Grado en INGENIERÍA INFORMÁTICA

ARQUITECTURA DE COMPUTADORES

102775

Departamento de Arquitectura de Computadores y Sistemas Operativos (DACSO)

PRÁCTICA nº 3

Paralelismo en el Acceso a Memoria y *Prefetching*

Acelerar el Rendimiento con programas *multi-thread* (Multi-core; Many-core y GPU)

Enunciado y Documentación

Medir, Evaluar y Optimizar Rendimiento en Proc. Multicore y GPUs

ENUNCIADO: Se analiza el efecto de las dependencias entre instrucciones, de las latencias de las operaciones de acceso a memoria, de la ejecución reordenada de instrucciones y del efecto de las instrucciones de *prefetching*.

Se introducirán los lenguajes OpenMP y Cilk+ para programar aplicaciones paralelas multi-thread (MIMD= Multiple-Instruction Multiple-Data) con variables compartidas.

Se mostrará el uso del lenguaje CUDA y de la librería de funciones Thrust para desarrollar aplicaciones que usan el paralelismo masivo para ser ejecutadas en GPUs.

En todos los casos, se medirá el rendimiento usando contadores H/W para entender mejor el comportamiento de la ejecución y se analizarán fragmentos críticos del código ensamblador.

OpenMP

Es una extensión de funciones (API) y sentencias especiales integradas en forma de *pragmas*, disponible para algunos lenguajes de programación (C, C++ y Fortran) que permite paralelizar una aplicación secuencial, repartiendo el cómputo entre varios threads (o hilos de ejecución) que colaboran compartiendo datos en variables comunes y sincronizándose de forma explícita. En este documento se mostrará el uso de las primitivas más básicas de OpenMP a través de ejemplos sencillos y se usará para ejecutar programas paralelos en procesadores multicore y multithread. OpenMP es un estándar soportado por muchos compiladores, entre ellos los compiladores gcc e icc. Para activar el uso de las directivas o comandos de OpenMP por parte del compilador se usa la opción –fopenmp en el compilador gcc, y de la opción –openmp en el compilador icc.

Cilk+

Es una extensión al lenguaje C++ para poder crear y sincronizar threads que se ejecuten en paralelo, y que colaboren compartiendo datos en variables comunes. El lenguaje está destinado a generar programas ejecutables en una CPU multicore. En este documento se explicará el uso muy básico de Cilk+, en concreto es uso de la primitiva **`cilk_for`**, mediante un ejemplo muy sencillo, que se ejecutará en una CPU multicore. El lenguaje Cilk+ fue creado en el MIT, como un proyecto de investigación y ha sido adoptado por Intel para sus plataformas multicore y sus aceleradores (Intel Phi). El compilador icc de Intel integra automáticamente el uso de primitivas Cilk+. La versión 4.9.0 de gcc (comando g++) soporta el lenguaje Cilk+ utilizando de forma explícita la opción de compilación –fcilkplus.

Procesador

Procesador Intel Core i7 950, repertorio x86-64

Clock Frequency: 3 – 3.333 GHz; *Out-of-Order Execution* (Dynamic Reordering)
4 Cores x 2-threads (8 H/W threads) + Capacidad SIMD de 128 bits
Private iL1 & dL1: 32KiB, Private L2: 256KiB, Shared L3: 8MiB (64-Byte cache line), 8 GiB DDR-RAM (25.6 GB/sec.)
CPUs: 0-7, Core0= {CPU0, CPU4}, Core1= {CPU1, CPU5}, Core2= {CPU2, CPU6}, Core3= {CPU3, CPU7}
El sistema operativo considera que hay 8 CPUs virtuales, aunque solamente haya 4 núcleos físicos de cómputo.

Una característica importante del procesador es que la frecuencia de funcionamiento varía según el número de núcleos de ejecución que se estén utilizando (Intel denomina a esta característica TurboBoost). Cuando hay uno o dos threads ejecutándose en un único núcleo de cómputo, el sistema funciona a la máxima frecuencia de reloj (3,333 GHz). En cambio, cuando funcionan todos los núcleos el sistema automáticamente reduce la frecuencia de reloj a 3 GHz, para evitar sobrepasar el límite máximo de potencia eléctrica (y de calor generado por el chip).

Sesión 1: TRABAJO PREVIO

Búsqueda binaria secuencial en un vector ordenado

A continuación se muestra el código de la función `BinarySearch`(), que implementa el algoritmo de búsqueda binaria de un número de tipo entero (`target`) en un vector ordenado (`index`[ ]) de N elementos. El algoritmo define un intervalo del vector (`L`, `R`), que inicialmente referencia a todo el vector, y en cada iteración del bucle se va dividiendo a la mitad, asegurando que el primer valor dentro del vector igual al valor buscado siempre está dentro de este intervalo. La función devuelve la posición del vector `index`[] con el primer valor igual o mayor al valor buscado.

```
unsigned int BinarySearch (const int *index, const int N, const int target) {
  unsigned long L=-1, R= N, M;
  int value;

  M = (R - L)>>1;  // Shift operation used to divide by 2. Assume N is not zero
  do {
       M = M + L;
       value = index[M];
       if (value < target)  L=M;  else  R=M;
       M = (R - L)>>1;
  } while (M);
  return R;  // apunta al primer valor del vector igual o mayor a 'target'
}
```

Pregunta 1a: Evaluar la complejidad computacional del algoritmo. Es decir, cuánto crece la cantidad total de operaciones que hace el programa al incrementar el tamaño del problema, N.

El siguiente código muestra el programa principal que utiliza la función anterior. La primera parte obtiene parámetros de la línea de comandos (usando **`argc`** y **`argv`**). Observar el código con detenimiento para poder identificar el significado de los tres parámetros que utiliza el programa. Se muestra un mensaje de ayuda si en tiempo de ejecución se usa la opción "H" o "h".

```
int main (int argc, char **argv) {
  int N=0, M=0, Rng= -1, i;
  char o = '1';
  int *A, *B, *C;

  if (argc>1) {  o = argv[1][0];    }
  if (argc>2) {  N = atoi(argv[2]); }
  if (argc>3) {  M = atoi(argv[3]); }

  if (o == 'H' || o == 'h')
    {  std::cout << "Arguments: opt N M" << std::endl;     exit(1);    }
```

Luego se reserva memoria de forma dinámica para los vectores A, B y C, de N, M y M elementos, respectivamente. Se utiliza una función **`malloc`** especial (disponible para el compilador icc) que reserva la memoria en direcciones cuyo valor es un múltiplo del valor que se le indica como segundo parámetro. Se verifica que el sistema haya logrado reservar memoria suficiente para los vectores (recordad que el tamaño de la memoria DRAM está limitado a 8 GBytes).

```
  // allocate free memory in addresses aligned to 64 Bytes
  A = (int *) _mm_malloc ( N*sizeof(int), 64);   B = (int *) _mm_malloc ( M*sizeof(int), 64);
  C = (int *) _mm_malloc ( M*sizeof(int), 64);

  if (A==NULL || B==NULL || C==NULL) // NULL addresses?
    { std::cout << "Malloc Error: no memory!\n" << std::endl;    return 0;  }
```

Posteriormente se inicializan los vectores A y B con números aleatorios y se ordena el contenido del vector A. La sintaxis **`std::`** indica el uso de clases de la librería estándar (**`cout`** representa la salida estándar y **`sort`** es una función de ordenación). Se muestran mensajes durante la ejecución para indicar el punto de la ejecución en el que está el programa.

```
  std::cout << "N (log)= " << N << " (" << bit_scan_reverse(N) << ")  M= " << M << std::endl;

  init_rand (A, N, 1, Rng);    init_rand (B, M, 2, Rng);

  std::cout << "Sorting " << N << " numbers" << std::endl;
  std::sort (A, A+N);
```

A continuación, según la opción seleccionada en el momento de la ejecución (**`o`**), se realiza el bucle que busca en el vector A[] los M valores contenidos en el vector B[], y que guarda el resultado de la búsqueda en el vector C[].

```
  switch (o) {  // different search implementations
    case '1':
          std::cout << "Binary Search" << std::endl;
          for (i = 0; i < M; i++)
             C[i] = BinarySearch (A, N, B[i]);
  }
```

Finalmente, si se escribe el carácter 'C' tras la opción de ejecución, el programa verifica que todos los resultados de la búsqueda sean correctos y contabiliza el número total de búsquedas de números aleatorios que han encontrado el valor en el vector A[]. Este número se puede utilizar para verificar que diferentes implementaciones del programa sean funcionalmente idénticas. Este código no es trivial de entender, y no es necesario hacerlo para alcanzar los objetivos de esta práctica.

```
  if (argv[1][1] == 'C')    // check results
  {
    long unsigned int n_matches=0;
    std::cout << "Checking Results ... ";
    for (i = 0; i < M; i++) {
      int pos = C[i], val = B[i];
      if (pos>0 && A[pos-1] >= val)
        std::cout <<A[pos-1]<<" in pos. "<< pos-1 <<" greater than "<< val <<"!!"<< std::endl;
      while (pos<N && A[pos] == val)
        { n_matches++; pos++; }
      if (pos<N && A[pos] < val)
        std::cout << A[pos] << " in pos. "<< pos <<" is lower than "<< val <<"!!"<< std::endl;
    }
    std::cout << " number of Matches = " << n_matches << std::endl;
  }
```

No se debe olvidar liberar la memoria reservada de forma dinámica (con funciones **free** especiales).

```
  _mm_free (A);  _mm_free (B);  _mm_free (C);  return 0;
}
```

El código generado por el compilador icc 16.0 con la opción –O3 se muestra a continuación, y al lado del código ensamblador se muestra el código equivalente. El compilador hace dos optimizaciones: (1) desenrollar el bucle del programa dos veces, y (2) substituir las instrucciones de salto que normalmente se utilizarían para codificar la instrucción if-then-else, por instrucciones de movimiento condicional (`cmovle` y `cmovg`). Se puede observar que el código ensamblador de cada una de las dos iteraciones desenrolladas consta de 9 instrucciones escalares: el compilador no ha vectorizado, es decir, no ha usado operaciones SIMD. Las 9 instrucciones se descomponen en 10 µoperaciones: un BRN, 8 INT y un LOAD.

```
BinarySearch: …
```

```
 0,72 | cf5:  add    %r8,%r9
 0,15 |       cmp    (%rdi,%r9,4),%r11d
45,71 |       cmovle %r9,%r10
 1,30 |       cmovg  %r9,%r8
 0,43 |       mov    %r10,%r9
 0,36 |       sub    %r8,%r9
 0,63 |       sar    %r9
 0,61 |       test   %r9,%r9
      |       je     d2f
 0,54 |       add    %r8,%r9
 0,18 |       cmp    (%rdi,%r9,4),%r11d
43,68 |       cmovle %r9,%r10
 1,19 |       cmovg  %r9,%r8
 0,34 |       mov    %r10,%r9
 0,37 |       sub    %r8,%r9
 0,60 |       sar    %r9
 0,57 |       test   %r9,%r9
      |       jne    cf5
```

```
REPEAT:
1.  M = M + L                 // INT
2.  c1= index[M] < target     // LOAD + INT
3.  L =  c1? M: L             // INT
4.  R = !c1? M: R             // INT
5.  M = R                     // INT
6.  M = M – L                 // INT
7.  M = M >> 1                // INT
8.  c2 = M == M               // INT
9.  if (c2) goto END          // BRN
10. M = M + L                 // INT
11. c1 = index[M] < target    // LOAD + INT
12. L =  c1? M: L             // INT
13. R = !c1? M: R             // INT
14. M = R                     // INT
15. M = M – L                 // INT
16. M = M >> 1                // INT
17. c2 = M == M               // INT
18. if (!c2) goto REPEAT      // BRN
END:
```

La primera optimización (desenrollar) no ofrece ningún beneficio en el rendimiento, pero la tarea de optimizar es muy compleja y los compiladores a veces utilizan estrategias generales que no siempre funcionan (al menos tampoco empeora el rendimiento). La segunda opción (usar instrucciones de movimiento condicional) sí que mejora el rendimiento.

Pregunta 1b: Describir con precisión lo que hacen las instrucciones 3 y 4. Cuando el compilador decide usar las instrucciones de movimiento condicional de datos en lugar de usar instrucciones de salto condicional, ¿qué dos beneficios se logran de cara al rendimiento de la ejecución?

Pregunta 1c: El compilador no ha podido vectorizar el bucle interno. Identificar e indicar las razones que impiden vectorizar.

Se toman medidas de la ejecución para diferentes tamaños del vector ordenado (N), siempre haciendo búsquedas de M= 100 millones de valores. Se mide el tiempo de ejecución del programa en dos casos: (a) "Solo Inicialización y Ordenación", que se salta la parte de realizar búsquedas y la verificación de resultados, pero que incluye el tiempo de generar N+M números aleatorios y de ordenar N números; y (b) "Ejecución completa con Búsqueda", realizando todo el proceso y las M búsquedas, sin la parte de verificación. Restando estos dos valores se puede aislar el rendimiento debido únicamente a las búsquedas.

Search M=100 Millones	Solo Inicialización y Ordenación		Ejecución completa con Búsqueda	
	Instructions	Time (seconds)	Instructions	Time (seconds)
N= 100.000	8,7 G	0,46	24,2 G	8,46
N= 1.000.000	9,3 G	0,56	28,2 G	16,63
N= 10.000.000	10,9 G	1,37	33,2 G	38,62
N= 100.000.000	32,5 G	10,8	57,5 G	72,7

Pregunta 1d: Calcular el valor de ICount únicamente para la fase de búsqueda: ¿cuánto crece en función de N y por qué?

Pregunta 1e: A partir de la frecuencia de reloj del procesador (3,333 GHz) calcular el valor de IPC (*Instructions Per Cycle*) de la fase de búsqueda y explicar su comportamiento a medida que crece N.

Search M=100 Millones	Solamente la fase de Búsqueda			
	ICount	Time (seconds)	CCount	IPC
N= ...				

Optimización: uso de instrucciones S/W de *prefetch*

Se sospecha que el problema de rendimiento de la búsqueda binaria puede ser debido a la **latencia** de los accesos a memoria, especialmente cuando el vector en el que se buscan los valores es muy grande. Para mejorar el rendimiento se aplica una estrategia de pre-búsqueda (*prefetching*) de datos, que consiste en pedir al sistema de memoria los valores de los dos caminos posibles en el árbol binario de búsqueda, antes de verificar cuál de los dos es el que hay que seguir. Tal como se ve en el código siguiente, la estrategia de *prefetch* no se utiliza en todas las iteraciones del bucle de búsqueda, sino solamente en las iteraciones internas (ni en las primeras, que explotan la localidad temporal de los accesos a memoria, ni en las últimas, que explotan la localidad espacial de los accesos).

```
#include "xmmintrin.h"

unsigned int BinSearchPref (const int *index, const int N, const int target) {
  unsigned long L=-1, R= N, M, Ml, Mr;         int value;
  M = (R - L)>>1;
  while (M>= N/1024)
  { // 1st phase: no prefetch (hot positions in index: temporal locality exploited in cache)
        M = M + L;
        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  while (M>=4)
  { // 2nd phase: prefetch (cold positions in index: very few memory access locality)
        M = M + L;

        // prefetch both possible destinations (Left and Right)
        Ml = (L+M)>>1;   Mr = (R+M)>>1;
        _mm_prefetch( (char const *)(&index[Ml]),_MM_HINT_NTA);
        _mm_prefetch( (char const *)(&index[Mr]),_MM_HINT_NTA);

        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  while (M)
  { // 3rd phase: no prefetch (spatial locality: vector now fits into single cache block)
        M = M + L;
        value = index[M];
        L= (value < target) ? M : L;    R= (value >= target) ? M : R;
        M = (R - L)>>1;
  }
  return R;
}
```

El código generado por el compilador icc 16.0 con la opción –O3 se muestra a continuación, y al lado del código ensamblador se muestra el código equivalente. El código corresponde con la segunda fase de la búsqueda, e incluye instrucciones de *prefetch*. En este caso el compilador decide no desenrollar el código. Además, el uso de instrucciones condicionales en lenguaje C simplifica el análisis del compilador para que vuelva a usar instrucciones de movimiento condicional (`cmovle` y `cmovg`). Cada iteración de la fase 2 de la búsqueda ejecuta ahora 15 instrucciones escalares, que se descomponen en 16 μoperaciones: un BRN, 12 INT, un LOAD y dos PREFETCH (que usan el mismo recurso MEM que las μoperaciones LOAD). Observar que las instrucciones LEA tienen la apariencia de ser operaciones de acceso a memoria, pero en cambio lo que hacen es calcular una dirección (*Load Effective Address*) pero sin acceder a memoria.

```
BinarySearchPref: …
```

```
 1,30 |43c:  add    %rcx,%rax
 0,35 |      lea    (%rcx,%rax,1),%r13
 1,23 |      shr    %r13
 1,22 |      prefet (%r14,%r13,4)
 5,42 |      lea    (%r10,%rax,1),%r13
 0,01 |      shr    %r13
      |      prefet (%r14,%r13,4)
 1,68 |      cmp    (%r14,%rax,4),%r9d
60,08 |      cmovle %rax,%r10
 2,46 |      cmovg  %rax,%rcx
 1,02 |      mov    %r10,%rax
 0,71 |      sub    %rcx,%rax
 1,36 |      shr    %rax
 1,32 |      cmp    $0x4,%rax
      |      jae    43c
```

```
REPEAT:
1.  M  = M + L                  // INT
2.  Ml = M + L                  // INT
3.  Ml = Ml >> 1                // INT
4.  PREFETCH ( index[Ml] )      // PREFETCH
5.  Mr = M + R                  // INT
6.  Mr = Mr >> 1                // INT
7.  PREFETCH ( index[Mr] )      // PREFETCH
8.  c1= index[M] < target       // LOAD + INT
9.  L =  c1? M: L               // INT
10. R = !c1? M: R               // INT
11. M = R                       // INT
12. M = M – L                   // INT
13. M = M >> 1                  // INT
14. c2 = M >= 4                 // INT
15. if (c2) goto REPEAT         // BRN
```

Se vuelven a tomar medidas de la ejecución (completa) para diferentes tamaños del vector ordenado (N), y siempre buscando M= 100 millones de valores. Se puede aislar el tiempo de la búsqueda a partir de los datos de la tabla anterior.

Search M=100 Millones	Prefetch: Ejecución completa con Búsqueda	
	Instructions	Time (seconds)
N= 100.000	21,0 G	7,99
N= 1.000.000	26,3 G	11,82
N= 10.000.000	33,3 G	25,54
N= 100.000.000	60,0 G	49,54

Pregunta 1f: Evaluar y valorar el *speedup* de la nueva versión respecto a la original (solamente las búsquedas).

Sesión 1: TRABAJO DURANTE la SESIÓN

La considerable mejora de rendimiento obtenida al usar la técnica de pre-búsqueda (*prefetching*) de datos indica que la latencia de los accesos a memoria es el principal cuello de botella del rendimiento. A continuación se quiere probar una estrategia alternativa que pretende obtener resultados similares a la técnica del *prefetching*. Consiste en combinar dos búsquedas en el mismo bucle, de forma que los accesos a memoria correspondientes a las dos búsquedas se puedan realizar a la vez o de forma solapada. El objetivo, expresado de forma más técnica, es aumentar el paralelismo a memoria de la ejecución.

```
typedef struct S { unsigned int P1; unsigned int P2; } tuple2;  // define a tuple2 data type

tuple2 BinarySearch2 (const int *index, const int N, const int target1, const int target2)
{
  unsigned long L1, R1, M1, L2, R2, M2;
  int value1, value2;

  L1=-1; R1= N;       L2=-1; R2= N;
  M1 = M2 = (R1 – L1)>>1;
  while (M1 && M2)
  {  // combine two searches
    M1 = M1 + L1;          M2 = M2 + L2;
    value1 = index[M1];    value2 = index[M2];
    L2= (value2 < target2) ? M2 : L2;     R2= (value2 >= target2) ? M2 : R2;
    L1= (value1 < target1) ? M1 : L1;     R1= (value1 >= target1) ? M1 : R1;
    M1 = (R1 – L1)>>1;     M2 = (R2 - L2)>>1;
  }
```

```
  if (M1) { // one search may take one more step than the other search
    M1 = M1 + L1;    value1 = index[M1];
    L1= (value1 < target1) ? M1 : L1;    R1= (value1 >= target1) ? M1 : R1;
    M1 = (R1 – L1)>>1;
  }
  else if (M2) {
    M2 = M2 + L2;    value2 = index[M2];
    L2= (value2 < target2) ? M2 : L2;    R2= (value2 >= target2) ? M2 : R2;
    M2 = (R2 - L2)>>1;
  }
  tuple2 T;   T.P1= R1; T.P2= R2;   return T;
}
```

El código anterior muestra cómo utilizar una estructura de datos compuesta de dos valores enteros (**tuple2**) para que una función pueda devolver directamente dos valores. Realizar dos búsquedas de forma simultánea solamente requiere duplicar las variables locales necesarias para cada búsqueda y cuidar el caso especial en que dos búsquedas no finalicen exactamente en la misma iteración del bucle. En este último caso, como máximo una de las búsquedas puede necesitar hacer una iteración más que la otra búsqueda. Para simplificar el código, los casos especiales se tratan fuera del bucle principal.

El código siguiente es el que realiza la llamada a la función anterior para realizar las M búsquedas de 2 en 2. Observar que la optimización es equivalente a desenrollar el bucle externo aplicado sobre la lista de valores a buscar, mientras que el bucle de búsqueda no se desenrolla (porque tiene dependencias recurrentes de datos).

```
case '2':  std::cout <<  "Binary Search - fuse 2 searches " << std::endl;
           for (i = 0; i < M; i+=2) {
             tuple2 T;
             T = BinarySearch2 (A, N, B[i], B[i+1]);
             C[i] = T.P1;  C[i+1] = T.P2;
           }
           break;
```

Realizar medidas de tiempo de la ejecución de la versión anterior para M= 100 millones de valores y los valores de N indicados en la siguiente tabla (ejecutar al menos 3 veces y usar el valor más pequeño, porque hay una cierta variabilidad de +/- 0,2 segundos). Recordad que los datos que se miden son de la ejecución completa del programa, incluyendo la inicialización y la ordenación. Los resultados se pueden usar para aislar el tiempo de la parte de la búsqueda a partir de los datos de la primera tabla de este documento. Es necesario compilar y ejecutar tal como se indica:

```
prompt$ icc –O3 binary_search.cpp –o search
prompt$ perf stat ./search 2 100000 100000000
```
(Option=2, N=100.000, M=100.000.000)

Search, M=100 Millones	Combine 2 searches: Ejecución completa	
	Instructions	Time (seconds)
N= 100K, 1M, 10M, 100M		

Pregunta 1g: Usar los resultados medidos para calcular la mejora (*speedup*) que se produce respecto a la ejecución single-thread del programa original. Valorar los resultados y comparar esta optimización con la optimización que usaba *prefetching*.

Pregunta 1h: Proponed una optimización que mejore aún más el tiempo de ejecución, quizás combinando o modificando alguna de las propuestas anteriores. Mostrar resultados de tiempo solamente para N= 100 millones y M= 100 millones. Para comprobar que los resultados de la nueva propuesta son idénticos a los anteriores se debe añadir a la opción el carácter 'C' para forzar la comprobación de cada valor.

Alternativamente, en lugar de proponer una versión mejorada, se puede implementar una versión equivalente a la versión con prefetching, pero sin utilizar instrucciones de prefetch, sino utilizando instrucciones "normales" de lectura a memoria. En este caso, se debe realizar una única búsqueda cada vez, en lugar de usar la estrategia de hacer dos búsquedas a la vez.