remember?

- Yes (100+ functions), but…
- Advanced features are not necessarily used

- Try to understand a few, basic language features thoroughly

MPI keywords in Conjugate Gradient in C/C++ (15 students)



24 functions, 8 constants

Pattern: Space Decomposition

- Incorrect mapping between the problem space and the program memory space

Symptoms:

- Segmentation fault (if array index is out of range)
- Incorrect or slightly incorrect output

Causes:

- Mapping in parallel version can be different from that in serial version
  - E.g., Array origin is different in every processor
  - E.g., Additional memory space for communication can complicate the mapping logic

Cures & preventions:

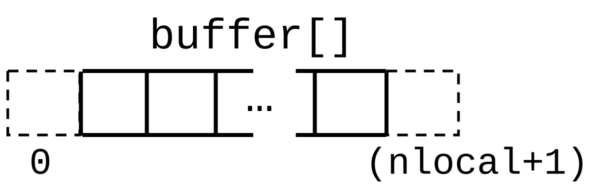
- Validate the memory allocation carefully when parallelizing the code

Decompose the problem space

```
MPI_Comm_size(MPI_COMM_WORLD &size);
MPI_Comm_rank(MPI_COMM_WORLD &rank);
nlocal = N / size;
buffer     = (int*)malloc((nlocal+2) * sizeof(int));
nextbuffer = (int*)malloc((nlocal+2) * sizeof(int));

/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 0; x < nlocal; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  ...

  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```



What are the bugs?

What are the defects?

```
MPI_Comm_size(MPI_COMM_WORLD &size);
MPI_Comm_rank(MPI_COMM_WORLD &rank);
nlocal = N / size;  N may not be divisible by size
buffer     = (int*)malloc((nlocal+2) * sizeof(int));
nextbuffer = (int*)malloc((nlocal+2) * sizeof(int));

/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  ...

  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- N may not by divisible by size
- Off by one error in inner loop

Pattern: Side-effect of Parallelization

- Ordinary serial constructs can cause defects when they are accessed in parallel contexts

Symptoms:

- Various correctness/performance problems

Causes:

- "Sequential part" tends to be overlooked
  - Typical parallel programs contain only a few parallel primitives, and the rest of the code is made of a sequential program running in parallel

Cures & preventions:

- Don't just focus on the parallel code
- Check that the serial code is working on one processor, but remember that the defect may surface only in a parallel context

Data I/O

```
/* Initialize cells with input file */
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
nskip = ...
for (x = 0; x < nskip; x++) { fscanf(fp, "%d", &dummy);}
for (x = 0; x < nlocal; x++) { fscanf(fp, "%d", &buffer[x+1]);}
fclose(fp);

/* Main loop */
...
```

- What are the defects?

Data I/O

```
/* Initialize cells with input file */
if (rank == 0) {
fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < nlocal; x++) { fscanf(fp, "%d", &buffer[x+1]);}
for (p = 1; p < size; p++) {
  /* Read initial data for process p and send it */
}
fclose(fp);
}
else {
  /* Receive initial data*/
}
```

- Filesystem may cause performance bottleneck if all processors access the same file simultaneously
  - (Schedule I/O carefully, or let “master” processor do all I/O)

Generating Initial Data

```
/* What if we initialize cells with random values... */
srand(time(NULL));
for (x = 0; x < nlocal; x++) {
  buffer[x+1] = rand() % 10;
}

/* Main loop */
...
```

- What are the defects?

- (Other than the fact that rand() is not a good pseudo-random number generator in the first place…)

What are the Defects?

```
/* What if we initialize cells with random values... */
srand(time(NULL));    srand(time(NULL) + rank);
for (x = 0; x < nlocal; x++) {
  buffer[x+1] = rand() % 10;
}

/* Main loop */
...
```

- All procs might use the same pseudo-random sequence, spoiling independence
- Hidden serialization in rand() causes performance bottleneck

Pattern: Synchronization

- Improper coordination between processes
  - Well-known defect type in parallel programming
  - Deadlocks, race conditions

Symptoms:

- Program hangs
- Incorrect/non-deterministic output

Causes:

- Some defects can be very subtle
- Use of asynchronous (non-blocking) communication can lead to more synchronization defects

Cures & preventions:

- Make sure that all communications are correctly coordinated

Communication

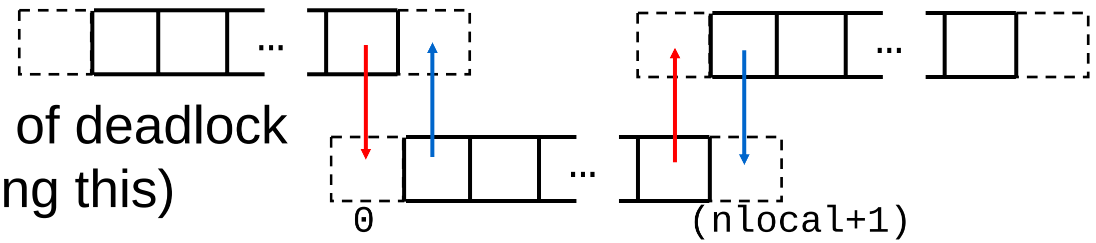
```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- Obvious example of deadlock (can't avoid noticing this)



Another Example

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- This causes deadlock too
- MPI_Ssend is a *synchronous* send (see the next slides.)

Yet Another Example

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

Potential deadlock

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- This may work (many novice programmers write this code)
- but it can cause deadlock with some implementation or parameters

Modes of MPI blocking communication

- http://www.mpi-forum.org/docs/mpi-11-html/node40.html
  - **Standard** (MPI_Send): may either return immediately when the outgoing message is buffered in the MPI buffers, or block until a matching receive has been posted.
  - **Buffered** (MPI_Bsend): a send operation is completed when the MPI buffers the outgoing message. An error is returned when there is insufficient buffer space
  - **Synchronous** (MPI_Ssend): a send operation is complete only when the matching receive operation has started to receive the message.
  - **Ready** (MPI_Rsend): a send can be started only after the matching receive has been posted.
- In our code MPI_Send won't probably be blocked in most implementations (each message's just one integer), but it should still be avoided.
- A "correct" solution could be:
  - (1) alternate the order of send and recv
  - (2) use MPI_Bsend with sufficient buffer size
  - (3) MPI_Sendrecv, or
  - (4) MPI_Isend/recv

Non-Blocking Communication

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Isend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request4);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- What are the defects?

What are the Defects?

```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[(x-1+N)%N]+buffer[(x+1)%N]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Isend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &request4);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- Synchronization (e.g. MPI_Wait, MPI_Barrier) is needed at each iteration (but too many barriers can cause a performance problem)

Pattern: Performance defect

- Scalability problem because processors are not working in parallel
  - The program output itself is correct
  - Perfect parallelization is often difficult: need to evaluate if the execution speed is unacceptable

Symptoms:

- Sub-linear scalability
- Performance much less than expected (e.g, most time spent waiting),

Causes:

- Unbalanced amount of computation
- Load balancing may depend on input data

Cures & preventions:

- Make sure all processors are "working" in parallel
- Profiling tool might help

Scheduling communication

```
if (rank != 0) {
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
}
if (rank != size-1) {
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
}
```

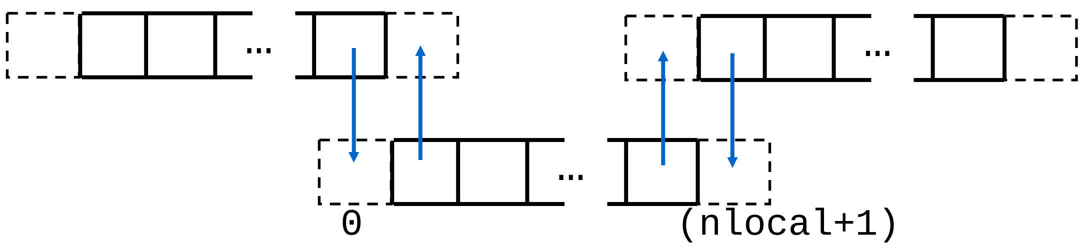
- Complicated communication pattern- does not cause deadlock

What are the defects?

What are the bugs?

```
if (rank != 0) {
  MPI_Ssend (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD, &status);
}
if (rank != size-1) {
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%size,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[1], 1, MPI_INT, (rank+size-1)%size,
    tag, MPI_COMM_WORLD);
}
```

- Communication requires O(size) time (a “correct” solution takes O(1))



1 Send → 0 Recv → 0 Send → **1 Recv**
2 Send → **1 Recv** → **1 Send** → 2 Recv
3 Send → 2 Recv → 2 Send → 3 Recv

Summary

- This is an attempt to share knowledge about common defects in parallel programming
  - Erroneous use of language features
  - Space Decomposition
  - Side-effect of Parallelization
  - Synchronization
  - Performance defect
- The slides will be available at
  - http://www.cs.umd.edu/~hollings/cs714/f06/lect04/index.shtml
- Homework (due Sep 19)
  - http://www.cs.umd.edu/~hollings/cs714/f06/homework1.pdf
  - Find defects in a given MPI program
- Programming assignments (later)
  - Try to avoid these defect patterns in your code