HPC debugging

Victor Eijkhout

2022

Profiling and debugging; optimization and programming strategies.

1 Analysis basics

- Measurements: repeated and controlled
  beware of transients, do you know where your data is?
- Document everything
- Script everything

2 Compiler options

- Defaults are a starting point
- use reporting options: `-opt-report`, `-vec-report`
  useful to check if optimization happened / could not happen
- test numerical correctness before/after optimization change
  (there are options for numerical corretness)

3 Optimization basics

- Use libraries when possible: don’t reinvent the wheel
- Premature optimization is the root of all evil (Knuth)

4 Code design for performance

- Keep inner loops simple: no conditionals, function calls, casts
- Avoid small functions: try macros or inlining
- Keep in mind all the cache,TLB, SIMD stuff from before
- SIMD: Fortran array syntax helps

5 Multicore / multithread

- Use `numactl`: prevent process migration
- ‘first touch’ policy: allocate data where it will be used
- Scaling behaviour mostly influenced by bandwidth

6 Multinode performance

- Influenced by load balancing
- Use HPCtoolkit, Scalasca, TAU for plotting
- Explore ‘eager’ limit (mvapich2: environment variables)

7 Classes of programming errors

Logic errors:
functions behave differently from how you thought,
or interact in ways you didn’t envision

Hard to debug

8 More classes of errors

Coding errors:
send without receive
forget to allocate buffer

Debuggers can help

Defensive programming

9 Defensive programming

- Keep It Simple (‘restrict expressivity’)
- Example: use collective instead of spelling it out
- easier to write / harder to get wrong
  the library and runtime are likely to be better at optimizing than you

10 Memory management

Beware of memory leaks:
keep allocation and free in same lexical scope

C++ does this automatically with RAII

11 Modular design

Design for debuggability, also easier to optimize

Separation of concerns: try to keep code aspects separate

Premature optimization is the root of all evil (Knuth)

12 MPI performance design

Be aware of latencies: bundle messages
(this may go again separation of concerns)

Consider ‘eager limit’

Process placement, reduction in number of processes

Debugging

13

Debugging is like being the detective in a crime movie where you are also the murderer. (Filipe Fortes, 2013)

What do you do when your program misbehaves?

- Insert print statements, recompile, run again.
- Run your program in a debugger
- (also: attach a debugger, inspect a core dump)

14 Simple example: listing

tutorials/gdb/c/hello.c

```
#include <stdlib.h>
#include <stdio.h>
int main() {
  printf("hello world\n");
  return 0;
}
```

15 Simple example: running

```
%% cc -g -o hello hello.c
# regular invocation:
%% ./hello
hello world
# invocation from gdb:
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... [version info]
Copyright 2004 Free Software Foundation, Inc. .... [copyright info
(gdb) run
Starting program: /home/eijkhout/tutorials/gdb/hello
Reading symbols for shared libraries +. done
hello world

Program exited normally.
(gdb) quit
%%
```

16 Source listing

```
%% cc -o hello hello.c
%% gdb hello
GNU gdb 6.3.50-20050815 # ..... version info
(gdb) list
```

Important to use the `-g` compile option!

17 Run with arguments

tutorials/gdb/c/say.c

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc,char **argv) {
  int i;
  for (i=0; i<atoi(argv[1]); i++)
    printf("hello world\n");
  return 0;
}
```

```
%% gdb say
.... the usual messages ...
(gdb) run 2
Starting program: /home/eijkhout/tutorials/gdb/c/say 2
Reading symbols for shared libraries +. done
hello world
hello world
```

18 Memory problems 1

```
// square.c
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",nmax);
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i); sum += squares[i];
  }
  printf("Sum: %e\n",sum);

%% cc -g -o square square.c
 %% ./square
5000
Segmentation fault
```

The debugger will stop at the problem.

19 Stack trace

Displaying a stack trace	
gdb	lldb
`(gdb) where`	`(lldb) thread backtrace`

```
(gdb) backtrace
#0  0x00007fff824295ca in __svfscanf_l ()
#1  0x00007fff8244011b in fscanf ()
#2  0x0000000100000e89 in main (argc=1, argv=0x7fff5fbfc7c0) at sq
```

20 Inspecting a stack frame

Investigate a specific frame	
gdb	clang
`frame 2`	`frame select 2`

Then `print` variables and such.

21 Out-of-bounds errors

```
// up.c
  int nlocal = 100,i;
  double s, *array = (double*) malloc(nlocal*sizeof(double))
  for (i=0; i<nlocal; i++) {
    double di = (double)i;
    array[i] = 1/(di*di);
  }
  s = 0.;
  for (i=nlocal-1; i>=0; i++) {
    double di = (double)i;
    s += array[i];
  }
```

22 Out of bounds in debugger

```
Program received signal EXC_BAD_ACCESS, Could not access mem
Reason: KERN_INVALID_ADDRESS at address: 0x0000000100200000
0x0000000100000f43 in main (argc=1, argv=0x7fff5fbfe2c0) at
15          s += array[i];
(gdb) print array
$1 = (double *) 0x100104d00
(gdb) print i
$2 = 128608
```

23 Breakpoints

Set a breakpoint at a line	
gdb	lldb
`break foo.c:12`	`breakpoint set [ -f foo.c ] -l 12`

24 Stepping

Stepping through a program		
gdb	lldb	meaning
`run`		start a run
`cont`		continue from breakpoint
`next`		next statement on same level
`step`		next statement, this level or next

Memory debugging

25 Program with problems

tutorials/gdb/c/square1.c

```
#include <stdlib.h>
#include <stdio.h>
//codesnippet gdbsquare1c
int main(int argc,char **argv) {
  int nmax,i;
  float *squares,sum;

  fscanf(stdin,"%d",&nmax);
  squares = (float*) malloc(nmax*sizeof(float));
  for (i=1; i<=nmax; i++) {
    squares[i] = 1./(i*i);
    sum += squares[i];
  }
  printf("Sum: %e\n",sum);
//codesnippet end

  return 0;
}
```

26 Valgrind output

```
%% valgrind square1
==53695== Memcheck, a memory error detector
==53695== [stuff]
10
==53695== Invalid write of size 4
==53695==    at 0x100000EB0: main (square1.c:10)
==53695==  Address 0x10027e148 is 0 bytes after a block of s
==53695==    at 0x1000101EF: malloc (vg_replace_malloc.c:236
==53695==    by 0x100000E77: main (square1.c:8)
==53695==
```

Parallel Debugging

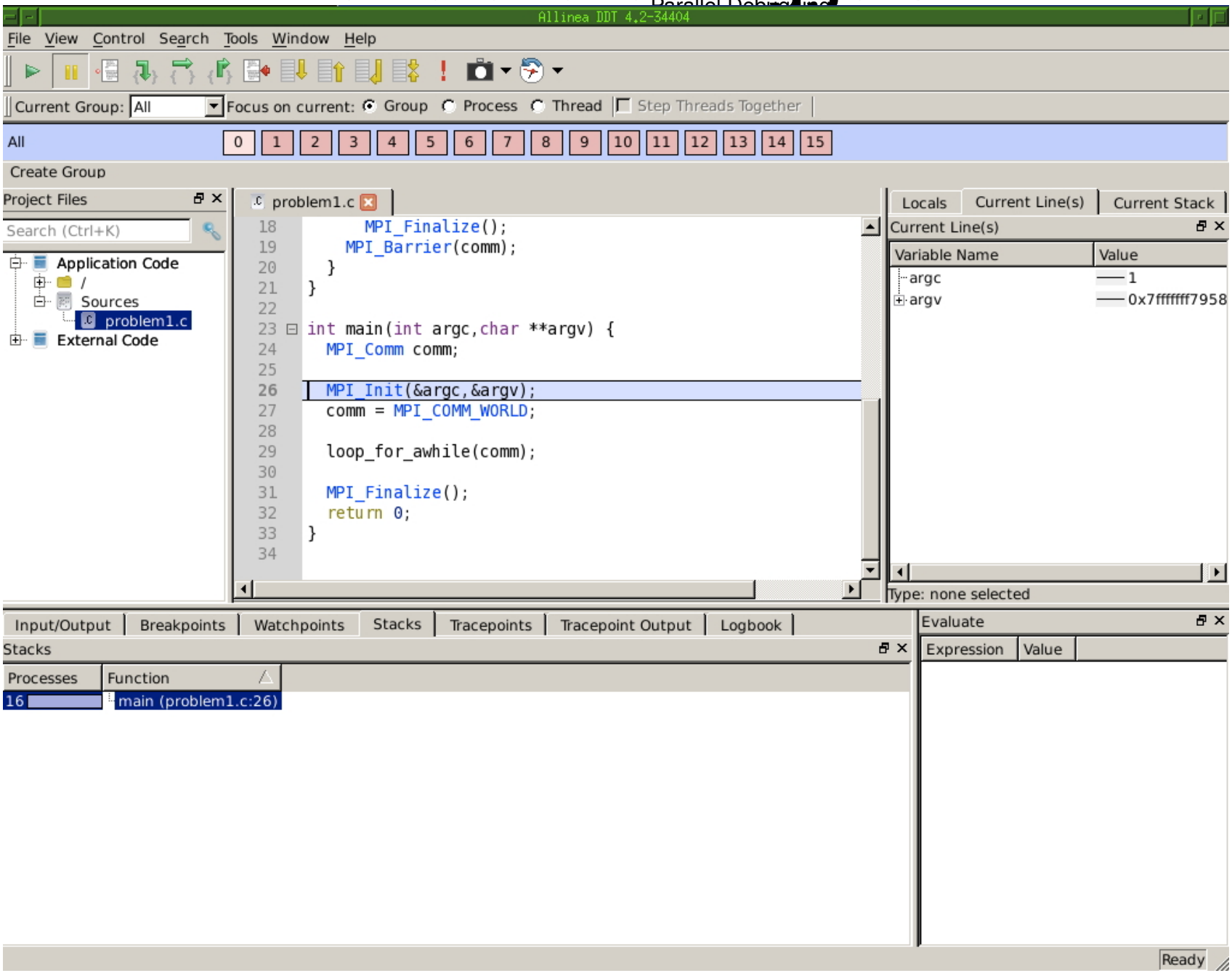
27 Debugging

I assume you know about gdb and valgrind...

- Interactive use of gdb, starting up multiple xterms
  feasible on small scale
- Use gdb to inspect dump:
  can be useful, often a program crashes hard and leaves no dump

Note: compile options `-g -O0`

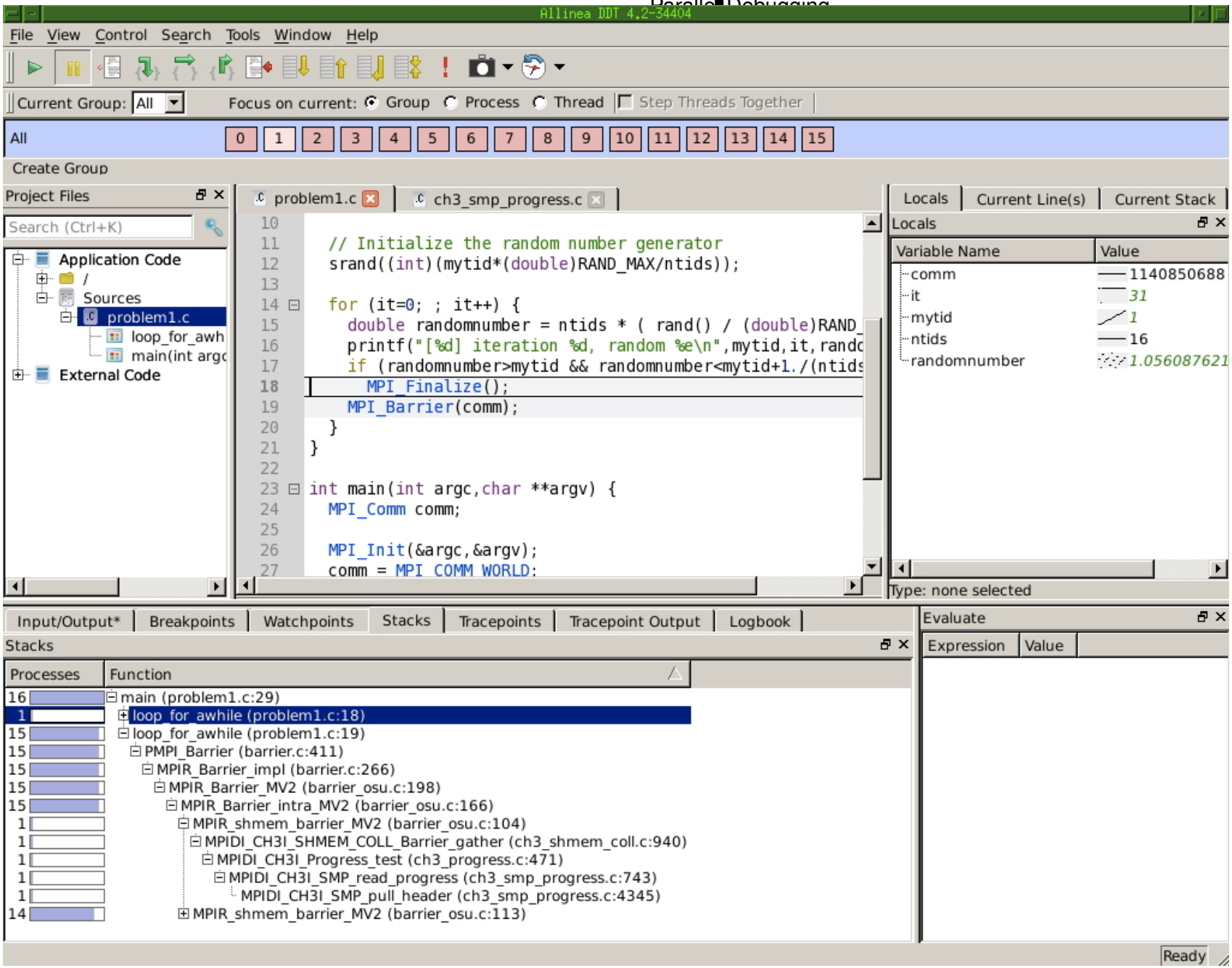
28 Parallel debuggers



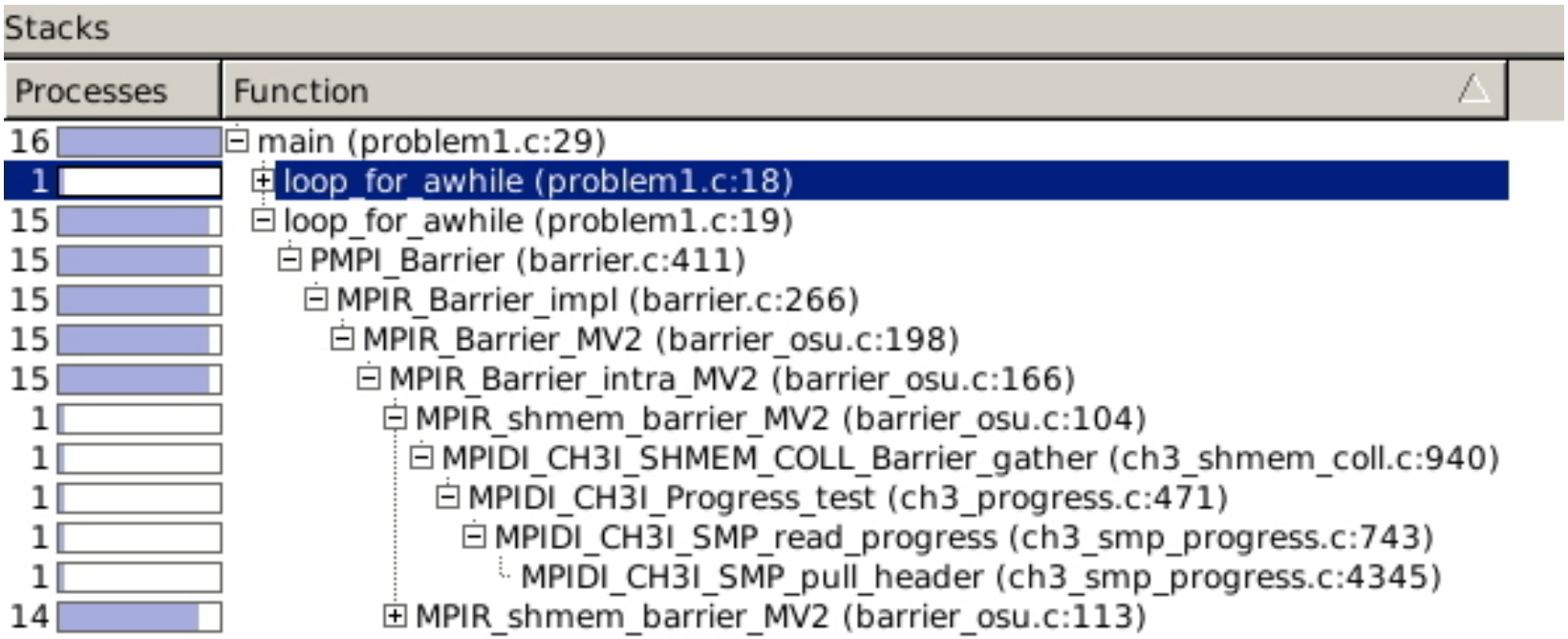
29 Buggy code

```
for (it=0; ; it++) {
  double randomnumber = ntids * ( rand() / (double)RAND_MAX
  printf("[%d] iteration %d, random %e\n",mytid,it,randomnum
  if (randomnumber>mytid && randomnumber<mytid+1./(ntids+1))
    MPI_Finalize();
  MPI_Barrier(comm);
}
```

30 Parallel inspection



31 Stack trace



32 Variable inspection