Valgrind and Valgrind4hpc

November 1, 2024

Woo-Sun Yang
User Engagement Group, NERSC

Introduction

- Valgrind is a suite of tools for debugging, profiling and usage analysis
  - **Memcheck**: memory error and leak detector
  - **Cachegrind**: measures the number of instructions for profiling
  - **Callgrind**: similar to Cachegrind but records the call history among functions
  - **Massif**: a heap profiler
  - **DHAT**: a dynamic heap analysis tool
  - **Helgrind**, **DRD**: pthreads error detectors
  - And more…
- Works for C, C++ and Fortran
- Tools add their own instrumentation code at runtime
  - Make it run slower

Introduction

- Valgrind is a suite of tools for debugging, profiling and usage analysis
  - Today: **Memcheck**: memory error and leak detector
  - Future training?:
    - **Cachegrind**: measures the number of instructions for profiling
    - **Callgrind**: similar to Cachegrind but records the call history among functions
    - **Massif**: a heap profiler
    - **DHAT**: a dynamic heap analysis tool
    - **Helgrind**, **DRD**: pthreads error detectors
    - And more…
- Works for C, C++ and Fortran
- Tools add their own instrumentation code at runtime
  - Make it run slower

To Use Valgrind

- Build your code with `-g` flag
  - Memcheck: compile with no optimization (e.g., `-O0`)

- Command to use valgrind:

```
valgrind [--tool=memcheck] <other valgrind-options>
prog <prog-options]
```

Memcheck: a Memory Error Detector

- Types of memory errors Memcheck can help find:
  - Out-of-bound access of heap array
  - Accessing uninitialized memory
  - Incorrect freeing of heap memory (double-freeing heap memory, mismatched use of `malloc`/`new`/`new[]` vs `free`/`delete`/`delete[]`)
  - Overlapping `src` and `dst` pointers in `memcpy`
  - Misaligned memory allocation
  - Memory leaks
- Memcheck uses 2 extra blocks for bookkeeping state of a memory block
  - **V (valid-value) bits**: Values defined?
  - **A (valid-address) bits**: Accessible?
  - Each byte in memory has 8 V bits and a single A bit

Memcheck Invocation

```
valgrind [--tool=memcheck] <other valgrind-options> \
   prog <prog-options]
```

- Can omit `--tool=memcheck` since this is the default tool
- Some common options for Memcheck:
  - `--leak-check=<no|summary|yes|full>`
    - If set to `full` or `yes`, each individual leak will be shown in detail
  - `--track-origins=<yes|no>`
    - Check whether to track the origin of uninitialized value
    - If originating from a heap block, shows where the block was allocated
  - `--suppressions=<filename>`
    - File from which to read descriptions of errors to suppress

Memcheck - Finding uninitialized values

- An uninitialized-value use error is reported when your program uses a value which hasn't been defined
- In the example code, `x` is not initialized to a value, yet it is used for a comparison

```
$ cat -n manuel1.c
...
   5  int x;
   6
   7  if (x==0xCAFEBABE)
   8  {
   9    printf ("x = %d\n", 99);
  10  }
  11  else
  12  {
  13    printf ("x = %d\n", 88);
  14  }
...
```

```
$ gcc -g -O0 -o manuel1 manuel1.c

$ valgrind ./manuel1
...
==1018824== Conditional jump or move depends on
uninitialised value(s)
==1018824==    at 0x400525: main (manuel1.c:7)
```

Memcheck - Detecting illegal frees

- Incorrect freeing of heap memory
- Code attempting to free the memory block pointed to by `p` multiple times in the for-loop

```
$ cat -n doublefree.c
...
  7   int i;
  8   void* p = malloc(177);
  9   for (i = 0; i < 2; i++)
 10     free(p);
...

$ gcc -g -O0 -o doubletree
doubletree.c
```

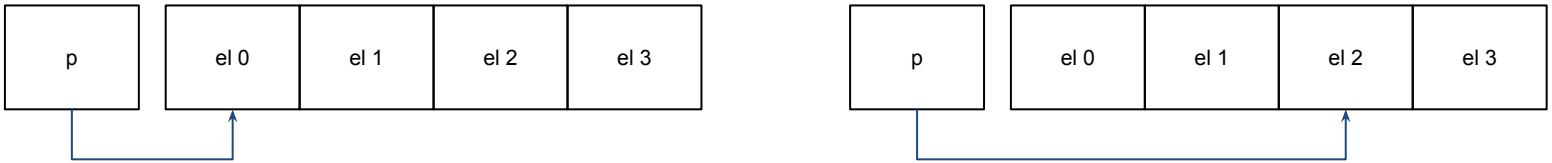
```
$ valgrind ./doublefree
...
==1064682== Invalid free() / delete / delete[] / realloc()
==1064682==    at 0x4E080EB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1064682==    by 0x400580: main (doublefree.c:10)
==1064682==  Address 0x523f040 is 0 bytes inside a block of
size 177 free'd
==1064682==    at 0x4E080EB: free (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1064682==    by 0x400580: main (doublefree.c:10)
==1064682==  Block was alloc'd at
==1064682==    at 0x4E056A4: malloc (in
/usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1064682==    by 0x400567: main (doublefree.c:8)
...
```

Memcheck - Detecting memory leaks

- Memory leak: When a program dynamically allocates a block of memory but doesn't free it after its use, the block cannot be reused, thus reducing available memory
- If a function with a memory leak is called repeatedly, available memory becomes less and less, reducing capability of the app

Memory leak types reported by Memcheck (1)

- Valgrind uses the terms “start pointer” and “interior pointer” when distinguishing different memory leak types
- A start pointer points to the start of the allocated block
- An interior pointer points to the middle of the block
  - Started as a start pointer, a pointer has been moved along
  - Sometimes an allocated block contains info about the actual data memory block (e.g., size), followed by the data block



Memory leak types reported by Memcheck (2)

- **Still reachable**
  - A start pointer to the block is found - a memory leak but the block still reachable
  - RRR -----> AAA or RRR -----> AAA -----> BBB
    - RRR: a well-defined pointer available at program exit
    - AAA & BBB: allocated memory blocks
- **Definitely lost**
  - No pointer to the block can be found
  - RRR --X--> AAA
- **Indirectly lost**
  - A block is lost because all the blocks that point to it are lost
  - RRR --X--> AAA -----> BBB
- **Possibly lost**
  - Blocks pointed to by all interior pointers directly or indirectly (as their correct state can be dependent on the context in the info part)
  - Can optionally activate heuristics by providing a context with `--leak-check-heuristics=(stdstring|length64|newarray|multipleinheritance|all)`
- https://developers.redhat.com/blog/2021/04/23/valgrind-memcheck-different-ways-to-lose-your-memory (for example codes)

Memcheck - Memory leak example (1)

- In the example code, the memory block pointed to by x is not freed
- Also, the code attempts to make out-of-bound memory access

```
$ cat memoryleak.c
...
  3  void f(void)
  4  {
  5     int* x = malloc(10 * sizeof(int));
  6     x[10] = 0;          // problem 1: heap block overrun
  7  }                      // problem 2: memory leak -- x not freed
  8
  9  int main(void)
 10  {
 11    f();
 12    return 0;
 13  }
$ gcc -g -O0 -o memoryleak memoryleak.c
```

Memcheck - Memory leak example (2)

```
$ valgrind --leak-check=full ./memoryleak
...
==1127011== Invalid write of size 4
==1127011==    at 0x400534: f (memoryleak.c:6)
==1127011==    by 0x400545: main (memoryleak.c:11)
==1127011==  Address 0x523f068 is 0 bytes after a block of size 40 alloc'd
==1127011==    at 0x4E056A4: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1127011==    by 0x400527: f (memoryleak.c:5)
==1127011==    by 0x400545: main (memoryleak.c:11)
==1127011==
==1127011==
==1127011== HEAP SUMMARY:
==1127011==     in use at exit: 40 bytes in 1 blocks
==1127011==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==1127011==
==1127011== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==1127011==    at 0x4E056A4: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1127011==    by 0x400527: f (memoryleak.c:5)
==1127011==    by 0x400545: main (memoryleak.c:11)
==1127011==
==1127011== LEAK SUMMARY:
==1127011==    definitely lost: 40 bytes in 1 blocks
==1127011==    indirectly lost: 0 bytes in 0 blocks
==1127011==      possibly lost: 0 bytes in 0 blocks
==1127011==    still reachable: 0 bytes in 0 blocks
==1127011==         suppressed: 0 bytes in 0 blocks
==1127011==
==1127011== For lists of detected and suppressed errors, rerun with: -s
==1127011== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Simple MPI code to run with Valgrind

```
$ cat -n memoryleak_mpi.c
...
   4 void f(void)
   5 {
   6    int* x = malloc(25000 * sizeof(int));
   7    x[25000] = 0;      // problem 1: heap block overrun
   8 }                     // problem 2: memory leak - x not freed
   9
  10 int main(int argc, char **argv)
  11 {
  12    int nproc, me;
  13    MPI_Init(&argc, &argv);
...
  16    f();
...
```

Running MPI codes with Valgrind

- Direct Valgrind output to a separate file for each MPI task, using `--log-file=...` flag
- Use `%p` for PID or `%q{some_env_var}`
  - `%q{SLURM_PROCID}` for MPI rank

```
$ srun -n 8 valgrind --leak-check=yes
--log-file=mc_%q{SLURM_JOB_ID}.%q{SLURM_PROCID}.out ./a.out

$ ls -l
-rw-------  1 elvis elvis  5481 Jun 23 08:56 mc_27100535.0.out
-rw-------  1 elvis elvis  5481 Jun 23 08:56 mc_27100535.1.out
-rw-------  1 elvis elvis  5481 Jun 23 08:56 mc_27100535.2.out
...
```

- Memcheck's MPI wrappers (for checking validity of MPI function args) not working with Cray MPI - tested with 3.23.0

Suppressing errors (1)

- Memcheck occasionally produces false positives or errors in the system library codes that you cannot change
- Can suppress them to focus on your own code
- Use **`--gen-suppressions=all`** to see what can be suppressed and create a suppression file for such errors
- Then use the suppression file with the **`--suppressions=<filename>`** flag
- Valgrind uses the default suppression file `$PREFIX/lib/valgrind/default.supp`

Suppressing errors (2)

```
$ srun -n 1 valgrind --leak-check=full
--gen-suppressions=all --log-file=errors
./memoryleak_mpi
```

```
$ grep -v -e '^==' errors > my.supp
$ vi my.supp        # Edit by hand
$ cat my.supp
```

```
{
   mysupp3
   Memcheck:Cond
   fun:add_entry
   fun:darshan_get_exe_and_mounts
   ...
   fun:main
}
```

Name

<tool>:<suppression type>

Calling context

"**fun**": function
"**...**": frame-level wildcard
(zero or more frames)

```
$ srun -n 8 valgrind --leak-check=full
--suppressions=my.supp
--log-file=ml.%q{SLURM_JOB_ID}.%q{SLURM_PROCID}.
out ./memoryleak_mpi
```

```
$ cat ml.32034480.0.out
...
==544049== LEAK SUMMARY:
...
==544049== ERROR SUMMARY: 18 errors from 6 contexts
(suppressed: 7 from 1)
```

Valgrind4hpc

- Valgrind4hpc is a HPE tool that aggregates duplicate Valgrind messages across MPI processes
  - Avoid duplication of messages and individual output files
  - Suppress known errors in HPE software
    - `$VALGRIND4HPC_BASEDIR/share/suppressions/{known,libmpich_cray,libpmi,misc}.supp`
- The tool works only for
  - Memcheck
  - Helgrind
  - DRD

Valgrind4hpc - How to run

- Use the commands in a batch session

```
$ module load valgrind4hpc
$ valgrind4hpc -n 8 --valgrind-args="--leak-check=yes"
./memoryleak_mpi
```

  - **`-n`**: the number of MPI tasks
  - Other `srun` flags (e.g., `-c 32`) are specified with the **`--launcher-args=...`** (or **`-l ...`**)
  - Valgrind arguments such as `--leak-check=yes` are passed with **`--valgrind-args=...`** (or **`-v ...`**)

Valgrind4hpc output

$ valgrind4hpc -n 8 --valgrind-args="
--leak-check=yes" ./memoryleak_mpi

```
RANKS: <0..7>

Invalid write of size 4
  at f (in memoryleak_mpi.c:7)
  by main (in memoryleak_mpi.c:16)
Address is 0 bytes after a block of size 40
alloc'd
  at malloc (in vg_replace_malloc.c:393)
  by f (in memoryleak_mpi.c:6)
  by main (in memoryleak_mpi.c:16)
```

```
RANKS: <0..7>

40 bytes in 1 blocks are definitely lost
  at malloc (in vg_replace_malloc.c:393)
  by f (in memoryleak_mpi.c:6)
  by main (in memoryleak_mpi.c:16)
```

```
RANKS: <0..7>

HEAP SUMMARY:
  in use at exit: 40 bytes in 1 blocks

LEAK SUMMARY:
   definitely lost: 40 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks

ERROR SUMMARY: 1 errors from 1 contexts
(suppressed 601)
```

Heap usage from execution trees (1)

- An execution tree (“xtree”) is made of a set of stack traces, each stack trace is associated with some resource consumptions or event counts
  - Typically to show a graphical or textual representation of the heap usage
- Get output in Callgrind or Massif format and use their tool (`callgrind_annotate` and `ms_print`) to turn it into an annotated heap usage profiling result

```
$ module rm darshan
$ srun -n 8 valgrind --xtree-memory=full
--xtree-memory-file=xtmemory.%q{SLURM_PROCID}.kcg
./memoryleak_mpi
$ callgrind_annotate --inclusive=yes
--sort=curB:100,curBk:100 xtmemory.0.kcg
```

`.kcg`: Callgrind
`.ms`: Massif

- Or ‘`valgrind4hpc -n 8 -v "--xtree-memory=full" -o xtmemory.kcg ./memoryleak_mpi`’ (separate files, `xtmemory.kcg.<pid>`)

Heap usage from execution trees (2)

```
...
-------------------------------------------------------------------------------
curB             curBk        totB               totBk        totFdB             totFdBk
-------------------------------------------------------------------------------

195,957 (100.0%) 602 (100.0%) 1,932,992 (100.0%) 789 (100.0%) 1,737,035 (100.0%) 187 (100.0%)  PROGRAM TOTALS


-------------------------------------------------------------------------------
curB             curBk        totB               totBk        totFdB             totFdBk       file:function
-------------------------------------------------------------------------------
195,957 (100.0%) 602 (100.0%) 1,911,132 (98.87%) 758 (96.07%) 1,719,355 (98.98%) 163 (87.17%)  memoryleak_mpi.c:main
100,000 (51.03%)   1 ( 0.17%)   100,000 ( 5.17%)   1 ( 0.13%)         0             0           memoryleak_mpi.c:f
 95,957 (48.97%) 601 (99.83%) 1,815,312 (93.91%) 762 (96.58%)    42,727 ( 2.46%)  95 (50.80%)  UnknownFile???:MPIR_Init_thread
 95,957 (48.97%) 601 (99.83%) 1,815,292 (93.91%) 761 (96.45%)    42,727 ( 2.46%)  95 (50.80%)  UnknownFile???:PMPI_Init
 95,109 (48.54%) 597 (99.17%)   123,965 ( 6.41%) 629 (79.72%)    28,856 ( 1.66%)  32 (17.11%)  UnknownFile???:MPIR_T_env_init

...
```

curB: current # of Bytes allocated
curBk: current # of Blocks allocated
totB: total allocated Bytes
totBk: total allocated Blocks
totFdB: total Freed Bytes
totFdBk: total Freed Blocks

Heap usage from execution trees (3)

```
--------------------------------------------------------------------------------
-- Auto-annotated source: memoryleak_mpi.c
--------------------------------------------------------------------------------
curB             curBk         totB                totBk         totFdB              totFdBk

...<snipped>...
      .            .                .             .                    .            .          void f(void)
      .            .                .             .                    .            .          {
100,000 (51.03%)   1 ( 0.17%)   100,000 ( 5.17%)   1 ( 0.13%)         0            0             int* x = malloc(25000 * sizeof(int));
      .            .                .             .                    .            .             x[25000] = 0;   // problem 1: heap block overrun
      .            .                .             .                    .            .          }          // problem 2: memory leak -- x not freed
      .            .                .             .                    .            .
      .            .                .             .                    .            .          int main(int argc, char **argv)
      .            .                .             .                    .            .          {
      .            .                .             .                    .            .             int nproc, me;
 95,957 (48.97%) 601 (99.83%) 1,811,132 (93.70%) 757 (95.94%)     42,727 ( 2.46%) 93 (49.73%)     MPI_Init(&argc, &argv);
      .            .                .             .                    .            .             MPI_Comm_size(MPI_COMM_WORLD, &nproc);
      .            .                .             .                    .            .             MPI_Comm_rank(MPI_COMM_WORLD, &me);
100,000 (51.03%)   1 ( 0.17%)   100,000 ( 5.17%)   1 ( 0.13%)         0            0             f();
      0            0                0             0          1,676,628 (96.52%) 70 (37.43%)     MPI_Finalize();
      .            .                .             .                    .            .             return 0;
      .            .                .             .                    .            .          }
```

Memory leaks from execution trees

- Similarly for memory leaks:

```
$ module rm darshan
$ srun -n 8 valgrind --xtree-leak=yes
--xtree-leak-file=xtleak.%q{SLURM_PROCID}.kcg
./memoryleak_mpi
$ callgrind_annotate --inclusive=yes
--sort=RB:100,PB:100,IB:100,DB:100 xtleak.0.kcg
```

- Or ‘`valgrind4hpc -n 8 -v ”--xtree-leak=yes” -o xtleak.kcg ./memoryleak_mpi`’

RB: Reachable Bytes
PB: Possibly lost Bytes
IB: Indirectly lost Bytes
DB: Definitely lost Bytes
…

Hands-on

- Multiple Valgrind versions on Perlmutter
  - `/usr/bin/valgrind`: v3.18.1
  - Valgrind modules: the latest - `valgrind/3.23.0`
  - Valgrind4hpc's `$VALGRIND4HPC_BASEDIR/bin/valgrind`: v3.20.0
- Can use any one of these for today's exercises
  - Training materials prepared with `/usr/bin/valgrind`
- We may retire the Valgrind modules in the future

Hands-on (cont'd)

- Exercise materials:

```
$ git clone https://github.com/NERSC/debugging
$ cd debugging/Valgrind/memcheck
```

- Follow the instructions in README.md for the following codes
  - `manuel1.c`: Valgrind exercise
  - `doublefree.c`: Valgrind exercise
  - `memoryleak.c`: Valgrind exercise
  - `memoryleak_mpi.c`: exercises on Valgrind, error suppressions, valgrind4hpc and heap usage/leak with xtree
- Optionally you can try the Fortran codes in the `fortran_memory` directory, too

Thank You!