Debugging Parallel Programs
Debugging Parallel Programs is Hard-er

- Parallel programs are subject to the usual bugs
- Plus: new timing and synchronization errors
- And: parallel bugs often disappear when you add code to try to identify the bug
Visual Debugging of Parallel Programs

● A global view of the multiprocessor architecture
  ■ Processors and communication links
● See which communication links are used
  ■ Perhaps even change the data in transmission
● Utilization of each processor
  ■ Can identify blocked processors, deadlock

● “step” through functionality?
  ■ Lack of a global clock

● Likely won’t help with data races
Debugging Parallel Programs

- Commercial debuggers
  - TotalView, ...

- The printf approach

- gdb, MPI gdb, ppu/spu gdb, ...

- Research debuggers
  - StreamIt Debugger, ...
Pattern-based Approach to Debugging

● “Defect Patterns”: common kinds of bugs in parallel programs
  ■ Useful tips to prevent them
  ■ Recipes for effective resolution

● Inspired by empirical studies at University of Maryland
  ■ http://fc-md.umd.edu/softwareday//presentations/Session0/Keynote.pdf

● At the end of this course, will try to identify some common Cell defect patterns based on your feedback and projects
Defect Pattern: Erroneous Use of Language Features

● **Examples**
  - Inconsistent parameter types for get/send and put/receive
  - Required function calls
  - 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, alignment issues

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

● **Prevention**
  - Check unfamiliar language features carefully
Does Cell have too many functions?

- Yes! But you may not need all of them
- Understand a few basic features

**Spe:**
- spe_create_thread
- spe_wait
- spe_write_in_mbox
- spe_stat_in_mbox
- spe_read_out_mbox
- spe_stat_out_mbox
- spe_write_signal
- spe_get_ls
- spe_get_ps_area
- spe_mfc_get
- spe_mfc_put
- spe_mfc_read_tag_status
- spe_create_group
- spe_get_event

**Mfc:**
- mfc_get
- mfc_put
- mfc_stat_cmd_queue
- mfc_write_tag_mask
- mfc_read_tag_status_all/any/immediate

**Spu:**
- spu_read_in_mbox
- spu_stat_in_mbox
- spu_write_out_mbox, spu_write_out_intr_mbox
- spu_stat_out_mbox, spu_stat_out_intr_mbox
- spu_read_signal1/2
- spu_stat_signal1/2
- spu_write_event_mask
- spu_read_event_status
- spu_stat_event_status
- spu_write_event_ack
- spu_read_decrementer
- spu_write_decrementer

Dr. Rodric Rabbah © Copyrights by IBM Corp. and by other(s) 2007 10 6.189 IAP 2007 MIT
Defect 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

● Cause
  ■ Mapping in parallel version can be different from that in serial version
    − Array origin is different in every processor
    − Additional memory space for communication can complicate the mapping logic

● Prevention
  ■ Validate memory allocation carefully when parallelizing code
Example Problem

A sequence of $N$ cells

\[
\begin{array}{cccccccccc}
2 & 1 & 6 & 8 & 7 & 1 & 0 & 2 & 4 & 5 & 1 & \ldots & 3
\end{array}
\]

- $N$ cells, each of which holds an integer $[0..9]$
  - $cell[0]=2$, $cell[1]=1$, $\ldots$, $cell[N-1]=3$
- In each step, cells are updated using values of neighboring cells
  - $cellnext[x] = (cell[x-1] + cell[x+1]) \mod 10$
  - $cellnext[0]=(3+1)$, $cellnext[1]=(2+6)$, $\ldots$
  - Assume the last cell is connected to the first cell
- Repeat for $steps$ times
Sequential Implementation

- Approach to implementation
  - Use an integer array `buffer[]` for current cell values
  - Use a second array `nextbuffer[]` to store the values for next step
  - Swap the buffers
/* 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 processor keeps 1/size cells
  - size = number of processors

- Each processor needs to:
  - update the locally-stored cells
  - exchange boundary cell values between neighboring processes
Decomposition

Where are the bugs?

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;
}
Where are the bugs?

Decomposition

\[ n_{local} = \frac{N}{size}; \quad N \text{ may not be divisible by } size \]

```c
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;
}
```

Diagram:
```
buffer[]
[------------------]
\[0\] [\[\ldots\]\[n_{local}+1\]]
```
Defect 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

- Preventions
  - Make sure that all communication is correctly coordinated
Communication

Where are the bugs?

/* 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 */
    receive (&nextbuffer[0], (rank+size-1)%size);
    send (&nextbuffer[nlocal], (rank+1)%size);
    receive (&nextbuffer[nlocal+1], (rank+1)%size);
    send (&nextbuffer[1], (rank+size-1)%size);
    tmp = buffer; buffer = nextbuffer; nextbuffer = tmp;
}

● Deadlock

Dr. Rodric Rabbah © Copyrights by IBM Corp. and by other(s) 2007
Modes of Communication

● Recall there are different types of sends and receives
  ■ Synchronous
  ■ Asynchronous
  ■ Blocking
  ■ Non-blocking

● Tips for orchestrating communication
  ■ Alternate the order of sends and receives
  ■ Use asynchronous and non-blocking messages where possible
Defect Pattern: Side-effect of Parallelization

- Ordinary serial constructs may have unexpected side-effects when they used concurrently

- Symptoms
  - Various correctness and performance problems

- Causes
  - Sequential part of code is overlooked
  - Typical parallel programs contain only a few parallel primitives, and the rest of the code is a sequential program running many times

- Prevention
  - 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 in SPMD Program

Where are the bugs?

/* 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 */
...

22 6.189 IAP 2007 MIT
Dr. Rodric Rabbah © Copyrights by IBM Corp. and by other(s) 2007
Data I/O in SPMD Program

Where are the bugs?

/* 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 */
...

- File system may cause performance bottleneck if all processors access the same file simultaneously
- Schedule I/O carefully
Data I/O in SPMD Program

Where are the bugs?

/* Initialize cells with input file */
if (rank == MASTER) {
    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*/
}

- Often only one processor (master) needs to do the I/O
Generating Initial Data

Where are the bugs?

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

/* Main loop */
...

Generating Initial Data

/* 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 processors might use the same pseudo-random seed (and hence sequence), spoiling independence
- Hidden serialization in rand() causes performance bottleneck
Defect Pattern: Performance Scalability

● Symptoms
  ■ Sub-linear scalability
  ■ Performance much less than expected
  ■ Most time spent waiting

● Causes
  ■ Unbalanced amount of computation
  ■ Load balancing may depend on input data

● Prevention
  ■ Make sure all processors are “working” in parallel
  ■ Profiling tools might help
Summary

● Some common bugs in parallel programming
  ■ Erroneous use of language features
  ■ Space decomposition
  ■ Side-effect of parallelization
  ■ Synchronization
  ■ Performance scalability

● There are other kinds of bugs as well: data race
Comment on Data Race Detection

- **Trace analysis can help**
  - Execute program
  - Generate trace of all memory accesses and synchronization operations
  - Build a graph of orderings (solid arrows below) and conflicting memory references (dashed lines below)
  - Detect races (when two nodes connected by dashed lines are not ordered by solid arrows)

- **Intel Thread Checker is an example**
  - More tools available for automatic race detection
Trend in Debugging Technology

- Trace-based
- Checkpointing
- Replay

One day… you’ll have the equivalent of TiVo for debugging your programs