Matthias Lieber
Center for Information Services and High Performance Computing (ZIH)

Introduction to Parallel Debugging

Parallel Programming with MPI, OpenMP, and Tools
Dresden, 8-12 February 2021

Acknowledgements for today's Parallel Debugging part:
Bettina Krammer, Rolf Rabenseifner, Joachim Protze,
Tobias Hilbrich, Mathias Korepkat

Center for Information Services &
High Performance Computing

Outline

Typical Bugs in parallel Programs

- Serial Bugs
- Parallel Bugs

How to deal with Bugs?

- Avoiding bugs, compiler options, tools

Serial Bugs

Memory access error / Segmentation fault

```
int *p;
p[100] = 123;
```

Undefined memory

```
int i, n;
for(i = 0; i < n; i++)
   // n = ??
```

Arithmetic error, e.g. overflow, division by zero, etc.

```
int x, y = 1000000;
x = y * y;
```

Memory leak

```
for(int i=0;i<200;++i)
{
  int *x = new int[100];
}
```

```
integer :: i
integer, pointer :: x(:)
do i=1,200
  allocate(x(100))
end do
```

Incorrect library usage

```
FILE *f;
f = fopen("file.txt","R");
```

Parallel Bugs

All serial bugs may also appear in parallel programs

Parallelism introduces two new classes of errors

- Data race: leads to non-deterministic behavior
- Deadlock: applications “hangs”

New ways to produce the known types of bugs with MPI and OpenMP

- Undefined memory
- Memory leaks
- Incorrect library usage

Parallel Bugs: Data Race

Data Race

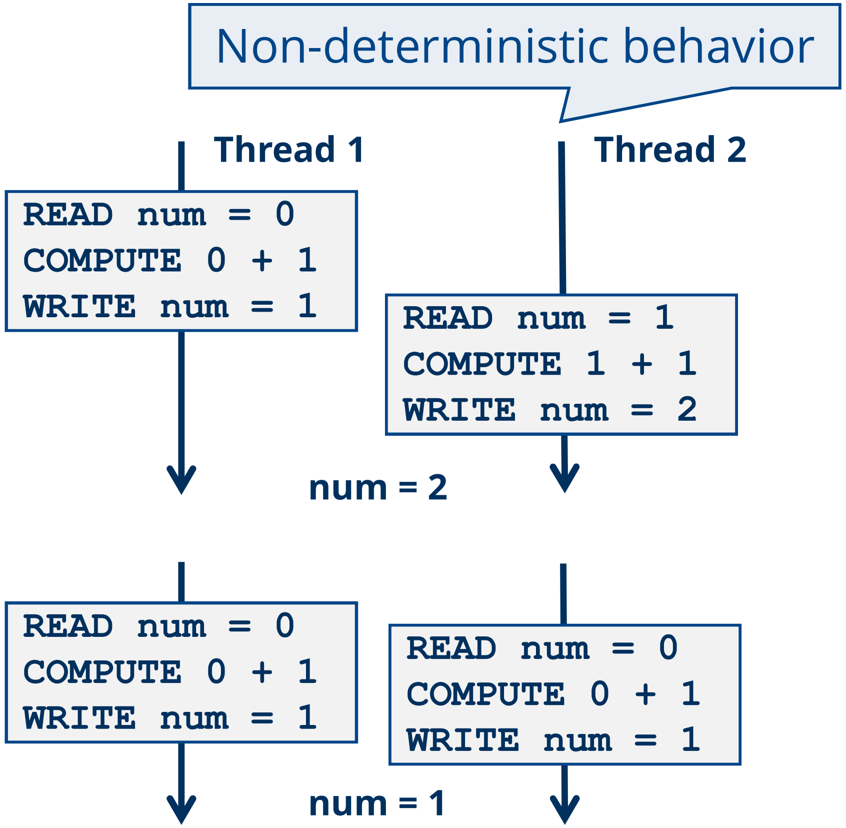
— Program behavior dependent on the execution order of potentially concurrent operations on the same shared resource.

Data Race with OpenMP

— Two threads access the same shared variable without ensuring a specific order and at least one thread modifies the variable

Example

```
int num = 0;
#pragma omp parallel
num = num + 1;
printf ("num = %d\n",num);
```



Parallel Bugs: Data Race

Data Race with MPI: Buffer overlaps

— MPI standard: Memory regions passed to MPI must not overlap (except when only used for sending)

Caution

— Derived data types may span non-contiguous regions: hard to identify

— 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, ...);
```

Parallel Bugs: Deadlock

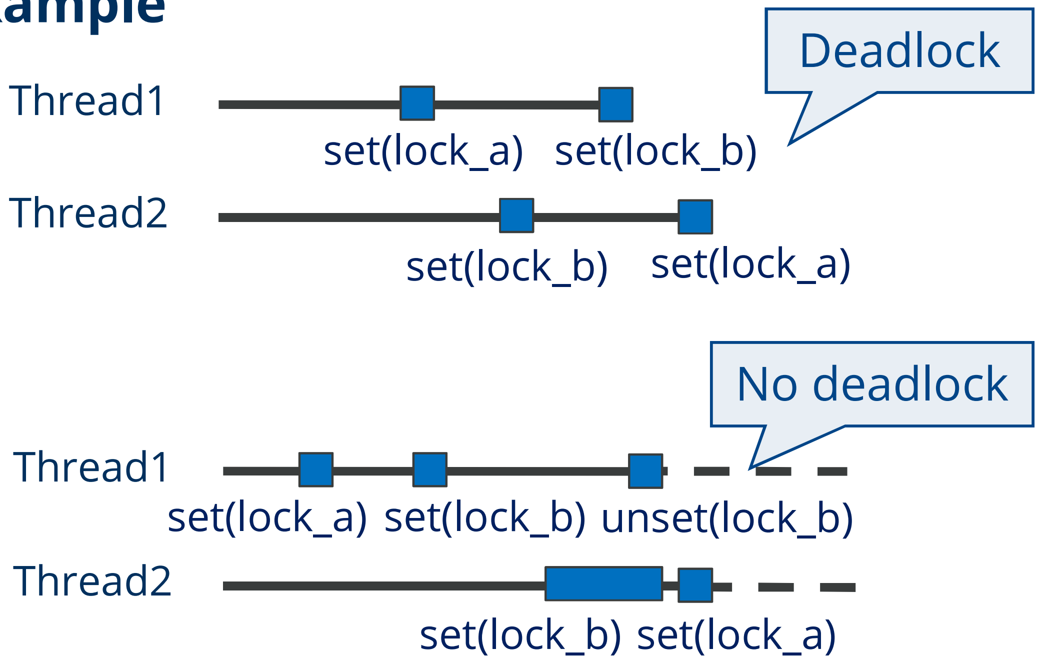
Deadlock

— Threads/processes wait infinitely for each other to release resources (e.g. locks, messages) while holding the resource the others are waiting for.

Deadlock with OpenMP

— Caution when using locks!

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);
  }
}
```

Parallel Bugs: Deadlock

Deadlocks with MPI

— Not all ranks in the communicator call the same collective operation

— Receive/wait without matching send (e.g. tag, dest, or comm not matching)

— Blocking (synchronous) send without matching receive

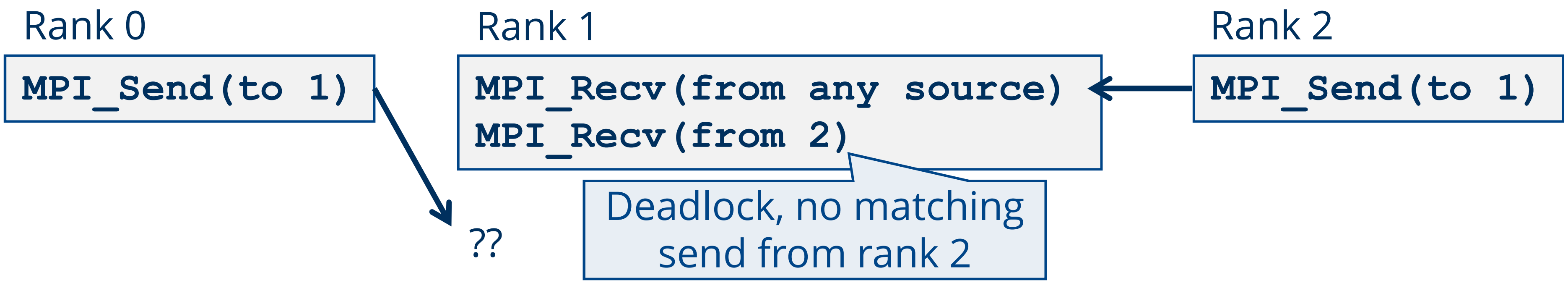
Caution

— Complex completions, e.g. Wait{all, any, some}

— Non-determinism, e.g. MPI_ANY_SOURCE, MPI_ANY_TAG

— MPI_Send might behave synchronous or non-synchronous

Example



Parallel Bugs: Undefined Memory with OpenMP

OpenMP private and undefined Memory

- Private variables are not initialized and not updated after the region
- Use firstprivate and lastprivate

Example

```
int num = 100;
#pragma omp parallel for private(num)
for(int i=0; i<100; ++i)
{
  if(i==99)
  {
    num = num + 42;
    printf("within omp for: num = %d\n",num);
  }
}
printf("after omp for: num = %d\n",num);
```

num is not initialized here

Prints trash, not 142 as in serial execution

Prints 100, not 142 as in serial execution

Solution

```
#pragma omp parallel for firstprivate(num) lastprivate(num)
```

Parallel Bugs: Undefined Memory with MPI

MPI Type Mismatch

- If sender and receiver datatype do not match, the result is unspecified
- Caution: MPI library may report the error, crash, or continue with corrupted memory in receive buffer

Examples

No error, data types match

Rank 0

```
// t0 = {MPI_INT, MPI_INT}
MPI_Type_contiguous(2, MPI_INT, &t0);
MPI_Type_commit(&t0);
MPI_Send(buf, 1, t0, 1, ...);
```

Rank 1

```
MPI_Recv(buf, 2, MPI_INT, 0, ...);
```

MPI_INT != MPI_FLOAT, typically not detected by MPI

Rank 0

```
// t0 = {MPI_INT, MPI_FLOAT}
MPI_Send(buf, 1, t0, 1, ...);
```

Rank 1

```
// t1 = {MPI_INT, MPI_INT}
MPI_Recv(buf, 1, t1, 0, ...);
```

Parallel Bugs: Undefined Memory with MPI

Messages shorter than receive buffer

— Receive buffer is allowed to be larger than actual message received

Caution

— Accidently sending less data than intended is not detected; parts of the receive buffer will not be written and may contain uninitialized memory

Examples

No errors in both examples,
Caution: last element(s) in receive buffer will not be written!

Rank 0

```
// t0 = {MPI_INT, MPI_INT}
MPI_Send(buf, 1, t0, 1, ...);
```

Rank 1

```
MPI_Recv(buf, 3, MPI_INT, 0, ...);
```

Rank 0

```
// t0 = {MPI_INT, MPI_FLOAT}

MPI_Send(buf, 1, t0, 1, ...);
```

Rank 1

```
// t1 = {MPI_INT, MPI_FLOAT,
//       MPI_LONG, MPI_DOUBLE}
MPI_Recv(buf, 1, t1, 0, ...);
```

Parallel Bugs: Memory Leaks with MPI

MPI Opaque Objects

— Used for communicators, requests, data types, windows, operations, …

— MPI allocates and frees memory for these objects on user's request, e.g. by calling MPI_Type_vector and MPI_Type_free

Caution

— Memory per object is not clear and depends on MPI implementation

— Losing handles to objects leads to memory leaks

— MPI internal limits may lead to MPI error messages and abort

Example

```
for(i=0; i<100000; ++i)
{
  MPI_Request request;
  MPI_Isend(..., &request);
}
```

User is responsible to free the request, either with a wait call or MPI_Request_free

Parallel Bugs: Wrong MPI Library Usage

MPI 3.1 Standard: 800+ pages

- Does your application conform to the MPI standard?
- E.g. complex calls like MPI_Alltoallw with 9 arguments, including 6 communicator sized arrays offer many opportunities for bugs

Examples

```
MPI_Type_contiguous(2, MPI_INTEGER, &newtype);
MPI_Send(buf, 1, newtype, dest, tag, MPI_COMM_WORLD);
```

Works with many implementations, but has two bugs

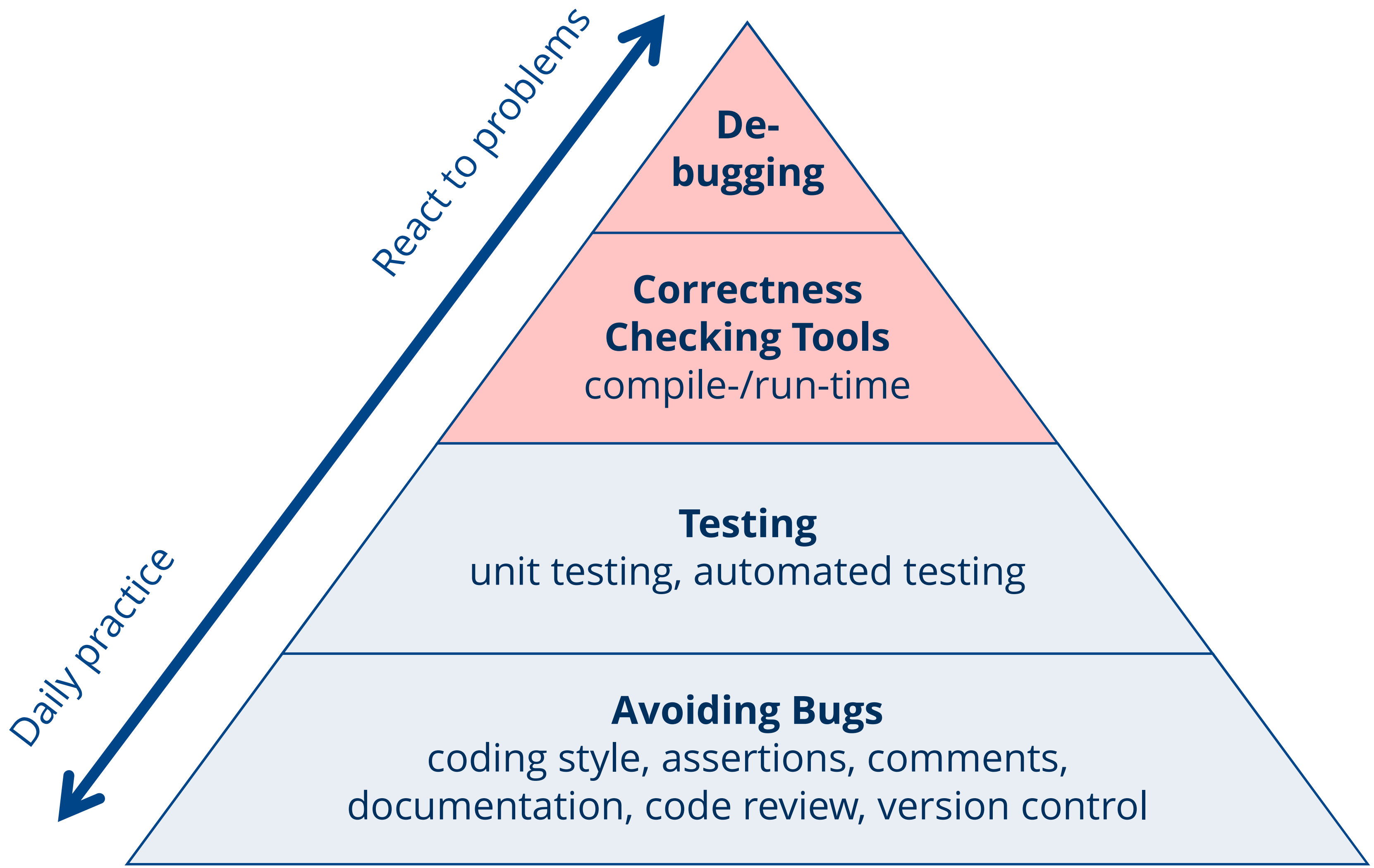
```
// Rank 0 of 2
MPI_Alltoallw( sbuf, {4, 2}, sdispl, {MPI_INT, MPI_INT},
               rbuf, {2, 2}, rdispl, {MPI_FLOAT, MPI_INT}, comm );
```

Inconsistency: sends 4 MPI_INT to itself, but receives 2 MPI_FLOAT from itself

Parallel Bugs: Summary

	MPI	OpenMP
Data race	Buffer overlap of concurrent comm operations, at least one receive	Concurrent access to shared variable, at least one write
Deadlock	Synchronizing communication calls without matching remote call	Risk when setting multiple locks at the same time
Undefined memory	MPI type mismatch; messages shorter than receive buffer	Inappropriate private; missing firstprivate or lastprivate
Memory leak	Not freeing MPI opaque objects	
Incorrect library usage	High potential of errors not clearly visible at compile-time or run-time	Less risk of hidden errors, more checks by compiler

How to Deal with Bugs?



Correctness Checking Tools

Compiler

- Compilers enable different compile-time and run-time checks
- Scope depends on compiler - consult your compiler's manual
- E.g. arithmetic errors, portability errors, memory errors, etc.

Memory error detection tools

- Detect memory leaks and invalid memory accesses
- Intel compiler: array bounds and pointer checking (see next slide)
- Valgrind: free software, but no MPI support, just run valgrind ./a.out
- DDT: includes "memory debugging" feature

Parallelization checking tools

- MUST: detects various MPI usage errors and (potential) deadlocks
- Intel Inspector: detects OpenMP data races and (potential) deadlocks

Compiler Flags (Intel Compiler 18.0.1)

		C/C++ Compiler	Fortran Compiler
Basics	Add debug info	–g	
	No optimization	–O0 (for debugging only, not production)	
Compile-time checks	Enable all warnings	–w3	–warn all
	Language standard conformance	–std=c99 / –std=c11 / –std=c++11 / –std=c++14 / …	–std90 / –std95 / –std03 / –std08 / –std15
Run-time checks	Floating point arithmetic errors	–fp–trap=all	–fpe–all0
	Array bounds and pointer accesses	–check–pointers=rw	–check pointers,bounds
	Further run-time checks	–check={stack,conversions, uninit}	–check {stack,uninit, format, …}

Large run-time overhead, not for production runs!

Call Traceback / Backtrace

Program aborts, but you don't know where?

- A call stack traceback gives you the location (source code line) of the error
- This is often sufficient to solve the problem

Getting a traceback

- –g –traceback (Intel Fortran only, though C compiler accepts –traceback)

```
% mpif90 -g -traceback -O0 heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
[...]
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
heatF-MPI-01       00000000004079CD  Unknown               Unknown  Unknown
libpthread-2.17.s  00007F9327C855E0  Unknown               Unknown  Unknown
heatF-MPI-01       0000000000404933  heatconduction_mp         192  heatF-MPI-01.F90
heatF-MPI-01       00000000004066E7  MAIN__                    494  heatF-MPI-01.F90
heatF-MPI-01       000000000040342E  Unknown               Unknown  Unknown
libc-2.17.so       00007F93275D2C05  __libc_start_main     Unknown  Unknown
heatF-MPI-01       0000000000403329  Unknown               Unknown  Unknown
% addr2line -e ./heatF-MPI-01 0000000000404933
/home/gpu59/Debugging/f90/heatF-MPI-01.F90:192
```

Check line 192

In case line numbers are not shown in table

Practical: Compiler Flags vs. Segmentation Fault

Run the commands below and observe the output of each srun. How helpful are the error messages to identify the source code line that causes the segmentation fault? Play with **`addr2line`** (see previous slide) if you feel it is required.

Optional, if you have the time: try both C and Fortran.

C:

```
% cd ~/Debugging/c
% mpicc -g -O0 heatC-MPI-01.c -o heatC-MPI-01
% srun -n 4 ./heatC-MPI-01
% mpicc -g -O0 -check-pointers=rw heatC-MPI-01.c -o heatC-MPI-01
% srun -n 4 ./heatC-MPI-01
```

`--export ALL` not required any more (changed environment variable)

Fortran 90:

```
% cd ~/Debugging/f90
% mpif90 -g -O0 heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
% mpif90 -g -O0 -traceback heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
% mpif90 -g -O0 -traceback -check pointers,bounds
        heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
```

Practical: C Version

```
% mpicc -g -O0 heatC-MPI-01.c -o heatC-MPI-01
% srun -n 4 ./heatC-MPI-01
srun: error: taurusi6447: tasks 0-3: Segmentation fault



% mpicc -g -O0 -check-pointers=rw heatC-MPI-01.c -o heatC-MPI-01
% srun -n 4 --export ALL ./heatC-MPI-01
CHKP: Bounds check error ptr=(nil) sz=8 lb=(nil) ub=(nil) loc=0x402e98
Traceback:
    at address 0x402e98 in function heatAllocate
    in file /home/h8/gpu59/Debugging/c/heatC-MPI-01.c line 34
    at address 0x40c44c in function main
    in file /home/h8/gpu59/Debugging/c/heatC-MPI-01.c line 432
    at address 0x7ffb710b4555 in function __libc_start_main
    in file unknown line 0
    at address 0x402529 in function _start
    in file unknown line 0

[...]
% addr2line -e heatC-MPI-01 0x402e98
/home/gpu59/Debugging/c/heatC-MPI-01.c:34
```

Segmentation fault – but where?

Pointer checking detected NULL pointer access

Traceback shows location of error

In case traceback is not shown, address is **loc** from above

Practical: Fortran Version

```
% mpif90 -g -O0 heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
[...]
forrtl: severe (174): SIGSEGV, segmentation fault occurred
Image              PC                Routine            Line        Source
heatF-MPI-01       00000000004079CD  Unknown               Unknown  Unknown
libpthread-2.17.s  00002BA82CD55630  Unknown               Unknown  Unknown
heatF-MPI-01       0000000000404933  Unknown               Unknown  Unknown
heatF-MPI-01       00000000004066E7  Unknown               Unknown  Unknown
heatF-MPI-01       000000000040342E  Unknown               U
libc-2.17.so       00002BA82D286555  __libc_start_main     U
heatF-MPI-01       0000000000403329  Unknown               U
% addr2line -e heatF-MPI-01 0000000000404933
/home/gpu59/Debugging/f90/heatF-MPI-01.F90:192

% mpif90 -g -O0 -traceback -check pointers,bounds heatF-MPI-01.F90 -o heatF-MPI-01
% srun -n 4 ./heatF-MPI-01
forrtl: severe (408): fort: (7): Attempt to use pointer THETANEW when it is not
associated with a target
Image              PC                Routine            Line        Source
heatF-MPI-01       0000000000410550  Unknown               Unknown  Unknown
heatF-MPI-01       0000000000403B8F  heatconduction_mp          71  heatF-MPI-01.F90
heatF-MPI-01       000000000040D6BC  MAIN__                    471  heatF-MPI-01.F90
heatF-MPI-01       000000000040342E  Unknown                        Unknown
libc-2.17.so       00002B1
heatF-MPI-01       000000
```

Segmentation fault – but where?

Trying out some addresses from above results in line 192

Traceback shows location of error – due to check option, the bug is detected earlier as above

Serial and Parallel Bugs: Summary with Tools

	Serial	MPI	OpenMP
Data race		MUST	Inspector
Deadlock			
Undefined memory	Compiler, DDT, Valgrind		
Memory access error			
Memory leak	DDT, Valgrind		
Incorrect library usage			
Arithmetic error	Compiler		