Using Valgrind

Introduction

This document will guide you through the process of using Valgrind on HPC Wales.

What is Valgrind?

Valgrind is a powerful debugging framework that is commonly used to identify a wide range of errors in serial and parallel code. This document will concentrate on introducing the memory error checking features of Valgrind; please be aware that it supports a range of other tools capable of analysing a variety of other problems including cache misses and threading errors.

Further details on accessing HPC Wales systems can be found in the User Guide and help can be obtained through the Support Desk:

E-mail	support@hpcwales.co.uk
Telephone	08452 572 207
Website	https://hpcwprod.service-now.com/

Table of Contents

Ewrop & Chymru: Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales: Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

Three Common Memory Management Errors

Valgrind’s `memcheck` tool can help to identify and diagnose a wide range of memory management errors. Here we concentrate on three of the most common types of error:

- memory errors, e.g. writing past the end of an array, freeing memory that has already been freed.
- memory leaks, i.e. failing to free dynamically allocated memory.
- uninitialised value errors, i.e. using a variable before it has been initialised.

Please note that this is not an exhaustive list – for the full range of checks available see the online documentation referenced at the end of this document.

Compiling Your Code For Valgrind

You need to recompile your code with `-g` to include the debugging information that Valgrind requires. It is also a good idea to use a low optimisation level when compiling (either `-O0` or `-O1`) to ensure that the errors reported by Valgrind are as accurate as possible. For example, to compile the serial C program `myexec` with the Intel C compiler:

```
[user.name@cf-log-001 ~]$ icc -g -O0 myexec.c -o myexec
```

Using Valgrind With Serial Code

Ensure that the `valgrind` module is loaded. Execute your program with:

```
[user.name@cf-log-001 ~]$ valgrind --leak-check=full ./myexec [args]
```

where `[args]` are the arguments you would usually pass to your executable. You will find that your program will execute much more slowly than usual; this is normal when using Valgrind. The `--leak-check=full` option tells Valgrind to print detailed information about every memory leak that it encounters. Other commonly-used options are:

- `--track-origins=yes` Track from where uninitialised value errors arise. Note that this imposes a significant overhead and will make execution of your code with Valgrind even slower.
- `--show-reachable=yes` Report if dynamically-allocated memory is still reachable when your program finishes (i.e. the program could have freed it before exiting). Also report indirectly-lost memory (see Valgrind documentation for what this means).

Output From Valgrind

Valgrind writes its output to standard output, so you'll find it wherever you normally find the output from your code. Note that Valgrind's output is sometimes rather opaque and there is not space here to cover all possible reports. For full details of Valgrind's messages, see the Explanation of error messages from Memcheck in the online Valgrind User Manual (http://valgrind.org/docs/manual/manual.html).

To illustrate typical Valgrind output we will use a simple example program (`test.c`) based on that found in the online Valgrind Quick Start guide(http://valgrind.org/docs/manual/quick-start.html):

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

void f(){

  int *x = malloc(sizeof(int)*10);
  x[10] = 0;

  printf("x[0]=%d\n",x[0]);

  return;
}

int main(){

  f();

  return 0;
}
```

We compile this with:

```
[user.name@cf-log-001 ~]$ icc -g -O0 test.c -o test
```

and execute Valgrind with:

```
[user.name@cf-log-001 ~]$ valgrind --leak-check=yes --track-origins=yes ./test
```

Memory Errors

Our example code contains an out-of-bounds access to the array x. The array contains ten elements but we have made a mistake when we try to assign a value to its final element. The portion of Valgrind's output that identifies this is:

```
==25632== Invalid write of size 4
==25632==    at 0x40056A: f (test.c:8)
==25632==    by 0x400592: main (test.c:17)
==25632==  Address 0x581d068 is 0 bytes after a block of size 40 alloc'd
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
```

The first column of the output is the process ID to which the message relates. This can be

Memory Errors

useful when debugging an MPI-based program (see below). "`Invalid write of size 4`" identifies that this error was caused by the program writing outside of the memory that it had allocated (in this case, it is trying to write 4 bytes of memory to an unallocated address).

The stack trace that follows the error type has two parts. Note that each component should be read from the bottom up. The first part of the trace identifies where in the code the invalid write occurs; in this case, it is on line 8 in the function `f` in `test.c`, and `f` itself is called on line 17 of the `main` function in `test.c`. The second part of the trace identifies where the array that is being incorrectly accessed is initialised in the code; here, `x` is initialised on line 7 of `test.c` by the call to `malloc`. This is particularly useful in identifying how large the array that you are trying to access actually is.

Note that Valgrind can only detect array out-of-bounds errors on dynamically-created arrays. If you were to declare `x` on the stack as `int x[10]` instead of using `malloc`, Valgrind would not report the subsequent out-of-bounds access (although your compiler might).

Memory Leaks

Our example code contains a memory leak because the array `x` is dynamically allocated in the function `f` but not subsequently freed. Valgrind identifies this:

```
==25632== HEAP SUMMARY:
==25632==     in use at exit: 40 bytes in 1 blocks
==25632==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==25632==
==25632== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
```

Valgrind tells us that the memory allocated on line 7 of `test.c` was not freed before the program exited.

At the end of the output, Valgrind summarises all the memory leaks that it found:

```
==25632== LEAK SUMMARY:
==25632==    definitely lost: 40 bytes in 1 blocks
==25632==    indirectly lost: 0 bytes in 0 blocks
==25632==      possibly lost: 0 bytes in 0 blocks
==25632==    still reachable: 0 bytes in 0 blocks
==25632==         suppressed: 0 bytes in 0 blocks
```

These are the various types of leak that Valgrind can identify. You should concentrate on fixing "definitely lost" and "possibly lost" memory leaks -- our code here contained an example of definitely lost memory. For full details of what the different categories of memory leak mean, see the online Valgrind documentation link.

Ewrop & Chymru: Buddsoddi yn eich dyfodol
Cronfa Datblygu Rhanbarthol Ewrop
Europe & Wales: Investing in your future
European Regional Development Fund
ERDF
Llywodraeth Cymru
Welsh Government

Uninitialised Values

In our code we have attempted to use an entry in `x` before a value has been assigned to it, and Valgrind reports that this is an uninitialised value error:

```
==25632== Use of uninitialised value of size 8
==25632==    at 0x53039BD: _itoa_word (in /lib64/libc-2.5.so)
==25632==    by 0x5306E5A: vfprintf (in /lib64/libc-2.5.so)
==25632==    by 0x530F3F9: printf (in /lib64/libc-2.5.so)
==25632==    by 0x400587: f (test.c:10)
==25632==    by 0x400592: main (test.c:17)
==25632==  Uninitialised value was created by a heap allocation
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
==25632==
==25632== Conditional jump or move depends on uninitialised value(s)
==25632==    at 0x53039C7: _itoa_word (in /lib64/libc-2.5.so)
==25632==    by 0x5306E5A: vfprintf (in /lib64/libc-2.5.so)
==25632==    by 0x530F3F9: printf (in /lib64/libc-2.5.so)
==25632==    by 0x400587: f (test.c:10)
==25632==    by 0x400592: main (test.c:17)
==25632==  Uninitialised value was created by a heap allocation
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
==25632==
==25632== Conditional jump or move depends on uninitialised value(s)
==25632==    at 0x5306ED4: vfprintf (in /lib64/libc-2.5.so)
==25632==    by 0x530F3F9: printf (in /lib64/libc-2.5.so)
==25632==    by 0x400587: f (test.c:10)
==25632==    by 0x400592: main (test.c:17)
==25632==  Uninitialised value was created by a heap allocation
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
==25632==

==25632== Conditional jump or move depends on uninitialised value(s)
==25632==    at 0x530763F: vfprintf (in /lib64/libc-2.5.so)
==25632==    by 0x530F3F9: printf (in /lib64/libc-2.5.so)
==25632==    by 0x400587: f (test.c:10)
==25632==    by 0x400592: main (test.c:17)
==25632==  Uninitialised value was created by a heap allocation
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
==25632==
==25632== Conditional jump or move depends on uninitialised value(s)
==25632==    at 0x5305B60: vfprintf (in /lib64/libc-2.5.so)
==25632==    by 0x530F3F9: printf (in /lib64/libc-2.5.so)
==25632==    by 0x400587: f (test.c:10)
==25632==    by 0x400592: main (test.c:17)
```

Uninitialised Values

```
==25632==  Uninitialised value was created by a heap allocation
==25632==    at 0x4C23E83: malloc (vg_replace_malloc.c:291)
==25632==    by 0x400559: f (test.c:7)
==25632==    by 0x400592: main (test.c:17)
```

Note that Valgrind has printed multiple errors even though they all relate to a problem with a single line of `test.c` (where we attempt to print `x[0]` before a value is assigned to it). This is because of the internal working of `printf`. The important thing to note is that the error occurs on line 10 of `test.c` (where we call `printf` with the uninitialised `x[0]`) and it stems from the memory that is allocated on line 7.

Valgrind and MPI

Valgrind includes a set of wrappers for MPI functions that can be used to reduce the number of false positives reported. With the `valgrind` module loaded, recompile your code as above. Make sure your batch submission script also loads the `valgrind` module, and then replace your line that calls `mpirun` with the following:

```
LD_PRELOAD=$VALGRIND_LD_PRELOAD mpirun [args] valgrind [vargs] ./myexec
```

where `[args]` are the arguments you normally pass to `mpirun`, `[vargs]` are the arguments for Valgrind (see above) and `myexec` is your executable.

To check that the wrappers have been used successfully, look for the lines:

```
valgrind MPI wrappers  9202: Active for pid 9202
valgrind MPI wrappers  9202: Try MPIWRAP_DEBUG=help for possible options
```

near the beginning of the execution output. These will be output by each process in your job. Bear in mind that the pid will be different when you run your code (in this case it is 9202).

You can provide the following options to the MPI wrappers by setting the MPIWRAP_DEBUG environment variable accordingly:

Value	Description
`verbose`	Show entries/exits of all wrappers. Also show extra debugging info such as the status of outstanding MPI_Requests resulting from uncompleted MPI_Irecvs.
`quiet`	Only print output if a programming error or catastrophic failure of the wrappers occurs.
`warn`	Print a warning for each unwrapped function used, up to a maximum of three warnings per function.
`strict`	Print an error message and abort the program if an unwrapped function is used.

Online Documentation

Both the full Valgrind manual and a handy quick start guide (on which this documentation is largely based) can be found online.