

CINECA - Consorzio Interuniversitario

Gruppo Supercalcolo - Settore Sistemi ad Alte Prestazioni

Valgrind

Scritto da:

Gian Franco Marras

Con la collaborazione di:

Cristiano Calonaci

Indice

1	Valgrind	2
1.1	Memcheck	2
1.2	Cachegrind	3
2	Come compilare il nostro programma	4
3	Come utilizzare Valgrind	5
3.1	Commandline flags	7
4	Messaggi di errore	9
4.1	Errori del tipo Illegal read / Illegal write	9
4.2	Uso di variabili inizializzate	10
4.3	Illegal frees	11
4.4	Quando un blocco di memoria è liberato con un'inappropriata funzione di deallocation	11
4.5	Passing system call parameters with inadequate read/write per- missions	11
4.6	Memory leak detection	12
5	Cachegrind	13
5.1	Come utilizzare Cachegrind	13

Capitolo 1

Valgrind

Valgrind è una suite di tools da linea di comando per il debugging e profiling di codici eseguibili su sistemi operativi basati su Linux.

Il codice del programma è implementato in modalità modulare ed è formato da diversi strumenti o *tools*, tra i quali quelli di maggior interesse sono:

- Memcheck;
- Cachegrind;

1.1 Memcheck

Memcheck è un “*memory checking tool*” che durante l’esecuzione del codice rileva molti problemi riguardanti la gestione della memoria, tra i quali l’incorretta allocazione o liberazione della stessa. I principali problemi che si possono rivelare sono i seguenti:

- scrittura o lettura in aree inappropriate dello stack;
- uso di valori non inizializzati;
- scrittura o lettura in una zona della memoria liberata in precedenza;
- modalità incorretta nella liberazione di memoria, o doppia liberazione degli stessi blocchi;
- memorie perse o mai liberate;
- utilizzo incorretto delle funzioni `malloc` o `new` e liberazione mediante `free` o `delete`;

1.2 Cachegrind

Cachegrind è un “*cache profiler*”. Esso simula il caricamento e il passaggio dei dati nei vari livelli di cache e della ram e prevedere dunque quante “*cache miss*” avvengono ad ogni livello di memoria per ogni singola istruzione o riga del codice.

Su processori x86 e AMD64, Cachegrind autorileva con il comando “**cpuid**” la configurazione dei vari livelli di cache.

L’output di Cachegrind può essere visualizzato con KCacheGrind, un’applicazione di KDE che rappresenta i risultati ottenuti in una modalità grafica più intuitiva.

Capitolo 2

Come compilare il nostro programma

Per una corretta analisi del codice attraverso l'utilizzo di Valgrind, è pienamente consigliato compilare il proprio programma con le opzioni `-g -O0`:

- **Compilazione di un programma in fortran 90**

Compilazione con un compilatore intel:

```
-bash$ ifort nomeprog.f90 -g -O0 -o nomeprog.exe
```

Compilazione con un compilatore GNU:

```
-bash$ gfortran nomeprog.f90 -g -O0 -o nomeprog.exe
```

- **Compilazione di un programma in C**

```
-bash$ gcc nomeprog.c -g -O0 -o nomeprog.exe
```

Se si lavora in C++, per migliorare la navigazione nel codice, è consigliabile utilizzare un ulteriore flag:

- **Compilazione di un programma in C++**

Compilazione con un compilatore intel:

```
-bash$ icc nomeprog.cpp -g -O0 -fno-inline -o nomeprog.exe
```

Il flag `-fno-inline` inibisce la scrittura inlining delle funzioni nel main.

Capitolo 3

Come utilizzare Valgrind

In questo capitolo spiegheremo brevemente le varie opzioni che si possono aggiungere al comando **Valgrind** per ottenere una migliore e maggiore funzionalità dello strumento stesso.

La prima opzione da aggiungere che andremo a vedere serve per identificare quale strumento si vuole utilizzare:

```
-bash$ valgrind --tool = NOMETOOL
```

dove **NOMETOOL** identifica il tool da utilizzare:

- **memcheck**
- **cachegrind**

Se non viene comunicato lo strumento da utilizzare, viene selezionato di default il primo.

Per provare Valgrind su un eseguibile potete semplicemente digitare:

```
-bash$ valgrind --tool = memcheck ./nomeprog.exe [prog-options]
```

oppure

```
-bash$ valgrind ./nomeprog.exe [prog-options]
```

dove naturalmente l'eseguibile è stato opportunamente compilato con le opzioni specificate in precedenza.

Se proviamo a lanciare il comando **valgrind ls -l**, Valgrind analizza la chiamata di sistema mentre viene eseguita, in tempo reale stampa a video alcuni messaggi riguardanti la gestione della memoria e inoltre mostra alcuni suggerimenti per rilanciare il comando e mostrare quindi una più dettagliata analisi.

```

-bash$ valgrind ls -l
==14333== Memcheck, a memory error detector.
==14333== Copyright (C) 2002-2007, and GNU GPL'd, by Julian Seward et al.
==14333== Using LibVEX rev 1732, a library for dynamic binary translation.
==14333== Copyright (C) 2004-2007, and GNU GPL'd, by OpenWorks LLP.
==14333== Using valgrind-3.2.3, a dynamic binary instrumentation framework.
==14333== Copyright (C) 2000-2007, and GNU GPL'd, by Julian Seward et al.
==14333== For more details, rerun with: -v
==14333==
total 288
drwxr-xr-x  5 cin8223a cineca 32768 Jul 24 17:53 Esercizi_svolti
-rw-----  1 cin8223a cineca  1260 Jul 26 15:42 filename.23856
-rw-r--r--  1 cin8223a cineca    62 Jul 25 15:51 if_totalview.txt
drwxr-xr-x  2 cin8223a cineca 32768 Jul 26 09:31 Valgrind
==14333==
==14333== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 10 from 5)
==14333== malloc/free: in use at exit: 19,890 bytes in 27 blocks.
==14333== malloc/free: 195 allocs, 168 frees, 86,053 bytes allocated.
==14333== For counts of detected errors, rerun with: -v
==14333== searching for pointers to 27 not-freed blocks.
==14333== checked 198,264 bytes.
==14333==
==14333== LEAK SUMMARY:
==14333==    definitely lost: 0 bytes in 0 blocks.
==14333==    possibly lost: 0 bytes in 0 blocks.
==14333==    still reachable: 19,890 bytes in 27 blocks.
==14333==    suppressed: 0 bytes in 0 blocks.
==14333== Rerun with --leak-check=full to see details of leaked memory.
-bash$

```

Questi messaggi possono essere direttamente stampati su files con l'opzione `--log-file=filename`; in questo caso verrà creato un file chiamato `filename.nPID`.

Se il file è già stato creato in precedenza, non verrà trascritto ma ne verrà creato uno nuovo chiamato `filename.nPID.1`. Se voglio un file senza il pid finale bisogna usare l'opzione `--log-file-exactly=filename`.

Si può utilizzare anche l'opzione `--log-file-qualifier=<VAR>` per modificare il file in base al valore della variabile d'ambiente `VAR`. Questa opzione è molto utile quando si deve analizzare un programma MPI. In questo caso, il numero PID inserito nel nome del file può essere sostituito con il numero del rank del processo.

Altre opzioni utili che si possono usare da linea di comando sono:

```
-bash$ valgrind -h
```

Mostra tutte le opzioni utilizzabili dal Valgrind (`--help`).

```
-bash$ valgrind --version
```

Visualizza la versione installata di Valgrind.

```
-bash$ valgrind -q
```

Visualizza solamente gli eventuali errori e ignora tutti gli altri messaggi(`--quiet`).

```
-bash$ valgrind -v
```

Mostra più informazioni su vari aspetti del programma analizzato (`--verbose`).

3.1 Commandline flags

In questa sezione verranno illustrati ulteriori opzioni da poter inserire con il comando `valgrind`.

Il comando

```
--leak-check = <no|summary|yes|full>
```

attiva una dettagliata analisi sull'eventuale “*memory leak*” di un programma. Un “*memory leak*” equivale ad un blocco di memoria allocata che non è stata liberata e che non ha un puntatore associato. Questi blocchi di memoria non possono essere liberati perchè non sono associati a nessun puntatore. Questo flag identifica quante volte queste perdite si verificano durante l'esecuzione di un codice. Se l'opzione è settata su `full` o `yes` esso fornisce maggiori particolari su ogni singola perdita.

```
--show-reachable = <yes|no> [default: no]
```

Quando questo comando è inabilitato, Valgrind mostra soltanto i blocchi di memoria allocati per cui non è possibile trovare un puntatore ad esso associato. Quando invece è abilitato indica alcune informazioni anche sui blocchi che possiedono un puntatore.

```
--leak-resolution=<low|med|high> [default: low]
```

Quando questa opzione è settata su `low` lo stesso messaggio di errore o di warning generato dal programma viene evidenziato solo una volta; Quando l'opzione è su `high` il messaggio viene indicato ogni qual volta il programma la invoca.

```
--num-callers=<number> [default: 12]
```


Valgrind di default mostra 12 livelli di chiamata di funzione per aiutare l'utente all'identificazione della locazione del programma. Questa opzione può cambiare il valore fino ad un massimo di 50 chiamate. Può essere utile su codici con molte chiamate innestate. Questa opzione non influisce sul numero di errori mostrati nel report. Un valore troppo alto può influire sia sulle prestazioni sia sul carico in memoria.

`--max-stackframe=<number> [default: 2000000]`

Può essere necessario utilizzare questa opzione se il codice ha una grande allocazione di array nello stack. Di solito, con il valore di default, Valgrind analizza il codice correttamente. Tuttavia se il codice alloca grandi strutture nello stack, Valgrind riporterà erroneamente un gran numero di accessi invalidi in scritture e lettura.

Capitolo 4

Messaggi di errore

L'opzione Memcheck di Valgrind è un ottimo strumento di debugging ma in genere può rilevare soltanto due generi di errori: l'uso degli indirizzi illegali e l'uso dei valori non definiti. Tuttavia è di grandissima utilità per identificare i problemi di allocazione di memoria che ci possono essere all'interno di un codice. In questo capitolo verranno presentati i principali errori che vengono identificati da Valgrind con lo strumento Memcheck, che cosa significano e come risolverli.

4.1 Errori del tipo Illegal read / Illegal write

Questo messaggio appare quando nel codice si fa riferimento ad una zona della memoria non appartenente alla variabile considerata. Consideriamo il seguente codice:

```
15 allocate (A(100,100,100))
16 allocate (B(100,100))
17 n=101
...
39 do i=1,100
40   do j=1,100
41     do k=1,N
42       B(i,j) = A(i,j,k)
43     enddo
44   enddo
45 enddo
```

Nel ciclo più interno viene richiesta la lettura della matrice nella posizione $A(i,j,N)$, cioè in una posizione della memoria non appartenente alla matrice stessa. Infatti come si può ben notare la terza dimensione in questo caso è inferiore alla variabile N .

```
==19518== Invalid read of size 4
```

```

==19518==    at 0x402D93: MAIN__ (val.f90:42)
==19518==    by 0x402629: main (in /Valgrind/a.out)
==19518== Address 0x55B5930 is 0 bytes after a block of
           size 4,000,000 alloc'd
==19518==    at 0x4A18D9E: malloc (vg_replace_malloc.c:149)
==19518==    by 0x408C52: for_allocate (in /Valgrind/a.out)
==19518==    by 0x408BB4: for_alloc_allocatable (in /Valgrind/a.out)
==19518==    by 0x4027F8: MAIN__ (val.f90:15)
==19518==    by 0x402629: main (in /Valgrind/a.out)

```

Ora consideriamo il seguente codice:

```

15 allocate (A(100,100,100))
16 allocate (B(100,100))
17 N=101
...
26 do i=1,100
27   do j=1,100
28     do k=1,N
29       A(i,j,k) = B(i,j)
30     enddo
31   enddo
32 enddo

```

In questo caso si cerca di scrivere in una zona della memoria non appartenente alla matrice A e il messaggio di errore stampato a video è del tipo:

```

==26069== Invalid write of size 4
==26069==    at 0x402B91: MAIN__ (val.f90:29)
==26069==    by 0x402629: main (in /Valgrind/a.out)
==26069== Address 0x55B5930 is 0 bytes after a block
           of size 4,000,000 alloc'd
==26069==    at 0x4A18D9E: malloc (vg_replace_malloc.c:149)
==26069==    by 0x408C56: for_allocate (in /Valgrind/a.out)
==26069==    by 0x408BB8: for_alloc_allocatable (in /Valgrind/a.out)
==26069==    by 0x4027F8: MAIN__ (val.f90:15)
==26069==    by 0x402629: main (in /Valgrind/a.out)

```

Nel secondo caso si cerca di scrivere nella posizione A(i,j,101) quando la matrice è stata allocata nella terza dimensione a 100.

4.2 Uso di variabili inizializzate

Consideriamo un semplice programma scritto in C dove viene stampata a video una variabile:

```

int main()
{
    int x;
    printf ("x = %d\n", x);
}

```

Come si può notare la variabile `x` non viene inizializzata. Questa variabile può avere qualsiasi valore e dunque se utilizzata può creare scenari differenti alle nostre aspettative. Per riconoscere questo tipo di errore Valgrind stampa a video il seguente messaggio.

```

==31854== Use of uninitialised value of size 8
==31854==    at 0x4DEC7BA: _itoa_word (in /lib64/tls/libc-2.3.4.so)
==31854==    by 0x4DEF978: vfprintf (in /lib64/tls/libc-2.3.4.so)
==31854==    by 0x4DF5F77: printf (in /lib64/tls/libc-2.3.4.so)
==31854==    by 0x4004E9: main (print.c:4)

```

4.3 Illegal frees

Invalid free()

Memcheck tiene traccia dei blocchi allocati dal programma con la chiamata *malloc* o *new*, e vede se vengono liberati con la rispettiva chiamata in modo regolare (*free* o *delete*).

4.4 Quando un blocco di memoria è liberato con un'inappropriata funzione di deallocation

Memcheck controlla che allocazioni del tipo *malloc* vengano liberate con la chiamata *free* e allocazioni *new* con la chiamata *delete*.

Mismatched free() / delete / delete []

4.5 Passing system call parameters with inadequate read/write permissions

Memcheck controlla tutti i parametri delle chiamate di sistema. Se una chiamata di sistema necessita leggere da un buffer proveniente dal programma, Valgrind controlla che l'intero buffer è rintracciabile e che abbia dati leggibili.

```

Syscall param write(buf) points to uninitialised byte(s)
Syscall param exit(error_code) contains uninitialised byte(s)

```

4.6 Memory leak detection

Memcheck tiene traccia di tutti i blocchi di memoria allocati. Quando il programma termina identifica quali blocchi non sono stati liberati e dunque quali dati sono stati persi.

Capitolo 5

Cachegrind

Per usare questo strumento bisogna specificare insieme al comando **Valgrind** l'opzione `--tool=cachegrind`. Cachegrind fa “cache simulations” e annota linea per linea il numero di cache miss. In particolare registra:

- L1 instruction cache reads and misses;
- L1 data cache reads and read misses, writes and write misses;
- L2 unified cache reads and read misses, writes and writes misses.

In un moderno processore, una “*cache miss*” in una cache L1 costa circa 10 cicli di tempo macchina, in L2 circa 200 cicli. Uno spropositato numero di cache miss durante l'esecuzione di un programma aumenta notevolmente i tempi di calcolo del codice. Una dettagliata conoscenza del numero di cache miss linea per linea può essere molto utile per identificare quale parti di codice possono essere migliorate. Con queste informazioni il programmatore può cambiare opportunamente la struttura del programma e diminuire perciò i tempi di calcolo.

5.1 Come utilizzare Cachegrind

Come per lo strumento Memcheck, il codice da analizzare deve essere compilato con le stesse opzioni descritte nel capitolo 2.

Successivamente deve essere lanciato Valgrind nel seguente modo:

```
-bash$ valgrind --tool=cachegrind ./nomeprog.exe
```

Cachegrind stamperà a video un sommario statistico delle varie cache. Inoltre stampa un file dal nome `cachegrind.out.nPID` (dove `nPID` è il numero pid del processo) dove trascrive linea per linea le varie proprietà di cache.

Questo passo dovrà essere fatto ogni qualvolta viene modificato il codice, proprio per monitorare il numero di cache miss in base all'implementazione fatta.

Il file `cachegrind.out.nPID` può essere letto con il comando `cg_annotate` nel seguente modo:

```
-bash$ cg_annotate --nPID cachegrind.out.nPID
```

dove nPID è sempre il numero pid del programma eseguito con Cachegrind.

Con questa chiamata viene stampato a video linea per linea del nostro programma tutte le varie fasi di lettura scrittura e cache miss del nostro programma. Per facilitare la lettura dell'output possiamo aggiungere sulla destra del video il nostro codice associato linea per linea con il seguente comando:

```
-bash$ cg_annotate --nPID cachegrind.out.nPID nomeprog.f90
```

Per una migliore visualizzazione del file cachegrind.out.nPID può essere visualizzato con un programma ad interfaccia grafica chiamato “Kcachegrind”:

```
-bash$ kcachegrind cachegrind.out.nPID
```