TECHNISCHE UNIVERSITÄT DRESDEN

Center for Information Services and High Performance Computing (ZIH)

Typical Bugs in parallel Programs

Parallel Programming Course, Dresden, 8.- 12. February 2016

Joachim Protze (protze@rz.rwth-aachen.de)

Matthias Lieber (matthias.lieber@tu-dresden.de)

Tobias Hilbrich (tobias.hilbrich@tu-dresden.de)

Center for Information Services & High Performance Computing

Some Common Serial Bugs

- Wrong memory access

Invalid access

```
int *p;
p[100] = 123; //Access to non-allocated memory
```

Undefined read

```
int i,n;
for (i = 0; i < n; i++) //value of n undefined
  …
```

- Arithmetic errors

```
int x, y=1000000;
x = y*y;                    //32bit Integer overflow
```

- Memory leaks

```
int do_work (int size)
{
    int *x;
    x = (int*) malloc (sizeof(int) * size);        //Memory never freed
}
```

- Erroneous usage of library interfaces

```
FILE *file;
file = fopen (“myfile.txt”, “rwa”);            //Invalid open mode
```

Heisenbugs

- A class of bugs that only manifests in certain runs of an application
- In worst case it may never occur in the presence of a debugging tool
- May result from:
  - Usage of uninitialized memory
  - Data races (a parallel problem, see later)
- Example:

```
int a[100], i;
int *p;
for (i = 0; i < 100; i++)
{
    if (a[i] > 1234)
        p[i] = 5678;
}
```

// a[i] might be > 1234 in some application run

- Very hard to track and identify as they are not easily reproducible

Parallel Bugs

- All serial bugs may also appear in a parallel application
- Usage of threads/processes introduces new classes of errors
  - Races
  - Deadlocks
- Communication with MPI introduces new classes of errors
  - Overlapping buffers (potential races)
  - Type mismatches (potential data trash)
  - Leaks of MPI resources (potential MPI error)
  - Various ways to produce deadlocks

Parallel Bugs – Race

- Race: Program behavior dependent on execution order of threads/processes due to unsynchronized write access to a shared state (e.g. variable)
- Example:

```
int x,y;
#pragma omp parallel
{
    x = omp_get_thread_num (); //write-write race (last writer wins)
    #omp barrier
    #omp master
        printf ("Master is:%d" ,x);
}
```

- Races are a frequent cause of heisenbugs, e.g.:

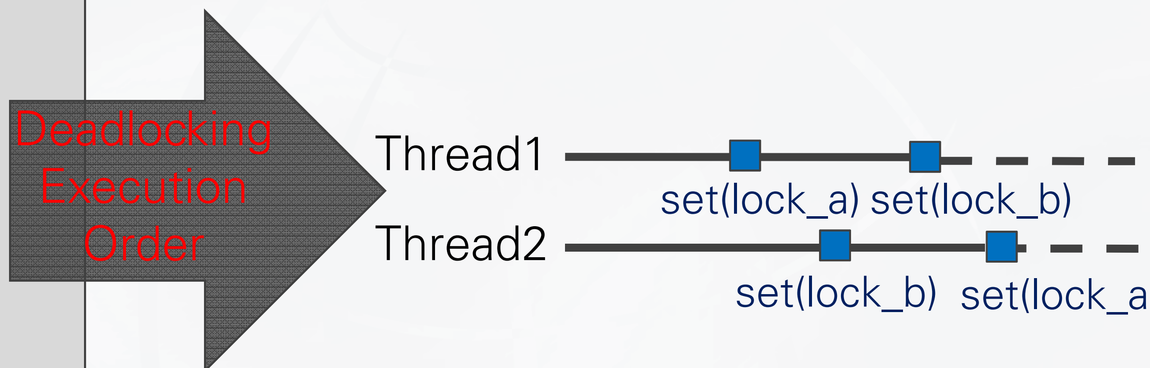
```
int x,y;
#pragma omp parallel
{
    #omp master
        compute_something();
    x = omp_get_thread_num ();
    #omp barrier
    #omp master
        printf ("Master is:%d" ,x);
}
```

// due to the computation the master almost always wins, but it is not guaranteed

Parallel Bugs – Deadlock

- Deadlock: A circular wait condition exists in the system that causes two or more parallel units to wait indefinitely
- In other words: “The application hangs …”
- Example:

```
#pragma omp parallel sections
{
    #omp section
    {
        omp_set_lock(&lock_a);
        omp_set_lock(&lock_b);
        omp_unset_lock (&lock_b);
        omp_unset_lock (&lock_a);
    }
    #omp section
    {
        omp_set_lock(&lock_b);
        omp_set_lock(&lock_a);
        omp_unset_lock (&lock_a);
        omp_unset_lock (&lock_b);
    }
}
```



// Thread1 waits for lock_b owned by Thread2, whereas Thread2 waits for lock_a owned by Thread1. Thus, neither thread can free a lock and both threads wait indefinitely.

MPI Usage Errors – Buffer Overlaps

- MPI standard: Memory regions passed to MPI must not overlap (except send-send)
- Complications
  - Derived data types may span non-contiguous regions
  - Collectives may both send and receive
- Examples:

Isend overlaps element buf[4] from the Irecv call!

```
MPI_Isend (&(buf[0])/*buf*/, 5/*count*/, MPI_INT,...);
MPI_Irecv (&(buf[4])/*buf*/, 5/*count*/, MPI_INT,...);
```

Recvbuf overlaps element buf[4] from the sendbuf!

```
MPI_Allreduce (&(buf[0])/*sendbuf*/,
       &(buf[4])/*recvbuf*/, 5/*count*/, MPI_INT,...);
```

MPI Usage Errors – Type Matching

- Example 1:
  - Consider type T1 = {MPI_INT, MPI_INT}

Rank 0	Rank 1
`MPI_Send (buf, 1, T1)`	`MPI_Recv (buf, 2, MPI_INT)`

No Error, types match

- Example 2:
  - T1 = {MPI_INT, MPI_FLOAT}
  - T2 = {MPI_INT, MPI_INT}

Rank 0	Rank 1
`MPI_Send (buf, 1, T1)`	`MPI_Recv (buf, 1, T2)`

Missmatch: MPI_FLOAT != MPI_INT

MPI Usage Errors – Resource Usage

- MPI uses opaque objects for communicators, requests, groups, data types, windows, operations, …
- Memory for these objects is allocated by the MPI library
- Complications
  - Amount of memory per object is not clear and depends on MPI implementation
  - Memory leaks
  - MPI internal limits may lead to MPI error messages and abort
- Example:

Applications should complete the outstanding communication associated with *request*

```
for(i=0; i<10000; ++i)
  MPI_Isend (..., &request);
MPI_Finalize ();
```

MPI Usage Errors – Deadlocks

- Various ways to create deadlocks with MPI:
  - Not all ranks call the same collective operation
  - Complex completions, e.g. Wait{all, any, some}
  - Non-determinism, e.g. MPI_ANY_SOURCE, MPI_ANY_TAG
  - Choices of implementation in MPI standard (e.g. MPI_Send might be blocking or non-blocking)
- Example:

Rank 0

```
MPI_Send (to:1)
MPI_Recv (from:1)
```

Rank 1

```
MPI_Send (to:0)
MPI_Recv (from:0)
```

Potential deadlock: ranks wait for each other

Avoiding Bugs

- Think and program, don’t hack !
- Focus on writing code, not on deciphering it
  - Write comments (short but helpful ones)
  - Use descriptive names, stick to a coding style
  - Use a helpful and consistent indentation
- Use programming techniques, e.g.:
  - Code reviewing
  - Pair programming (one programmer codes, other comments & reviews)
  - Check for pre/post conditions, e.g. *assert(pointer != NULL)*
- Think about an verbose execution mode of your code
  - The outputs give helpful hints on were an application is buggy

Find the Bug Early

- Use compiler flags for compile-time and run-time checks (like –Wall)
  - May detect syntax errors, portability errors, invalid reads
  - Consult your compiler's manual!
- Check your code periodically with runtime tools, at least before production runs
  - Memory debugging tools detect memory leaks and invalid memory accesses
    - Valgrind (no MPI support, but free software): *valgrind ./a.out*
    - Allinea DDT (serial, OpenMP, MPI)
  - Check OpenMP parallelization for races and (potential) deadlocks
    - Intel Inspector XE
  - Detect MPI usage errors and (potential) deadlocks
    - MUST

Find the Bug Early: Call Stack Traceback

- In case of a program crash, a simple call stack traceback sometimes is sufficient to find the reason
- Producing a traceback is system-dependent (compiler / MPI library)
- For example, on Taurus with BullxMPI you might get:

```
[taurusi4002:22098:0] Caught signal 11 (Segmentation fault)
==== backtrace ====
 2 0x00000000000548cc mxm_handle_error() debug.c:641
 3 0x000000000054a3c mxm_error_signal_handler()  debug.c:616
 4 0x0000003b332326a0 killpg()  ??:0
 5 0x0000000000401997 heatAllocate()  heatC-MPI-01.c:39
 6 0x0000000000402e4b main()  heatC-MPI-01.c:432
 7 0x0000003b3321ed5d __libc_start_main()  ??:0
 8 0x0000000000401769 _start()  ??:0
```

Signal handler in MPI library

Line 39 causes the crash

Libc calls main()

- Need to compile with –g to get source code location (if not, you get `??:0`)
- In case you only get an address (usually hex number like `0x00401997`):
  - `addr2line -e <executable> <address>` tells you the location

Getting more Information from the Intel Compiler

- Common flags:
  - -g (produce debug information)
  - -O0 (disable optimization)
- Intel C Compiler compile-time information:
  - -Wall (enable almost all warnings)
  - -Wuninitialized (check for uninitialized variables – not reliable!)
  - -std=c89 / -std=c99 / -std=c++11 (strictly conform to C/C++ standard)
- Intel Fortran Compiler compile-time information:
  - -warn all (enable all warnings)
  - -std90 / -std95 / -std03 / -std08 (strictly conform to Fortran standard)
- Intel Fortran Compiler run-time information:
  - -traceback (call stack traceback when severe error occurs)
  - -check (run-time checking, e.g. array bounds, uninitialized variables)
  - -fpe0 (abort on floating point exceptions, e.g. division by zero, overflow)