Curso 2022-2023

PRÁCTICA 1: SECUENCIAL Y MÁS

18/09/2022
Prácticas de Programación Concurrente y Paralela
Pablo Revuelta Sanz & José Ranilla

Tipo de Práctica: Abierta[1]
Entrega Documentación: Ver sección "¿Qué debe entregar el alumno?"
Duración: 1 sesión (última semana de septiembre/primera de octubre)

1 Def. **Prácticas Abiertas** son las que requieren diseñar / implementar algoritmos y la entrega de una documentación. Su duración puede ser superior a una sesión de prácticas. Tienen calificación numérica y su peso en la nota final será proporcional al número de horas presenciales / individuales.

ÍNDICE

Objetivos

Familiarizar al alumnado con la metodología de trabajo. En esta práctica los algoritmos a implementar serán secuenciales. El trabajo realizado, y los resultados obtenidos, serán reutilizados en las siguientes prácticas.

Más concretamente los objetivos son:

- Aplicar los fundamentos del análisis teórico (a priori) de los algoritmos.
- Recordar y mejorar las destrezas en la programación secuencial básica en *C* sobre entornos *Linux*, usando ficheros *makefile* y variables dinámicas.
- Usar en Python los *drivers* construidos en C.
- Incidir en los tipos de almacenamiento 1D (*row-* o *column-major orders*).
- Comparar el rendimiento de las distintas implementaciones entre sí, así como con las estimaciones teóricas.

Pérdida de Rendimiento

Son muchos los factores que influyen en la pérdida de rendimiento de los programas como, por ejemplo, no usar *middleware* (librerías, compiladores, etc.) de elevado rendimiento, falta de rigurosidad (exceso de simplicidad) de los modelos teóricos, no comprender la arquitectura de los procesadores (muy compleja incluso en los de un único núcleo), etc. Todo ello hace que obtener valores óptimos de la eficiencia sea difícil al depender de:

- Aspectos inherentes al problema (ley de *Amdahl* y otras).
- Aspectos de Diseño: granularidad, balanceado, etc.
- Aspectos específicos de la arquitectura: diferentes según sea MC, MD, SIMT.

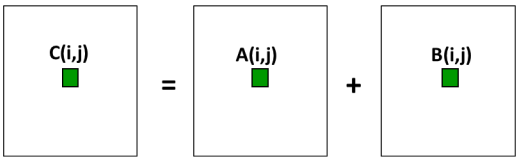
El programador, obviamente, puede mejorar/empeorar las pérdidas de rendimiento en función de cómo plasme en sus programas el conocimiento de las técnicas de HPC, conocimiento que en general es básico, cuando no inexistente.

En la actualidad, la jerarquía de memoria de los ordenadores es uno de los subsistemas que más impactan en el bajo de rendimiento. Simplificando, dos son los principios básicos que rigen el buen uso de las cachés, y son:

- La **Localidad Espacial**: acceder a posiciones consecutivas es memoria es más rápido que con otros patrones de acceso.
- La **Localidad Temporal**: acceder a datos recientemente usados también es más económico, temporalmente hablando.

Así, los algoritmos mostrados en la Figura 1, que resuelven el mismo problema ($C = A + B$), no son igual de rápidos, uno será más lento que el otro, y no siempre. En función del lenguaje de programación usado el "Algoritmo 1" será más rápido (si el lenguaje de programación es *Fortran* o afines) o más lento (se usa el lenguaje *C* o afines). Esto es debido a la forma en que cada lenguaje (familia de lenguajes) almacena los datos y cómo los recorre cada algoritmo de la Figura 1. En otras palabras: lo bien/mal que se explota la Localidad Espacial.

Algo similar sucede con la Localidad Temporal. El "Algoritmo 2" de la Figura 2 accede dos veces a cada elemento del vector *X*, igual que el "Algoritmo 1", pero lo hace en el "mismo tiempo".



Algoritmo 2

```
para (i = 0; i < n; i++)
    para (j = 0; j < n; j++)
        C[i][j] = A[i][j] + B[i][j]
```

Algoritmo 1

```
para (j = 0; j < n; j++)
    para (i = 0; i < n; i++)
        C[i][j] = A[i][j] + B[i][j]
```

Figura 1. Dos algoritmos equivalentes para el problema $\boldsymbol{C = A + B}$.

Algoritmo 1

```
s1 = 0.0;
for (j=0; j<n; j++)
   s1 = s1 + func1(X[j])
s2 = 0.0;
for (j=0; j<n; j++)
   s2 = s2 + func2(X[j])
```

Algoritmo 2

```
s1 = 0.0; s2 = 0.0;
for (j=0; j<n; j++)
{
   s1 = s1 + func1(X[j])
   s2 = s2 + func2(X[j])
}
```

Figura 2. Explotando la Localidad Temporal.

Por todo lo dicho, y otros factores, en HPC las estructuras con más de una dimensión (matrices, tensores, etc.) se suelen almacenar como vectores (estructuras 1D), fijando el programador el tipo de ***order*** o ***layout*** más interesante en función de los patrones de acceso que se precisen o sean necesarios. La idea es mejorar, o facilitar, la Localidad Espacial almacenando consecutivamente en memoria todos los elementos, como se muestra en la Figura 3.

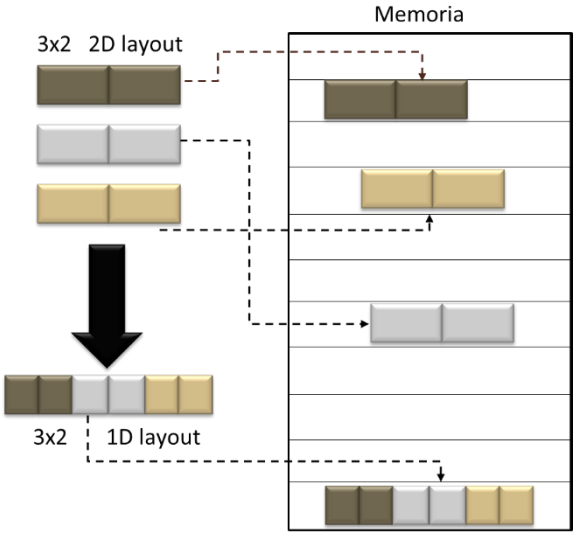


Figura 3. 3x2 2D *layout* vs. 3x2 1D *layout*.

Row- y Column-major orders (layouts)

En computación, *row-* y *column-major orders* (*layouts*) son estrategias para almacenar linealmente estructuras multidimensionales.

Formalmente, para matrices (2D), $\forall_{0 \leq i < m; 0 \leq j < n}(i,j) \in N^2$

- $row_{major}(i,j) = i * n + j$
- $colum_{major}(i,j) = j * m + i$

Es decir, en *row-major* los elementos se almacenan consecutivamente por filas en memoria, mientras que en *colum-major* la consecutividad en el almacenamiento es por columnas. Estos tipos de almacenamiento son usados de forma profusa en el intercambio de matrices entre programas escritos en diferentes lenguajes (p. ej. llamadas a librerías). También son importantes para el rendimiento porque el acceso a elementos contiguos en memoria es más rápido (Localidad Espacial; influencia de las jerarquías de memorias: las cachés).

Atendiendo a la naturaleza del problema, la forma de recorrer los elementos de las matrices, el lenguaje de programación, etc. se elige uno (*order*) u otro. Existen otros tipos de almacenamiento más complejos y de mayor rendimiento para los núcleos computacionales de estrategias más avanzadas (p. ej. almacenamiento por bloques).

Cabe resaltar que el alumnado no está habituado al uso de este tipo de proyecciones, con lo que es habitual que cometa errores al acceder a los datos (acceso al dato no deseado y/o acceso fuera del espacio de memoria reservado).

Una forma de trabajar con estas proyecciones codificando los programas en C como si fuesen estructuras 2D es recurriendo al uso de **Funciones *inline*** y ***Macros***. Ambas son herramientas habituales para aumentar la legibilidad del código sin un coste computacional elevado. En realidad, el compilador sustituye la llamada por su propio código en la generación del ejecutable.

Por ejemplo:

```
static inline int IDX(int i, int j, int lda) { return i*lda+j; }
```

o

```
#define IDX(i, j, lda) i*lda+j
```

permiten una codificación/uso como

$$M[IDX(i,j,n)]$$

siendo M una matriz de dos dimensiones almacenada como un vector. Nótese que según el almacenamiento sea por filas o por columnas el significado y uso de los parámetros en *inline* y *macros* puede (debe) cambiar.

Librerías de Álgebra Lineal y layouts

El abanico de librerías de alto rendimiento para Álgebra Lineal es amplio, existiendo productos comerciales (p. ej. la MKL de Intel incluida en el *toolbox* OneAPI), de código abierto o libre distribución (BLAS, LAPACK, OpenBLAS, ATLAS, GotoBLAS, cuBLAS, etc.), productos que las incluyen (p. ej. Matlab incluye MKL), herramientas que las usan/necesitan (p. ej. Python).

Todas ellas explotan, con mayor o menor acierto, la Localidad Espacial y Temporal, por lo que recurren al uso de proyecciones (layouts, orders) 1D, por lo general Row- y Column-major. Es decir, las estructuras multimensionales (matrices 2D en nuestro caso) deben estar almacenadas como vectores 1D, tanto las que son pasadas como argumento a las funciones de las librerías como las que retornan las propias funciones. Hay, como no, excepciones, pero la tónica general es la descrita.

Por razones históricas todas estas librerías son "*compatibles*" con column-major order, fueron inicialmente desarrolladas en *Fortran* por matemáticos, físicos, etc. En la actualidad la mayoría dispone de interfaz para que el programador indique el tipo de order, pero no todas: cuBLAS trabaja siempre con column-major order. En esta práctica, también en la práctica 3, se usará column-major order.

Las librerías descritas, además, son implementaciones paralelas y vectorizadas. Obviamente, se pueden usar en modo secuencial, pero en algunos casos no es sencillo. En otros casos lo complejo es construir (compilar, enlazar y ejecutar).

Las librerías usadas en esta asignatura son:

- *MKL* con el compilador *ICC*. Para el modo paralelo basta con usar "-qmkl=parallel" al compilar/enlazar. Para el modo secuencial "-qmkl=sequential".

- Para el compilador GCC se usará OpenBLAS, basada en GotoBLAS (GotoBLAS es una de las mejores librerías para el producto matricial). En nuestros equipos hay 3 variantes, tanto dinámicas como estáticas, de OpenBLAS, que son:
  a) Multihilo (paralela, de nombres libopenblasp.so y libopenblasp.a).
  b) Monohilo (secuencial, de nombres ibopenblas.so y libopenblas.a).
  c) OpenMP (versión adaptada para ser usada en regiones paralelas de OpenMP de nombres libopenblaso.so y libopenblaso.a).

  Para cada variante hay versión normal (libopenblasp, libopenblaso y libopenblas) y de 64bits (libopenblasp64, libopenblaso64 y libopenblas64). Más aún, en /opt/openblas está lo mismo compilado manualmente. Son equivalentes a las anteriores, que han sido instaladas con paquetes (rpm) o por dependencias (Python, más concretamente *numpy*). No se debe mezclar su uso porque, aunque son lo mismo, sus nombres, tablas de símbolos, etc. son diferentes y puede producir errores. Las de /opt/openblas son todas de 64bits y se llaman libopenblas (no cambia el nombre, cambia el directorio donde están almacenadas las versiones monohilo, multihilo y para OpenMP).
  Por ejemplo, si en Python (numpy) se usa libopenblas.so de /opt/openblas se generará un *core* porque "al no llevar los sufijos 64 y p" piensa que es la normal monohilo.

En los prototipos suministrados para las prácticas se incluirán ficheros *Makefile*, que muestran cómo compilar en cada caso, y *scripts* para SGE con las definiciones de variables de entorno (cuando sea necesario).

El Producto Matricial

El producto matricial es uno de los núcleos computacionales más usados y, por tanto, más estudiado. Es simple de entender y de codificar por el alumnado (al menos en una de sus versiones, la seleccionada para esta práctica).

Sean A, B y C tres matrices tal que $C \in R^{mxn}, A \in R^{mxk}, B \in R^{kxn}; n, m, k > 1$. El **Núcleo Computacional Producto Matricial** (PM, en adelante) se define como $C = \beta C + \alpha AB$, con α y β dos constantes $\in R$. Esta formulación es la estándar en las librerías de alto rendimiento (Lapack, Plapack, MKL, ATLAS, cuBLAS, OpenBlas, etc.). Si $\alpha = 1$ y $\beta = 0$ se obtiene la expresión estándar $C = AB$.

Un algoritmo secuencial clásico para $C = AB$, conocido como *Naïve 3-Loop*, es el que se muestra en Algoritmo 1. El alumnado debe abordar el problema general, esto es, $C = \beta C + \alpha AB$, por compatibilidad con las librerías de alto rendimiento.

Como se ha comentado, el producto matricial es la esencia de muchos problemas, pero también de métodos matemáticos. Por ejemplo, en la transparencia 24 del tema de teoría "*Evaluación Prestaciones*" se presenta un algoritmo iterativo "plagado" de productos matriciales. Ese algoritmo es una "*modificación de los profesores*" del algoritmo multiplicativo de *Lee* & *Seung* (LSA). El algoritmo LSA realiza la descomposición de una matriz en el producto de otras dos matrices de menor rango con elementos no negativos (*NMF*: *Non-negative Matrix Factorization*). LA NMF es una herramienta esencial en problemas tales como el agrupamiento automático de documentos, el análisis de datos, el tratamiento de imágenes, la separación de fuentes sonoras, la bioinformática, etc.

```
for (i=0; i<m; i++)
{
        for (j=0; j<n; j++)
        {
                C[i][j] = 0.0;
                for (r=0; r<k; r++)
                        C[i][j] = C[i][j] + A[i][r] * B[r][j];
        }
}
```

Algoritmo 1. Algoritmo secuencial *Naïve 3-Loop* con $\boldsymbol{\alpha = 1}$ y $\boldsymbol{\beta = 0}$

Producto Matricial y Localidad Espacial

La parte superior de la Figura 4 muestra el proceso, y los datos, que el algoritmo *Naïve 3-Loop* utiliza (recorre) de las matrices A y B para obtener el valor de un elemento genérico de la matriz C. Supuesto *order* por filas, el usado en el lenguaje C, los elementos de la fila matriz A usados están almacenados en memoria (o en la caché) consecutivamente (explota la Localidad Espacial), no así los elementos de la columna de B utilizada. Si el *order* es por columnas ocurre algo similar, siendo los elementos de A los que ahora no están consecutivos.

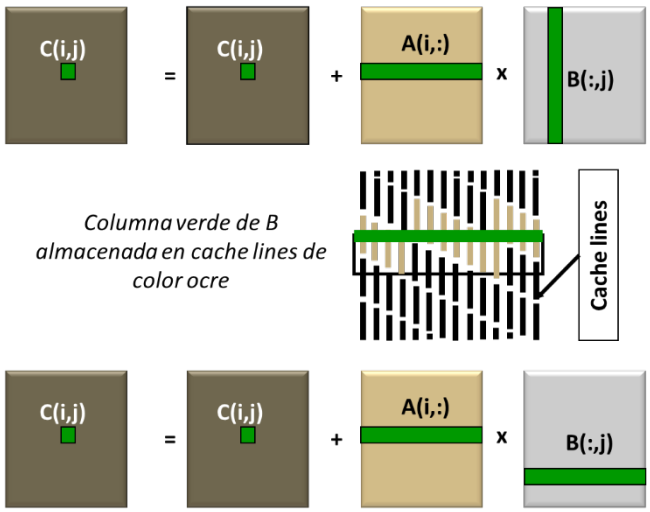


Figura 4. El Producto Matricial y la Localidad Espacial suponiendo *row-major*: Superior proceso normal; Inferior con la transpuesta de la matriz B.

Para evitar la pérdida de rendimiento que supone recorrer los elementos de B (o los de A según el caso) se puede realizar una transformación simple: trabajar sobre la transpuesta de B (o de A), obteniendo el patrón de acceso que muestra la parte inferior de la Figura 4. Así, previo al producto matricial se hace la transpuesta de la matriz B si es row-major, o de la matriz A si es column-major. Obviamente, trabajar con la transpuesta de una de las matrices implica modificar el algoritmo *Naïve 3-Loop*.

Prototipos de apoyo

En la carpeta /opt/PracticasPCP2022_2023/Practica01/ se dispone de un fichero tipo *.tgz*, de nombre PRAC01.tgz, con las ayudas para realizar la práctica. El alumno debe copiar el fichero a una carpeta vacía de su *home* y ejecutar, dentro de ella, la orden "tar zxvf PRAC01.tgz" para descomprimir. En las siguientes subsecciones se describen los detalles, y la utilidad, del contenido del fichero PRAC01.tgz.

PythonBasicos: Multiply.py

Prototipo autocontenido que implementa 2 formas de resolver $C = AB$ para distintos tamaños, variando el número de repeticiones. La primera es una versión "escalar" del Algoritmo 1 mostrado en este documento. La otra usa el operador "@" ("A @ B", que es equivalente a "np.matmul(A, B)" o "np.dot(A, B)") que lo resuelve en paralelo o secuencial, dependiendo de la configuración del entorno, y siempre vectorizando.

El prototipo, además, calcula la diferencia (norma matricial de Frobenius, también llamada de Hilbert-Schmidt) entre las soluciones obtenidas en cada planteamiento. Es un ejemplo de uso de funcionalidades Python, funcionalidades/enfoques que el alumno puede usar/copiar para sus soluciones.

Este prototipo se debe ejecutar con el *script* "EjecutaMultiply.sh". Se puede ejecutar en modo interactivo/local (en una terminal ejecutar el comando "./EjecutaMultiply.sh") o con SGE en los nodos de cálculo (orden "Colaxxxx EjecutaMultiply.sh").

PythonBasicos: ElementwiseAdd.py

Resuelve el problema "dadas 2 matrices $A \in R^{mxn}, B \in R^{mxn}; n, m > 1$ obtener otra matriz $C \in R^{mxn}$ tal que sus elementos sean la suma, elemento-a-elemento, de los elementos de las filas de A con los correspondientes elementos de las columnas de B".

Ilustra como hacer proyecciones 2D/1D (en ambos sentidos) jugando con las *layouts*, el uso del operador transpuesta, la copia de matrices, etc., operaciones todas ellas necesarias y útiles, con toda seguridad, para el desarrollo de las prácticas de la asignatura.

Como siempre, se incluye un script, de nombre "EjecutaElementwiseAdd.sh", apto tanto para la ejecución interactiva/local como con el gestor SGE.

PythonBLASviaC: BLAS.c

Ejemplo de integración entre Python, C y las librerías MKL/OpenBLAS.
El código C, de nombre BLAS.c, es un *wrapper* (función) que llama a la función cblas_dgemm (hace el producto matricial) de MKL (ficheros EjecutaMKL.sh y MakefileMKL) o de OpenBLAS (ficheros EjecutaOpenBLAS.sh y MakefileOpenBLAS). Los *makefile* suministrados crean una librería dinámica, de nombre LibBLAS.so, que será usada desde Python.

El código Python, de nombre BLAS.py, integra la librería creada (LibBLAS.so) y compara el tiempo (y la precisión) de la solución con LibBLAS.so con el del operador "@" de Python. Hay que recordar que Python (numpy) usa la librería OpenBLAS del sistema, por lo que el error entre la solución del operador "@" y la solución de LibBLAS.so cuando se enlaza con OpenBLAS debe ser menor que cuando se usa MKL.

Este ejemplo se puede ejecutar tanto en modo interactivo/local como con el gestor SGE.

PrototiposParaAlumnos: PRAC01.c

El subdirectorio "PrototiposParaAlumnos" contine:

- Script, de nombre EjecutaPrac01.sh, que se puede usar para ejecutar tanto en interactivo/local como con el gestor SGE.
- Típico fichero de cabeceras, de nombre Prototipos.h, y fichero de ordenes de compilación (Makefile). El Makefile no crea ejecutables, crea librerías dinámicas, en el directorio de nombre LIBS, para ser usadas desde Python. Makefile crea librerías compiladas con Gcc e Icc usando el *flag* de optimización -O.
- Plantilla en C, de nombre PRAC01.c, preparada para que el alumno complete con sus funciones.
- Programa Python, de nombre PRAC01.py, que ejecuta y compara las soluciones en C con las de Python.

Trabajo a realizar por el alumno

En primer lugar, debe realizar el estudio teórico, que consistirá en:

1. Calcular el TPP_{dp} (*Theoretical Peak double precision floating point Performance*) de los nodos de cálculo a usar en esta práctica, que son: a) nodos con procesador I3 (Intel®

Core™ i3-2100 CPU @ 3.10GHz con 2 núcleos), b) nodos Xeon biprocesador (Intel® Xeon® E5620 CPU @ 2.40GHz con 4 núcleos cada procesador) y c) nodos AMD (AMD Ryzen 7 3700X @ 3.6GHz con 8 núcleos).
El TPP_{dp} se calcula con las expresiones dadas en las clases expositivas o con cualquier otra formulación (en este último caso se debe adjuntar en la documentación la expresión usada y la referencia a la fuente –que debe ser rigurosa–).

2. Derivar del TPP_{dp} el tiempo por *flop* teórico mínimo[2] (constante t_c de la Ec. 1 en la transparencia del tema de teoría "*Evaluación Prestaciones*" (ecuación $T_{ar}(n,p) = nt_c$)) secuencial y paralelo.
3. Estimar la complejidad temporal y espacial de $C = \beta C + \alpha AB$.
4. Calcular la complejidad temporal y espacial del subproblema transpuesta de una matriz.

Seguidamente debe ejecutar los prototipos suministrados. Se recomienda visualizar los ficheros proporcionados y entender su filosofía. Estos prototipos tienen por objetivo facilitar el trabajo del alumno en esta práctica y las sucesivas.

En tercer lugar, el alumno tiene que implementar una función en lenguaje C para resolver el problema $C = \beta C + \alpha AB$, con α y β dos constantes cualesquiera $\in R$. Esta función debe ser codificada en el fichero PRAC01.c. y debe resolver:

- El supuesto base, el mostrado en Algoritmo 1 (pero para α y β dos constantes cualesquiera).
- El caso en el que se use la transpuesta de A (ver sección anterior).

La función será secuencial y utilizará column-major order. Nótese que se habla de función, no de programa. Esto es así porque el objetivo es construir una librería dinámica (.so) para ser usada desde Python, desde el programa PRAC01.py.

La función debe tener el siguiente prototipo:

double MyDgemm(int, int, int, int, double, double*, int, double*, int, double, double*, int);

donde

- Se retorna el tiempo empleado, un número real. El tiempo se calcula con la función definida en el fichero "Prototipos.h" y construida en el fichero "PRAC01.c".
- El primer argumento, un entero, sirve para seleccionar entre el supuesto básico o la variante con la transpuesta de A. En el fichero "Prototipos.h" hay dos definiciones ("Normal" y "TransA") para ser usadas con/en este argumento.
- El resto de los argumentos tienen el mismo tipo y significado que los correspondientes en cualquiera de las librerías mencionadas (son sus 11 últimos argumentos). Para una descripción más completa buscar en *internet* la cadena "cblas_?gemm" (llevará a www.intel.com) y/o "cublasDgemm" (llevará a https://docs.nvidia.com/cuda/cublas).
- La función MyDgemm recibe las matrices en su forma original. Esto es, la variante que trabaja con la transpuesta de A debe hacer, internamente, la operación "transpuesta de A".

2 La inversa de la velocidad, esto es, la inversa de TPP_{dp}. Recordar que TPP_{dp} está expresado en Gflops.

En relación con la experimentación, por ahora el alumno solo debe garantizar que sus códigos son correctos, coherentes, correctamente integrados y que ejecutan sin problema. También debe comprobar que la ejecución en los nodos de cálculo (SGE) concluye correctamente. En sucesivas prácticas se especificará qué y cómo se debe experimentar.

¿Qué debe entregar el alumno?

La memoria, documento en formato PDF, de esta práctica se subirá al Campus Virtual junto con la documentación de la práctica 3.

El fichero PRAC01.c, modificado por el alumno, es el único fichero que hay que entregar en esta práctica. En la corrección de la práctica se usará un programa Python similar a PRAC01.py. En otras palabras, todo lo que el alumno desarrolle debe estar en este fichero y la función MyDgemm, a la que NO se le puede alterar el prototipo, será la única interfaz con Python y con las funciones que el alumne cree. En cualquier caso, debe dejarse en el directorio /home/PCPxxxxx/Practica01.

En resumen, por ahora el alumno no debe entregar nada.