# Valgrind

Philip Blakely

Laboratory for Scientific Computing, University of Cambridge

Part I

# Valgrind

# Valgrind

- http://valgrind.org/
- Pronunciation: The "Val" as in the word "value". The "grind" is pronounced with a short 'i' – ie. "grinned" (rhymes with "tinned") rather than "grined" (rhymes with "find").
- Origin: Valgrind is the name of the main entrance to Valhalla.
- Valgrind is an emulator, supporting a wide range of modern processors.
- It contains a number of tools from memory usage (amount, uninitialised data) to performance.

# Supported processors

As of Version 3.13 (June 2017):

- x86/Linux: Up to and including SSE3 instructions.
- x86_64 (AMD64/Linux): Up to and including AVX2 instruction.
- ARM64/Linux
- AMD64/Darwin (Mac OS X - 10.9.x and later)

# Memcheck - overview

Valgrind's default tool is Memcheck: `--tool=memcheck`

- All memory allocated via malloc, new, new[] (and similar) is tracked.
- Memory allocated on stack (local variables) is checked for initialization.
- Checks (mostly) heap based memory only.

# Memcheck

Errors detected:

- Access to data outside a heap-allocated region
- Segmentation faults
- Branch statements based on uninitialised data
- Mismatched `new []` and `delete` (for example)

And not detected...

- Floating point exceptions
- Copying or arithmetic operations on uninitialised data (Note that sqrt, sin, etc. involve branches internally and will be picked up.)

# Memcheck - Example

- `valgrind ./UninitData`
- Invalid write of size n
- Conditional jump depends on uninitialised value
- Add `--track-origins=yes`

# Memcheck - Run-time macros

Sometimes you want to check whether particular data is initialised, without artificially introducing a branch-statement based on it.

```
#include <valgrind/memcheck.h>
VALGRIND_CHECK_MEM_IS_ADDRESSIBLE(dataPtr, sizeof(double));
VALGRIND_CHECK_MEM_IS_DEFINED(dataPtr, sizeof(double));
VALGRIND_CHECK_VALUE_IS_DEFINED(myVariable);
```

will cause Memcheck to test for the data being initialised.
The first two have return values of the first address not addressible/initialised.

# Memcheck - Attaching a debugger

- Older versions of valgrind allowed you to attach a debugger in the same terminal to a valgrind running process.
- Now, the process is slightly more complex, but allows better diagnostics.
- Pass the options: `--vgdb=yes --vgdb-error=0`
- Now at start-up, valgrind displays:
- Now invoke `gdb` on the same executable, and within `gdb`:

```
target remote | /usr/lib/valgrind/../../bin/vgdb
```

# Memcheck - checking memory status

- From within **gdb** attached as above, you can check which variables are initialized.

```
(gdb) print data
      (float *) 0x5a20040
(gdb) monitor get_vbits 0x5a20040 24
ffffffff 00000000 00000000 00000000 00000000 00000000
```

  Showing that the first 4 bytes are undefined, and the next 20 are defined.

```
monitor get_vbits 0x5a20038 24
-------- -------- ffffffff 00000000 00000000 00000000
```

  Showing that the previous 8 bytes are not accessible (and may cause a seg-fault if accessed).

- Now introduce uninitialized data from stack and try again...

# Leak checking

- The option `--leak-check=yes` will generate a list of all memory not freed using `free` / `delete` at the end of the program.
- Use your judgement as to whether this is important or not; if a memory allocation is only done once (and leaks), it may not matter.
- There are different levels of memory loss:
  - Definitely lost: No pointers to the data exist at program exit.
  - Still reachable: A pointer to the leaked memory can be found via another pointer from non-lost memory
  - Indirectly lost: A pointer to the leaked memory can only be found from otherwise lost memory.
  - Possibly lost: A pointer offset from the originally allocated data start has been found, i.e. it may be possible to recover the original pointer and free the memory.
- Example: `valgrind --leak-check=full DataLeak`

# Suppressions

- All valgrind errors can be suppressed.
- This may be useful for ignoring errors from libraries over which you have no control.
- Use the `--gen-suppressions` option to write a suppressions file.

# SGCheck

- SGCheck aims to do what Memcheck does, only for stack-based arrays.
- It does this heuristically, assuming that instructions accessing a particular array should always access that array.
- Thus, a loop that attempts to access out-of-bounds of an array:

```
int a[10];
for(int i=0 ; i <= 10 ; i++) a[i] = i;
```

  will be caught.
- Some false positives may arise, though.
- See `valgrind --tool=exp-sgcheck ./SGDemo`
- However, I haven't used this tool much; most data in HPC applications is on the heap.

# Massif - overview

- Massif periodically takes a snapshot of live heap-allocated memory: its size and point of allocation.
- It can therefore pick up on memory leaks that are cleaned up at program exit (via destructor or similar).
- To run using Massif:

```
valgrind --tool=massif ./MassifDemo
```

- and post-process with:

```
ms_print massif.out.12345 > massif.out.12345.pp
```

# Massif snapshot frequency

- The frequency of snapshots is heuristically defined, but Massif will try to capture memory-use peaks.
- Typically, it will take between 50 and 100 snapshots over any program.
- Detailed snapshots with complete stack-traces of the provenance of all memory allocated are only taken occasionally. Use `--detailed-freq=1` for more of these.

# Callgrind

- Callgrind profiles code according to numbers of instructions used, and estimation of branch prediction and cache use
- `valgrind --tool=callgrind ./md_demo`
- Visualise using `kcachegrind`, at source-line and instruction level
- Caveats: The cache and branch predicition may not be particularly realistic.

# Helgrind

- Thread error detector.
- Detects potential deadlocks and data-races with pthreads
- Pthreads not advisable for HPC development; use MPI or OpenMP.

# DRD

- Data-Race Detector.
- Able to detect data-races and other conflicts within threaded programs.
- Some OpenMP errors may be detectable with this tool.

# DHat

- Dynamic heap analysis tool.
- Analyses how much data is used out of dynamically allocated arrays and how long blocks are allocated for.
- Useful for detecting:
  - Arrays which are sparsely used (reconsider data-layout)
  - Code lines which allocate short-lived blocks of memory (reuse existing heap memory, change algorithm, or consider a memory pool)
  - Code-lines which allocate blocks of memory that do not leak, but exist for a long time within the code (may well be intentional, but can suggest regions that are almost memory-leak-like).

# .valgrindrc

You can create a `/.valgrindrc` file containing, for example:

```
--memcheck:--leak-check=yes
--callgrind:cache-sim=yes
--callgrind:branch-sim=yes
--callgrind:simulate-wb=yes
--callgrind:simulate-hwpref=yes
--callgrind:cacheuse=yes
--callgrind:dump-instr=yes
```

to save typing at the command line.

# Valgrind documentation

See /usr/share/doc/valgrind/valgrind_manual.pdf.gz
Very detailed description of valgrind and well written.