Metodología de la Programación Paralela

Curso 2023/24

Valgrind

Guía Rápida de Usuario

Facultad de Informática

UNIVERSIDAD DE MURCIA

Índice

1. Acerca de Valgrind

Valgrind es una herramienta muy útil para depurar problemas en el código fuente, incluyendo problemas de memoria y de rendimiento. Esta herramienta puede recibir varios parámetros en función de la utilidad que se le desee dar (`valgrind -help`).

2. Depuración de Errores de Memoria

Para depurar errores de memoria se recomienda utilizar las opciones `-tool=memcheck -leak-check=yes`, que permitirán detectar, entre otros, problemas de uso de memoria no incializada, lectura/escritura de memoria que ha sido liberada, lectura/escritura fuera de los límites de bloques de memoria reservada dinámicamente y fugas de memoria.

Si se desea que la salida de Valgrind se imprima en un fichero en lugar de por pantalla (por ejemplo, cuando sea muy larga o se quiera almacenar) se debe añadir la opción `-log-file=<fichero>`

Supongamos el siguiente programa `prueba.c`:

```
int main(int argc, char **argv)
{
    int *array = (int *) malloc(20*sizeof(int));

    array[0] = 1;
    array[1] = 1;

    for(int i=2; i<20; i++) {
        array[i] = array[i-1] + array[i-2];
    }
    array[20] = 20;

    /* No se libera la memoria antes de salir */
    return 0;
}
```

A la hora de compilar el programa es muy importante incluir el flag de depuración al compilar (`-g`) o no aparecerán los números de línea del código fuente donde están los errores.

```
$ gcc -g prueba.c -o prueba
$ valgrind --tool=memcheck --leak-check=yes ./prueba
```

Y obtenemos (por pantalla):

```
==8926== Memcheck, a memory error detector
==8926== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==8926== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==8926== Command: ./prueba
==8926==
==8926== Invalid write of size 4
==8926==    at 0x40057D: main (prueba.c:16)
==8926==  Address 0x51f1090 is 0 bytes after a block of size 80 alloc'd
==8926==    at 0x4C2B6CD: malloc (vgpreload_memcheck-amd64-linux.so)
==8926==    by 0x40050C: main (prueba.c:8)
==8926==
==8926==
==8926== HEAP SUMMARY:
==8926==     in use at exit: 80 bytes in 1 blocks
==8926==   total heap usage: 1 allocs, 0 frees, 80 bytes allocated
==8926==
==8926== 80 bytes in 1 blocks are definitely lost in loss record 1 of 1
==8926==    at 0x4C2B6CD: malloc (vgpreload_memcheck-amd64-linux.so)
==8926==    by 0x40050C: main (prueba.c:8)
==8926==
==8926== LEAK SUMMARY:
==8926==    definitely lost: 80 bytes in 1 blocks
==8926==    indirectly lost: 0 bytes in 0 blocks
==8926==      possibly lost: 0 bytes in 0 blocks
==8926==    still reachable: 0 bytes in 0 blocks
==8926==         suppressed: 0 bytes in 0 blocks
==8926==
==8926== For counts of detected and suppressed errors, rerun with: -v
==8926== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 2 from 2)
```

Como se puede observar, el programa avisa de dos problemas de gestión de memoria:

- En la línea 16 de `prueba.c` se ha hecho una escritura fuera del límite de memoria asignado al array. Concretamente en `array[20]`;
- Hay una reserva de memoria en la línea 8 de `prueba.c` que no se libera: `array = (int *) malloc(20*sizeof(int));`

3. Profiling

Valgrind también incluye una extensión para realizar profiling. El profiling permite estudiar el tiempo de ejecución de las diferentes partes de un programa con el objetivo de identificar cuellos de botella y así optimizar el rendimiento de la aplicación.

Por ejemplo, dado el programa `matrices.c` que realiza el producto de dos matrices, podríamos estar interesados en conocer qué partes del código tardan más tiempo. El programa tiene dos rutinas principales: generación de los datos de entrada (`generar`) y realización de la multiplicación (`multiplicar`).

```
/* Genera números aleatorios que no estén repetidos */
void generar(double *m, int t)
{
    int i;
    for(i=0; i<t; i++) {
        while(1) {
            double valor = (20.*rand()) / RAND_MAX-10.;
            for(int j=0; j<i; j++) {
                if(m[j] == valor) { continue; }
            }
            m[i] = valor;
            break;
        }
    }
}
```

```
void multiplicar(int n, double *a, double *b, double *c)
{
    int i, j, k;
    double s;

    /* Multiplicación */
    for(i=0; i<n; i++) {
        for(j=0; j<n; j++) {
            s = 0.0;
            for(k=0; k<n; k++)
                s += a[i*n+k] * b[k*n+j];
            c[i*n+j] = s;
        }
    }
}
```

Compilar con:

```
$ gcc -g -Wall matrices.c -o matrices -lm
```

Podemos realizar profiling del programa utilizando Valgrind/Callgrind con la siguiente orden:

```
$ valgrind --tool=callgrind ./matrices
```

La ejecución de valgrind genera un fichero con el formato:

```
callgrind.out.<numero>
```

donde en cada ejecución, <numero> tiene un valor mayor. Ese fichero (por ejemplo, `callgrind.out.5436`) se puede utilizar como entrada para KCachegrind para analizar la información de profiling:

```
$ kcachegrind callgrind.out.5436
```

Si se observa en la Figura 1 el porcentaje del tiempo de ejecución que se dedica a cada parte del programa, la líneas 71 y 72 utilizan el 49.50 %, respectivamente, del tiempo de la función main (total 99 % del tiempo). En cambio, la funcionalidad principal del código (la multiplicación de matrices) sólo abarca un 0.99 % del tiempo. Eso parece indicar un cuello de botella que habría que solucionar.

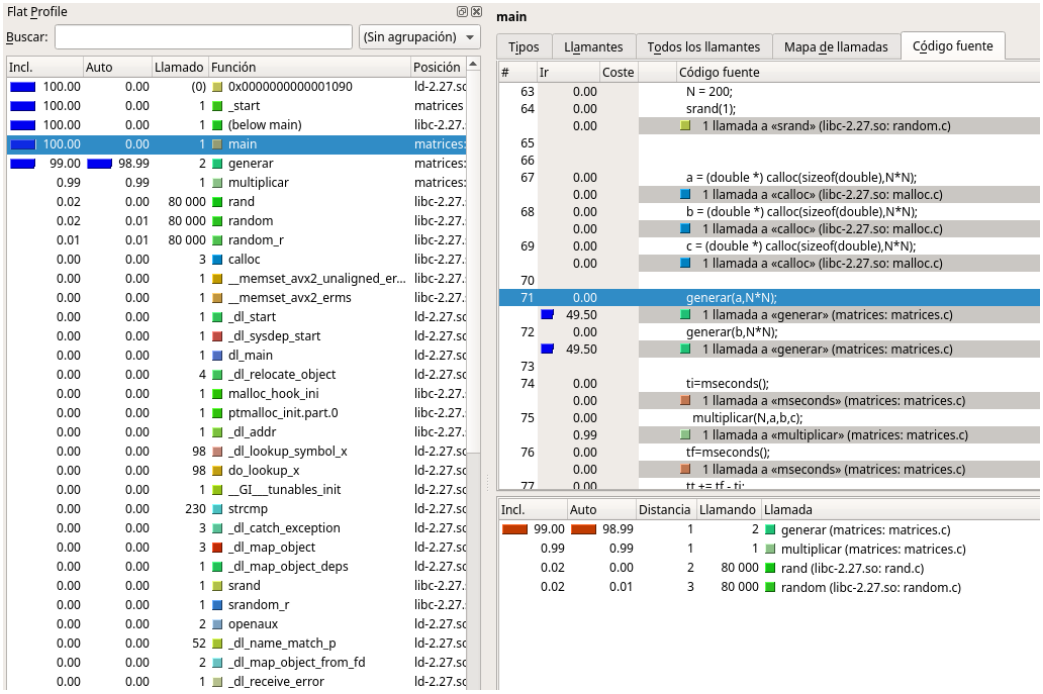

Figura 1: Información de profiling mostrada por KCachegrind.