Valgrind

Valgrind is a tool for memory debugging and profiling.

About Valgrind

Valgrind is an open-source tool, used mainly for debuggig memory-related problems, such as memory leaks, use of uninitalized memory etc. in C/C++ applications. The toolchain was however extended over time with more functionality, such as debugging of threaded applications, cache profiling, not limited only to C/C++.

Valgind is an extremely useful tool for debugging memory errors such as off-by-one. Valgrind uses a virtual machine and dynamic recompilation of binary code, because of that, you can expect that programs being debugged by Valgrind run 5-100 times slower.

The main tools available in Valgrind are :

- **Memcheck**, the original, must used and default tool. Verifies memory access in you program and can detect use of unitialized memory, out of bounds memory access, memory leaks, double free, etc.
- **Massif**, a heap profiler.
- **Hellgrind** and **DRD** can detect race conditions in multi-threaded applications.
- **Cachegrind**, a cache profiler.
- **Callgrind**, a callgraph analyzer.
- For a full list and detailed documentation, please refer to the official Valgrind documentation.

Installed versions

There are two versions of Valgrind available on the cluster.

- Version 3.8.1, installed by operating system vendor in /usr/bin/valgrind. This version is available by default, without the need to load any module. This version however does not provide additional MPI support. Also, it does not support AVX2 instructions, **debugging of an AVX2-enabled executable with this version will fail**
- Version 3.11.0 built by ICC with support for Intel MPI, available in module Valgrind/3.11.0-intel-2015b. After loading the module, this version replaces the default valgrind.
- Version 3.11.0 built by GCC with support for Open MPI, module Valgrind/3.11.0-foss-2015b

Usage

Compile the application which you want to debug as usual. It is advisable to add compilation flags -g (to add debugging information to the binary so that you will see original source code lines in the output) and -O0 (to disable compiler optimizations).

For example, lets look at this C code, which has two problems :

```
#include <stdlib.h>

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[10] = 0; // problem 1: heap block overrun
}             // problem 2: memory leak -- x not freed

int main(void)
{
   f();
   return 0;
}
```

Now, compile it with Intel compiler :

```
$ module add intel
$ icc -g valgrind-example.c -o valgrind-example
```

Now, lets run it with Valgrind. The syntax is :

valgrind [valgrind options] <your program binary> [your program options]

If no Valgrind options are specified, Valgrind defaults to running Memcheck tool. Please refer to the Valgrind documentation for a full description of command line options.

```
$ valgrind ./valgrind-example
==12652== Memcheck, a memory error detector
==12652== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==12652== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==12652== Command: ./valgrind-example
==12652==
==12652== Invalid write of size 4
==12652== at 0x40053E: f (valgrind-example.c:6)
==12652== by 0x40054E: main (valgrind-example.c:11)
==12652== Address 0x5861068 is 0 bytes after a block of size 40 alloc'd
==12652== at 0x4C27AAA: malloc (vg_replace_malloc.c:291)
==12652== by 0x400528: f (valgrind-example.c:5)
==12652== by 0x40054E: main (valgrind-example.c:11)
```

```
==12652==
==12652==
==12652== HEAP SUMMARY:
==12652== in use at exit: 40 bytes in 1 blocks
==12652== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==12652==
==12652== LEAK SUMMARY:
==12652== definitely lost: 40 bytes in 1 blocks
==12652== indirectly lost: 0 bytes in 0 blocks
==12652== possibly lost: 0 bytes in 0 blocks
==12652== still reachable: 0 bytes in 0 blocks
==12652== suppressed: 0 bytes in 0 blocks
==12652== Rerun with --leak-check=full to see details of leaked memory
==12652==
==12652== For counts of detected and suppressed errors, rerun with: -v
==12652== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 6 from 6)
```

In the output we can see that Valgrind has detected both errors - the off-by-one memory access at line 5 and a memory leak of 40 bytes. If we want a detailed analysis of the memory leak, we need to run Valgrind with –leak-check=full option :

```
$ valgrind --leak-check=full ./valgrind-example
==23856== Memcheck, a memory error detector
==23856== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
==23856== Using Valgrind-3.6.0 and LibVEX; rerun with -h for copyright info
==23856== Command: ./valgrind-example
==23856==
==23856== Invalid write of size 4
==23856== at 0x40067E: f (valgrind-example.c:6)
==23856== by 0x40068E: main (valgrind-example.c:11)
==23856== Address 0x66e7068 is 0 bytes after a block of size 40 alloc'd
==23856== at 0x4C26FDE: malloc (vg_replace_malloc.c:236)
==23856== by 0x400668: f (valgrind-example.c:5)
==23856== by 0x40068E: main (valgrind-example.c:11)
==23856==
==23856==
==23856== HEAP SUMMARY:
==23856== in use at exit: 40 bytes in 1 blocks
==23856== total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==23856==
==23856== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==23856== at 0x4C26FDE: malloc (vg_replace_malloc.c:236)
==23856== by 0x400668: f (valgrind-example.c:5)
==23856== by 0x40068E: main (valgrind-example.c:11)
==23856==
==23856== LEAK SUMMARY:
```

```
==23856== definitely lost: 40 bytes in 1 blocks
==23856== indirectly lost: 0 bytes in 0 blocks
==23856== possibly lost: 0 bytes in 0 blocks
==23856== still reachable: 0 bytes in 0 blocks
==23856== suppressed: 0 bytes in 0 blocks
==23856==
==23856== For counts of detected and suppressed errors, rerun with: -v
==23856== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 6 from 6)
```

Now we can see that the memory leak is due to the malloc() at line 6.

Usage with MPI

Although Valgrind is not primarily a parallel debugger, it can be used to debug parallel applications as well. When launching your parallel applications, prepend the valgrind command. For example :

```
$ mpirun -np 4 valgrind myapplication
```

The default version without MPI support will however report a large number of false errors in the MPI library, such as :

```
==30166== Conditional jump or move depends on uninitialised value(s)
==30166== at 0x4C287E8: strlen (mc_replace_strmem.c:282)
==30166== by 0x55443BD: I_MPI_Processor_model_number (init_interface.c:427)
==30166== by 0x55439E0: I_MPI_Processor_arch_code (init_interface.c:171)
==30166== by 0x558D5AE: MPID_nem_impi_init_shm_configuration (mpid_nem_impi_extensions.c:1091
==30166== by 0x5598F4C: MPID_nem_init_ckpt (mpid_nem_init.c:566)
==30166== by 0x5598B65: MPID_nem_init (mpid_nem_init.c:489)
==30166== by 0x539BD75: MPIDI_CH3_Init (ch3_init.c:64)
==30166== by 0x5578743: MPID_Init (mpid_init.c:193)
==30166== by 0x554650A: MPIR_Init_thread (initthread.c:539)
==30166== by 0x553369F: PMPI_Init (init.c:195)
==30166== by 0x4008BD: main (valgrind-example-mpi.c:18)
```

so it is better to use the MPI-enabled valgrind from module. The MPI versions requires library :

$EBROOTVALGRIND/lib/valgrind/libmpiwrap-amd64-linux.so

which must be included in the LD_PRELOAD environment variable.

Lets look at this MPI example :

```
#include <stdlib.h>
#include <mpi.h>

int main(int argc, char *argv[])
{
```

```
        int *data = malloc(sizeof(int)*99);

        MPI_Init(&argc, &argv);
        MPI_Bcast(data, 100, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Finalize();

        return 0;
}
```

There are two errors - use of uninitialized memory and invalid length of the buffer. Lets debug it with valgrind :

```
$ module add intel impi
$ mpiicc -g valgrind-example-mpi.c -o valgrind-example-mpi
$ module add Valgrind/3.11.0-intel-2015b
$ mpirun -np 2 -env LD_PRELOAD $EBROOTVALGRIND/lib/valgrind/libmpiwrap-amd64-linux.so valgrind
```

Prints this output : (note that there is output printed for every launched MPI process)

```
==31318== Memcheck, a memory error detector
==31318== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31318== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31318== Command: ./valgrind-example-mpi
==31318==
==31319== Memcheck, a memory error detector
==31319== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==31319== Using Valgrind-3.9.0 and LibVEX; rerun with -h for copyright info
==31319== Command: ./valgrind-example-mpi
==31319==
valgrind MPI wrappers 31319: Active for pid 31319
valgrind MPI wrappers 31319: Try MPIWRAP_DEBUG=help for possible options
valgrind MPI wrappers 31318: Active for pid 31318
valgrind MPI wrappers 31318: Try MPIWRAP_DEBUG=help for possible options
==31319== Unaddressable byte(s) found during client check request
==31319== at 0x4E35974: check_mem_is_addressable_untyped (libmpiwrap.c:960)
==31319== by 0x4E5D0FE: PMPI_Bcast (libmpiwrap.c:908)
==31319== by 0x400911: main (valgrind-example-mpi.c:20)
==31319== Address 0x69291cc is 0 bytes after a block of size 396 alloc'd
==31319== at 0x4C27AAA: malloc (vg_replace_malloc.c:291)
==31319== by 0x4007BC: main (valgrind-example-mpi.c:8)
==31319==
==31318== Uninitialised byte(s) found during client check request
==31318== at 0x4E3591D: check_mem_is_defined_untyped (libmpiwrap.c:952)
==31318== by 0x4E5D06D: PMPI_Bcast (libmpiwrap.c:908)
==31318== by 0x400911: main (valgrind-example-mpi.c:20)
==31318== Address 0x6929040 is 0 bytes inside a block of size 396 alloc'd
==31318== at 0x4C27AAA: malloc (vg_replace_malloc.c:291)
```

```
==31318== by 0x4007BC: main (valgrind-example-mpi.c:8)
==31318==
==31318== Unaddressable byte(s) found during client check request
==31318== at 0x4E3591D: check_mem_is_defined_untyped (libmpiwrap.c:952)
==31318== by 0x4E5D06D: PMPI_Bcast (libmpiwrap.c:908)
==31318== by 0x400911: main (valgrind-example-mpi.c:20)
==31318== Address 0x69291cc is 0 bytes after a block of size 396 alloc'd
==31318== at 0x4C27AAA: malloc (vg_replace_malloc.c:291)
==31318== by 0x4007BC: main (valgrind-example-mpi.c:8)
==31318==
==31318==
==31318== HEAP SUMMARY:
==31318== in use at exit: 3,172 bytes in 67 blocks
==31318== total heap usage: 191 allocs, 124 frees, 81,203 bytes allocated
==31318==
==31319==
==31319== HEAP SUMMARY:
==31319== in use at exit: 3,172 bytes in 67 blocks
==31319== total heap usage: 175 allocs, 108 frees, 48,435 bytes allocated
==31319==
==31318== LEAK SUMMARY:
==31318== definitely lost: 408 bytes in 3 blocks
==31318== indirectly lost: 256 bytes in 1 blocks
==31318== possibly lost: 0 bytes in 0 blocks
==31318== still reachable: 2,508 bytes in 63 blocks
==31318== suppressed: 0 bytes in 0 blocks
==31318== Rerun with --leak-check=full to see details of leaked memory
==31318==
==31318== For counts of detected and suppressed errors, rerun with: -v
==31318== Use --track-origins=yes to see where uninitialised values come from
==31318== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 4 from 4)
==31319== LEAK SUMMARY:
==31319== definitely lost: 408 bytes in 3 blocks
==31319== indirectly lost: 256 bytes in 1 blocks
==31319== possibly lost: 0 bytes in 0 blocks
==31319== still reachable: 2,508 bytes in 63 blocks
==31319== suppressed: 0 bytes in 0 blocks
==31319== Rerun with --leak-check=full to see details of leaked memory
==31319==
==31319== For counts of detected and suppressed errors, rerun with: -v
==31319== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 4 from 4)
```

We can see that Valgrind has reported use of unitialised memory on the master process (which reads the array to be broadcasted) and use of unaddresable memory on both processes.