Defect Patterns in High Performance Computing

Taiga Nakamura
(edits by A. Sussman)
University of Maryland

Notes

- MPI project to be posted today, due Wed., March 1, 6PM, via email
- Office hours? Scheduled, or by appointment?
- Send questions for readings, starting Thursday
  – additional readings posted soon

Background

- Debugging and testing parallel code is hard
  – How can bugs be prevented or found/fixed effectively?
- "Knowing" common defects (bugs) will reduce the time spent debugging
  – Novice developers can *learn* how to detect/prevent them
  – Someone may develop tools and/or improve language
- HPCS project built "Defect patterns" for high performance programming (HPC)
  – Based on the empirical data collected in various studies
  – Examples in this presentation are shown in C + MPI (Message Passing Interface)

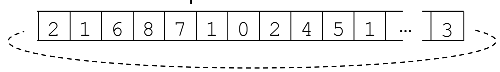
Differentiating Factors of HPC

- **Platform**: Computational power of today's HPC systems is achieved by massively parallel systems. Writing a scalable program on these systems is difficult.
- **Performance**: Slow execution speed can be a defect even if the output is correct. Achieving good performance on multiple processors is often difficult
- **Language**: Developers usually use special HPC languages and libraries (MPI, OpenMP, UPC, CAF, CUDAß, ...), each with their own ways of handling issues such as communication and synchronization. SPMD (Single Program, Multiple Data) approach is dominant
- **Developers**: Software often developed by scientists and grad students without formal training in software engineering. Traditional software engineering processes or practices are not necessarily used in HPC projects
- **Tools**: The use of modern tools (IDEs, graphical debuggers, defect detection tools, profiling tools, etc.) is not as common as in other domains
- **Portability**: Portability is very important for HPC applications since they must be run on various platforms depending on the computational resources available
- **Validation**: Given the nature of HPC applications, the correct outputs are not always known, so debugging is particularly challenging and costly.

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

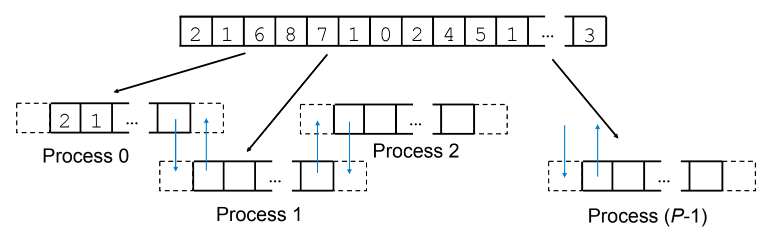
```
/* Initialize cells */
int x, n, *tmp;
int *buffer     = (int*)malloc(N * sizeof(int));
int *nextbuffer = (int*)malloc(N * sizeof(int));
FILE *fp = fopen("input.dat", "r");
if (fp == NULL) { exit(-1); }
for (x = 0; x < N; x++) { fscanf(fp, "%d", &buffer[x]); }
fclose(fp);

/* 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 stores (1/P) of the cells
  - *P* : 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

- We simulate the process of writing parallel code and discuss what kinds of defects can appear.
- Defect types are shown as:
  - Pattern descriptions (symptoms, causes, cures & preventions)
  - Concrete examples in MPI implementation

Pattern: **Erroneous use of parallel language features**

- "Simple" mistakes that are common for novices: language usage, choice of function, etc.
  - E.g., forgotten mandatory function calls for init/finalize
  - E.g., inconsistent parameter types between send and recv

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:

- Understand subtleties and variations of language features
- In a large code, confine parallel function calls to a particular part of the code to help make fewer errors

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 */
...
```

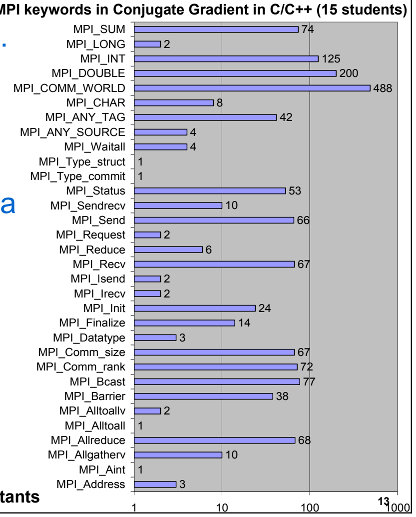
- Passing NULL to MPI_Init is invalid in MPI-1 (ok in later MPI versions)
- MPI_Finalize must be called by all processes in **every** execution path

Does MPI Have Too Many Functions To Remember?

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

- **Lesson**: 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 (maybe 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

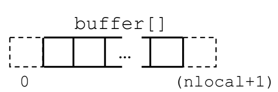
Cures & preventions:

- Validate array origin, whether buffer includes guard buffers, whether buffer refers to global space or local space, etc. - these can change while parallelizing the code
- Encapsulate the mapping logic to a dedicated function
- Consider designing serial code that is easy to parallelize

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

- Loop boundary and array indexes must be changed to reflect the effect of space decomposition (a sequential implementation should have been written to make parallelization easier)
- **Lesson**: make sure the parallel code works correctly on 1 proc

Pattern: Hidden Serialization

- Side-effect of parallelization: ordinary serial constructs can cause defects when they are used in parallel contexts
  - E.g. I/O hotspots
  - E.g. Hidden serialization in library functions

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 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 < P; p++) {
  /* Read initial data for process p and send it */
}
fclose(fp);
}
else {
  /* Receive initial data*/
}
```

- **Lesson**: filesystem may cause performance bottleneck if all processors access the same file simultaneously
  - Schedule I/O carefully, let "master" processor do all I/O, or use parallel 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 */
...
```

- **Lesson**: all processors might use the same pseudo-random sequence, breaking independence assumption (correctness)
- **Lesson**: Hidden serialization in the library function rand() causes performance bottleneck

Pattern: Synchronization

- Improper coordination between processes
  - Well-known defect type in parallel programming
  - Some defects can be very subtle

Symptoms:

- Deadlocks: some execution path can lead to cyclic dependencies among processes and nothing ever happens
- Race conditions: incorrect/non-deterministic output and there can be performance defects due to synchronization too

Causes:

- Use of asynchronous (non-blocking) communication can lead to more synchronization defects
- Too much synchronization can be a performance problem

Cures & preventions:

- Make sure that all communications are correctly coordinated
  - Check the communication pattern with specific number of processes/threads using diagrams

Communication

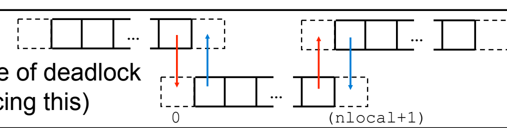
```
/* Main loop */
for (n = 0; n < steps; n++) {
  for (x = 1; x < nlocal+1; x++) {
    nextbuffer[x] = (buffer[x-1]+buffer[x+1]) % 10;
  }
  /* Exchange boundary cells with neighbors */
  MPI_Recv (&nextbuffer[0], 1, MPI_INT, (rank+P-1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal],1,MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1],1,MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1], 1, MPI_INT, (rank+P-1)%P,
    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+P-1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[1],1,MPI_INT,(rank+P-1)%P,
    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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Ssend (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &status);
  MPI_Send (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD);
  MPI_Recv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    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 MPI implementations, runtime environments and/or execution 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 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 probably won't block in most implementations (each message is just one integer), but it should still be avoided for correctness
- A "correct" solution for this defect could be:
  - (1) alternate the order of send and recv
  - (2) use MPI_Bsend with sufficient buffer size
  - (3) use MPI_Sendrecv, or
  - (4) use MPI_Isend and MPI_Irecv

An Example Fix

```
/* 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 */
  if (rank % 2 == 0) { /* even ranks send first */
    MPI_Send (..., (rank+P-1)%P, ...);
    MPI_Recv (..., (rank+1)%P, ...);
    MPI_Send (..., (rank+1)%P, ...);
    MPI_Recv (..., (rank+P-1)%P, ...);
  } else {              /* odd ranks recv first */
    MPI_Recv (..., (rank+1)%P, ...);
    MPI_Send (..., (rank+P-1)%P, ...);
    MPI_Recv (..., (rank+P-1)%P, ...);
    MPI_Send (... , (rank+1)%P, ...);
  }
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```


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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Isend (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    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[1],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD, &request1);
  MPI_Irecv (&nextbuffer[nlocal+1], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &request2);
  MPI_Isend (&nextbuffer[nlocal], 1, MPI_INT, (rank+1)%P,
    tag, MPI_COMM_WORLD, &request3);
  MPI_Irecv (&nextbuffer[0],1,MPI_INT,(rank+P-1)%P,
    tag, MPI_COMM_WORLD, &request4);
  tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}
```

- Synchronization (e.g. MPI_Wait, MPI_Test) is needed at each iteration (but too much synchronization 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 decide if the execution speed is not acceptable

Symptoms:

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

Causes:

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

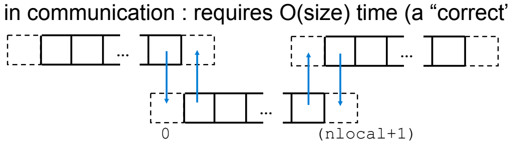
- Complicated communication pattern - does not cause deadlock

What are the defects?

What are the bugs?

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

- Serialization in 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

Discussion

- What are good and bad things about using MPI?
- What can be done to help prevent these defect patterns?
  - If MPI and the source language (C, Fortran) are fixed?
  - If these can be changed?