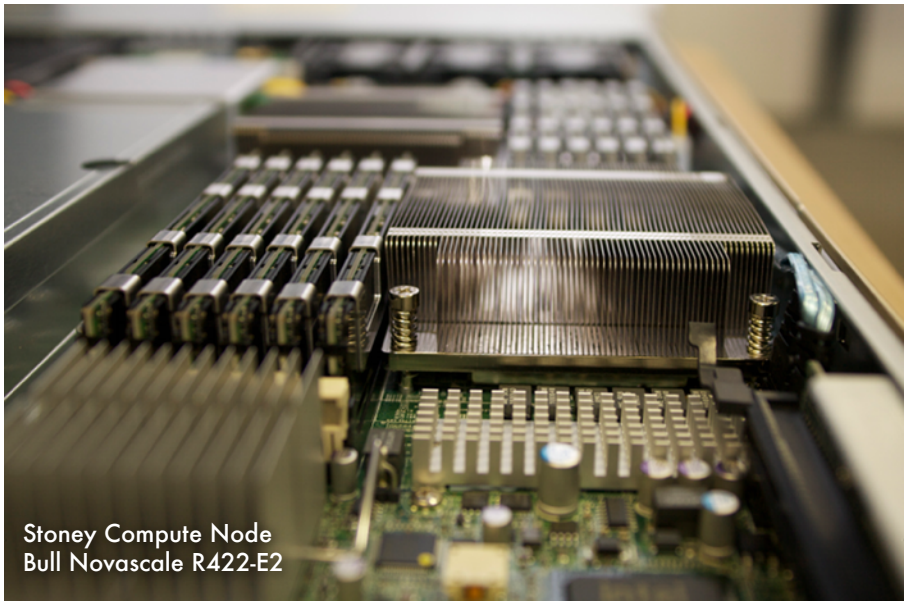
ICHEC

TECHNICAL REPORT

Mr. Ivan Girotto
ICHEC Computational Scientist



Stoney Compute Node
Bull Novascale R422-E2

Using Valgrind :: detecting memory errors

Valgrind is a suite of command line tools both for debugging and profiling codes on Linux system. Memcheck is Valgrind's heavyweight memory checking tool. All reads and writes of memory are checked, and calls to malloc/new/ free/delete are intercepted.

Introduction

Valgrind can detect common errors such as:

- Reading/writing freed memory or incorrect stack areas.
- Using values before they have been initialised
- Incorrect freeing of memory, such as double freeing heap blocks.
- Misuse of functions for memory allocations: new(), malloc(), free(), deallocate(), etc.
- Memory leaks - unintentional memory consumption often related to program logic flaws which lead to loss of memory pointers prior to deallocation.
- Overlapping source and destination pointers in memcpy() and related functions.

Valgrind also provides different profiling tools: Cachegrind, Callgrind and Massif. This tutorial discusses only the Memcheck tool. For full documentation make reference to the Valgrind User Manual.

Program Compilation

When testing for memory problems It is recommended to compile the code with both the debugging options -O0 (no optimization) and -g (debugging information). Using Valgrind with code that has been compiled with optimisation options could give incorrect results.

Using Valgrind


```
C:
$(CC) filecode.c -g -O0 -o fileprog.x

F90:
$(FC) filecode.f90 -g -O0 -o fileprog.x

C++:
$(CXX) filecode.cpp -g -O0 -fno-inline -o
fileprog.x
```

The -fno-inline flag avoids the inlining of functions into the main program and makes it easier to see the function-call chain. These examples can also be applied using the MPI compiler wrappers.

Using Valgrind

On both Stokes and Stoney ICHEC provides a module file for Valgrind which you should load first. When you are ready to begin, just run your application as you would normally, but place valgrind –tool=memcheck in front of your usual command-line invocation.

Load the Valgrind module and call nameprog.x using Valgrind:

```
$module load valgrind
$valgrind [--tool=memcheck] ./nameprog.x [prog-options]
```

Memcheck is the default, so if you want to use it you can omit the –tool flag. Valgrind executes the memcheck of the UNIX system call ls and prints to the standard output memory check information and some suggestions about how to get more information. This output can be redirect to a file with the option –log-file=filename.

Sample output for the ls command:

```
igirotto@stokes1:~> valgrind ls -ltr

==14159== Memcheck, a memory error detector.
==14159== Copyright (C) 2002-2008, and GNU GPL'd, by Julian Seward et al.
==14159== Using LibVEX rev 1878, a library for dynamic binary translation.
==14159== Copyright (C) 2004-2008, and GNU GPL'd, by OpenWorks LLP.
==14159== Using valgrind-3.4.0, a dynamic binary instrumentation framework.
==14159== Copyright (C) 2000-2008, and GNU GPL'd, by Julian Seward et al.
==14159== For more details, rerun with: -v
==14159==
total 120
-rw-r--r-- 1 igirotto ichec  230 2009-05-11 10:31 test.sh
drwxr-xr-x 3 igirotto ichec 4096 2009-05-11 12:25 GOTOLIB
-rw------- 1 igirotto ichec   27 2009-05-12 15:26 test.out
-rw------- 1 igirotto ichec   52 2009-05-12 15:26 test.err
-rw-r--r-- 1 igirotto ichec 6276 2009-05-12 15:26 compute_env.txt
drwxr-xr-x 4 igirotto ichec 4096 2009-05-12 16:20 WORKDIR
drwxr-xr-x 5 igirotto ichec 4096 2009-05-15 17:07 Benchmark
==14159==
==14159== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 5 from 2)
==14159== malloc/free: in use at exit: 17,803 bytes in 14 blocks.
==14159== malloc/free: 481 allocs, 467 frees, 55,184 bytes allocated.
==14159== For counts of detected errors, rerun with: -v
==14159== searching for pointers to 14 not-freed blocks.
==14159== checked 121,104 bytes.
==14159==
==14159== LEAK SUMMARY:
==14159==    definitely lost: 0 bytes in 0 blocks.
==14159==      possibly lost: 0 bytes in 0 blocks.
==14159==    still reachable: 17,803 bytes in 14 blocks.
==14159==         suppressed: 0 bytes in 0 blocks.
==14159== Rerun with --leak-check=full to see details of leaked memory.
```

Parallel Version

Valgrind can also be used for debugging parallel programs. Debugging POSIX pthreads is supported through the tool Heldgrind (see Valgrind User Manual). Debugging of distributed-memory applications which use the MPI message passing standard as is common in high performance computing environments is aslo possible. This support consists of a library of wrapper functions for the PMPI_* interface. When incorporated into the application's address space, either by direct linking or by LD_PRELOAD, the wrappers intercept calls to PMPI_Send(), PMPI_Recv(), etc. They then use client requests to inform Valgrind of memory state changes caused by the function being wrapped. This reduces the number of false positives that Memcheck would otherwise typically report for MPI applications.

The wrappers also take the opportunity to carefully check the size and defined-ness of buffers passed as arguments to MPI functions, hence detecting errors such as passing undefined data to PMPI_Send(), or receiving data into a buffer which is too small.

To use Valgrind in parallel like this requires us to use a pbs script so the execution can be orchestrated by the batch processing system.

%p is replaced with the current process ID. This is very useful for programs that invoke multiple processes.

You need to compile your application with the same compiler and mpi module that is used the script. Using a different MPI-library will generate a lot of false messages in your output file.

Sample pbs script:

```
#!/bin/bash

#PBS -l walltime=00:05:00
#PBS -l nodes=1:ppn=8
#PBS -A sci_test
#PBS -o test_valg.out
#PBS -e test_valg.err

module load intel-fc intel-cc
module load mvapich2-intel
module load valgrind/3.4.0

cd $HOME/Example

export LD_PRELOAD=/ichec/packages/valgrind/3.4.0/lib/valgrind/amd64-linux/
libmpiwrap.so

mpiexec valgrind -v --log-file=Valgrind.%p ./mpi_hello.x
```

Error Detecting and Error Messages

Thanks to the Memcheck feature Valgrind is excellent free software for code debugging. It is suitable for making quality control checks of code. It is able to detect two kinds of errors: access at bad memory address and find uninitialised values. We will now show how Valgrind error messages of can be interpreted and some common program errors that can be tracked easily.

It is easy to see the presence of the errors in the following code. There are two incorrect instructions for accessing at memory. The instructions make refer to 4 bytes beyond the space allocated for the buffer a. Valgrind checks them and print a message on the standard output. It is not so easy understood the Valgrind message. With a little bit of patience it is possible find some important errors present into the code. To make it easier to read the output you may wish to suppress some errors. This is possible but requires a little work, for more information see the Valgrind user manual. When Valgrind finds an error it tells you what kind of error it is and below the stack traces tells you where the problem occurred.

Find further information on the the Valgrind Project Homepage: http://www.valgrind.org

Example 1 - Can you spot the error?

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

#define DIM 1000

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

  float *a, k = 2.0, sup;
  int i;

  a = (float *) malloc( sizeof(float) * DIM );

  for( i = 0; i < DIM; ++i) a[i] = 0.0;

  sup = k;
  k = a[i];      // memory out-of-bound error in read operation
  a[i] = sup;    // memory out-of-bound error in write operation

  printf("GOOD END \n");

  free(a);

  return(EXIT_SUCCESS);
}
```

Resulting Valgrind Output:

```
igirotto@stokes1: ~/Example > gcc -O0 -g -Wall err_memory_access.c
igirotto@stokes1: ~/Example > valgrind ./a.out
…
==13072==
==13072== Invalid read of size 4
==13072==    at 0x400636: main (01_memory_access.c:18)
==13072==  Address 0x50cafd0 is 0 bytes after a block of size 4,000 alloc'd
==13072==    at 0x4A200CB: malloc (vg_replace_malloc.c:207)
==13072==    by 0x4005E8: main (01_memory_access.c:13)
==13072==
==13072== Invalid write of size 4
==13072==    at 0x400649: main (01_memory_access.c:19)
==13072==  Address 0x50cafd0 is 0 bytes after a block of size 4,000 alloc'd
==13072==    at 0x4A200CB: malloc (vg_replace_malloc.c:207)
==13072==    by 0x4005E8: main (01_memory_access.c:13)
GOOD END
==13072==
==13072== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 3 from 1)
==13072== malloc/free: in use at exit: 0 bytes in 0 blocks.
==13072== malloc/free: 1 allocs, 1 frees, 4,000 bytes allocated.
==13072== For counts of detected errors, rerun with: -v
==13072== All heap blocks were freed -- no leaks are possible.
```

Example 2 - And this time?

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

void f(void)
{
   int* x = malloc(10 * sizeof(int));
   x[9] = 2;
} // problem: memory leak -- x not freed

int main(int argc, char * argv[])
{
   f();
   printf("GOOD END \n");
   return 0;
}
```

Resulting Valgrind Output:

```
igirotto@stokes1:~/Example> gcc -O0 -g -Wall err_heap_memory_leak.c
igirotto@stokes1:~/Example> ./a.out
GOOD END
igirotto@stokes1:~/Example> valgrind --leak-check=full ./a.out
…
GOOD END
==6037==
==6037== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 3 from 1)
==6037== malloc/free: in use at exit: 40 bytes in 1 blocks.
==6037== malloc/free: 1 allocs, 0 frees, 40 bytes allocated.
==6037== For counts of detected errors, rerun with: -v
==6037== searching for pointers to 1 not-freed blocks.
==6037== checked 75,176 bytes.
==6037==
==6037== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6037==    at 0x4A200CB: malloc (vg_replace_malloc.c:207)
==6037==    by 0x400559: f (err_heap_memory_leak.c:6)
==6037==    by 0x400576: main (err_heap_memory_leak.c:12)
==6037==
==6037== LEAK SUMMARY:
==6037==    definitely lost: 40 bytes in 1 blocks.
==6037==      possibly lost: 0 bytes in 0 blocks.
==6037==    still reachable: 0 bytes in 0 blocks.
==6037==         suppressed: 0 bytes in 0 blocks.
```

Here you can see that Valgrind tracks the bytes of memory lost. It keeps trace of all the data allocated at runtime and at the end it checks which haven't been correctly freed. In this case the function f() has been called only once and it wasn't dangerous. Usually a memory leak is a tedious problem that simply grows the memory requirement unnecessarily. If a code has memory leak increasing the problem size at some point may well trigger a crash.

Example 3 - And this time?

```
 1    #include <stdlib.h>
 2    #include <stdio.h>
 3    #define DIM 1039596
 4
 5    int main(int argc, char *argv[])
 6    {
 7      float *a;
 8      int i, j;
 9      int mydim = DIM;
10
11      a = (float *)malloc(sizeof(float)*mydim);
12
13      printf("I'm here but how do I find the ERROR!\n");
14
15      j = mydim;
16      for(i=0; i<mydim; i++)
17        {
18          a[i] = i;
19          if(i > mydim-2)
20            {
21              a[j+1000] = a[i];
22              a[mydim-1] = a[i];
23            }
24        }
25
26      printf("a = %f \n", a[0]);
27      printf("GOOD END \n");
28
29      free(a);
30
31      return(EXIT_SUCCESS);
32    }
```

The output below comes from the execution of the code above. The difference between the two executions is that in the second one we are simulating what is commonly done when a code segfaults. It is natural to set some printf() in the code to see if a portion of the code runs (line 13). In this case this rudimental technique doesn't work. If you put a printf() after the malloc() the code works fine, or so it seems.

Curious Results:

```
igirotto@stokes1:~/Example> gcc 01_my_illegal.c -O0 -g -Wall -o illegal
igirotto@stokes1:~/Example> ./illegal
Segmentation fault (core dumped)
Add printf()
igirotto@stokes1:~/Example> emacs 01_my_illegal.c
igirotto@stokes1:~/Example> gcc 01_my_illegal.c -O0 -g -Wall -o illegal
igirotto@stokes1:~/Example> ./illegal
I'm here but how do I find the ERROR!
a = 0.000000
GOOD END
```

This doesn't means that the printf() solved the problem but only that it has hidden it and maybe our result is wrong! In this case checking the quality of the code with Valgrind could make a difference. As shown below Valgrind detects the incorrect write out of the bound of the array (line 21) and prints it to output.

Curious Results:

```
…
==13546== My PID = 13546, parent PID = 19441.  Prog and args are:
==13546==    ./illegal
==13546==
==13546== Invalid write of size 4
==13546==    at 0x400671: main (01_my_illegal.c:21)
==13546==  Address 0x515c380 is not stack'd, malloc'd or (recently) free'd
==13546==
 …
```

A segmentation fault (often shortened to segfault) is a particular error condition that can occur during the operation of computer software. A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed. This is managed by the operating system's memory management layer. In this case the call to the printf() function changed something in the memory mapping which has the side effect of masking the out of bounds error. Consequently results may vary from one compiler to another.

Using gdb it is easy to show what is happening. In the first case when we try to read the value of a[j+1000] we get an error message, because we attempted to access a memory location that it is not allowed. In the second case where the code has been compiled with line 13 uncomment the memory mapping is changed and we are able to access at the value of a[j +1000]. This behaviour is inline with what we have already seen.

gdb:

```
igirotto@stokes1:~/Example> icc -O0 -g -o illegal 01_my_illegal.c
igirotto@stokes1:~/Example> gdb illegal
...
(gdb) break 21
Breakpoint 1 at 0x400635: file 01_my_illegal.c, line 21.
(gdb) run
Starting program: /panfs/panasas/home-igirotto/Example/illegal

Breakpoint 1, main (argc=1, argv=0x7fffef16bac8) at 01_my_illegal.c:21
21                  a[j+1000] = a[i];
(gdb) print a[j+1000]
Cannot access memory at address 0x2b5fbc3f8360
(gdb) continue
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400652 in main (argc=1, argv=0x7fff45b544a8) at 01_my_illegal.c:21
21                          a[j+1000] = a[i];
```

gdb continued:

```
igirotto@stokes1:~/Example> icc -O0 -g -o illegal 01_my_illegal.c
igirotto@stokes1:~/Example> gdb illegal
...
(gdb) break 21
Breakpoint 1 at 0x400644: file 01_my_illegal.c, line 21.
(gdb) run
Starting program: /panfs/panasas/home-igirotto/Example/illegal
I'm here but how do I find for finding the ERROR!

Breakpoint 1, main (argc=1, argv=0x7fff6cc17568) at 01_my_illegal.c:21
21                  a[j+1000] = a[i];
(gdb) print a[j+1000]
$1 = 0
(gdb) continue
Continuing.
a = 0.000000
GOOD END
```

Useful Valgrind Options	
–help-debug	Print Valgrind help command plus debugging option.
–q	Show only the error message and ignore the others (–quite).
–version	Print Valgrind software version.
- -leak-check = <no|summary|full>	exec memory-leak analysis. A detected memory-leak means a block of allocated memory has not been freed and to which all references have been lost. So the block can now not be deallocated. This flags shows how many memory leak have been matched. The option full shows a lot of detail. When doing leak detection Valgrind trakcs all memory block allocations. When the program finishes it prints which blocks have not been freed.
- -show-reachble=<no|yes> [default: no]	Print some information about blocks of memory not deallocated but which have references.
–show-resolution=<low|med|high> [default: low]	If the option low is enabled each single message will print only the first time it will be matched. High prints the same message for each occurrence.

Stack Error

The following example shows a very common error in code. The example is in Fortran but the same thing could be happen in C using static allocation (e.g float array[9000]). Here on a stokes compute node you can see the stack limit is set to 2GB. This code that appears correct

```
igirotto@r2i1n15:~> ulimit -s
2097152
```

gives SIGSEV or Segmentation Fault.

As discussed the common approach is to put print statements into the code but it is not so useful. It seems that at runtime the program is blocked on the function call. Even the core file is not so useful because it could point to unrelated instructions which are trying to access another area of the stack. Valgrind typically gives some information about the stack problem. It suggests to increasing the stack frame size as it knows that the stack size is not sufficient.

If you have to use this flag, you may wish to consider rewriting your code to allocate on the heap rather than on the stack. As discussed further in Valgrind's description of –max-stackframe, a requirement for a large stack is a sign of potential portability problems. You are best advised to place all large data in heap-allocated memory.

Example 3 - And this time?

```
1     program test_oob
      2
      3       implicit none
      4
      5       real b(10)
      6       integer i
      7
      8       do i=1, 10
      9
     10          write(*,*) "Before set_val !"
     11          call set_val(b(i), 30000);
     12
     13       enddo
     14
     15       write(*,*) "--> ", b(1)
     16
     17     end program
     18
     19     subroutine set_val(in, dim)
     20
     21       implicit none
     22
     23       integer dim
     24       real in, out(dim*dim)
     25
     26       write(*,*) "Inside set_val !"
     27       in = out(1) + out(dim)
     28
     29     end subroutine
```

Crash & Core Dump:

```
igirotto@r2i1n15:~/Example> ./a.out
Before set_val !
Segmentation fault (core dumped)
```

And now with Valgrind:

```
igirotto@r2i1n15:~/Example> module load valgrind
igirotto@r2i1n15:~/Example> valgrind ./a.out
Before set_val !
==21654== Warning: client switching stacks?  SP change: 0x7fefff3c0 -->
0x7286c4fc0
==21654==          to suppress, use: --max-stackframe=3600000000 or greater
==21654== Invalid write of size 8
==21654==    at 0x402CE2: set_val_ (4-my-bigstack-write.f90:26)
==21654==    by 0x402B74: MAIN__ (4-my-bigstack-write.f90:11)
==21654==    by 0x402ACB: main (in /panfs/panasas/home-igirotto/WORKDIR/Codes/
Codes_example/testDebug/badcall/a.out)
==21654==  Address 0x7286c4fb8 is on thread 1's stack
==21654== Can't extend stack to 0x7286c4430 during signal delivery for thread 1:
==21654==   no stack segment
==21654==
==21654== Process terminating with default action of signal 11 (SIGSEGV):
dumping core
==21654==  Access not within mapped region at address 0x7286C4430
==21654==    at 0x402CE2: set_val_ (4-my-bigstack-write.f90:26)
==21654==  If you believe this happened as a result of a stack overflow in your
==21654==  program's main thread (unlikely but possible), you can try to
increase
==21654==  the size of the main thread stack using the --main-stacksize= flag.
==21654==  The main thread stack size used in this run was 16777216.
==21654==
==21654== Process terminating with default action of signal 11 (SIGSEGV)
==21654==  Access not within mapped region at address 0x7286C4F30
==21654==    at 0x491C62C: _vgnU_freeres (vg_preloaded.c:56)
==21654==  If you believe this happened as a result of a stack overflow in your
==21654==  program's main thread (unlikely but possible), you can try to
increase
==21654==  the size of the main thread stack using the --main-stacksize= flag.
==21654==  The main thread stack size used in this run was 16777216.
==21654==
==21654== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 3 from 1)
==21654== malloc/free: in use at exit: 5,383 bytes in 8 blocks.
==21654== malloc/free: 8 allocs, 0 frees, 5,383 bytes allocated.
==21654== For counts of detected errors, rerun with: -v
==21654== searching for pointers to 8 not-freed blocks.
==21654== checked 171,072 bytes.
…
Segmentation fault
```

Valgrind Core Dump:

```
igirotto@r2i1n15:~/WORKDIR/Codes/Codes_example/testDebug/badcall> ls -ltr
vgcore.21654
-rw------- 1 igirotto ichec 14299101 2009-05-26 16:05 vgcore.21654
```

If your program dies as a result of a fatal core-dumping signal, Valgrind will generate its own core file (vgcore.NNNNN) containing your program's state. You may use this core file for post-mortem debugging with gdb or similar.

In general, allocating large structures on the stack is a bad idea, because you can easily run out of stack space, especially on systems with limited memory or which expect to support large numbers of threads each with a small stack.

Warning: client switching stacks?: Valgrind spotted such a large change in the stack pointer that it guesses the client is switching to a different stack. At this point it makes a best effort guess where the base of the new stack is, and sets memory permissions accordingly.

You may get many bogus error messages following this, if Valgrind guesses incorrectly. At the moment "large change" is defined as a change of more that 2000000 in the value of the stack pointer register.