Conceptos y Métodos de la Computación Paralela (CMCP)

Prácticas

José E. Román

Índice

1. OpenMP

Para las prácticas del bloque de OpenMP se dispone de la máquina `knights.dsic.upv.es` (aunque en realidad se pueden hacer en cualquier otra máquina con varios núcleos).

Nota: si se utiliza alguna máquina distinta a las proporcionadas, se debe incluir en el documento entregado un resumen de las características del hardware.

1.1. Uso básico

Para compilar un programa OpenMP podemos utilizar cualquier compilador que lo soporte, por ejemplo el de GNU o el de Intel. Con el compilador de GNU:

```
$ gcc -fopenmp -Wall -o nthreads1 nthreads1.c
```

donde la opción `-Wall` se recomienda para que el compilador muestre todos los *warnings* que encuentre. También se podría añadir la opción `-O` para compilación optimizada.

En el caso del compilador de Intel sería como sigue

```
$ icc -qopenmp -Wall -o nthreads1 nthreads1.c
```

A la hora de ejecutar el programa, se puede indicar el número de hilos mediante la variable de entorno correspondiente, bien asignando el valor en la misma línea

```
$ OMP_NUM_THREADS=4 ./nthreads1
```

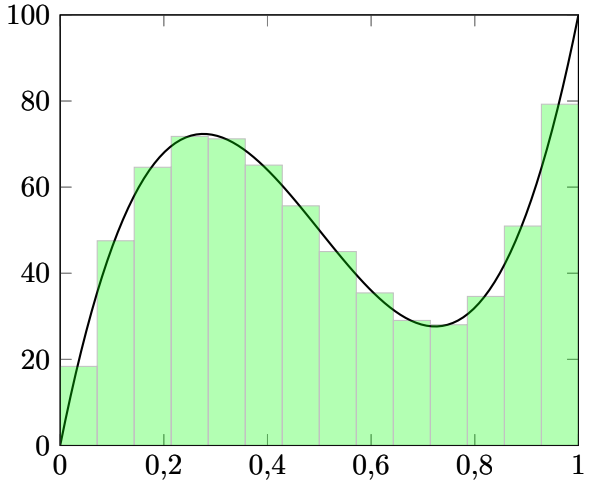

Figura 1: Interpretación geométrica de una integral.

o bien exportando la variable

```
$ export OMP_NUM_THREADS=4
$ ./nthreads1
```

E1 **Ejecución de programas**

Nota: No es necesario incluir este ejercicio en el portafolio.

Ejecutar los programas `nthreads1.c`, `nthreads2.c` y `nthreads3.c`, observando la salida. El objetivo es entender el funcionamiento de la directiva `parallel`, y cómo actúan las diferentes formas de cambiar el número de hilos. Repetir las ejecuciones varias veces para observar el indeterminismo de la salida.

1.2. Paralelización de bucles

E2 **Integración numérica**

Consideramos el problema del cálculo numérico de una integral de la forma

$$\int_a^b f(x)dx.$$

La técnica que se va a utilizar para calcular numéricamente la integral es sencilla, y consiste en aproximar mediante rectángulos el área correspondiente a la integral, tal y como puede verse en la Figura 1. Se puede expresar la aproximación realizada de la siguiente forma

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} f(x_i) \cdot h = h \cdot \sum_{i=0}^{n-1} f(x_i), \tag{1}$$

donde n es el número de rectángulos considerado, $h = (b-a)/n$ es la anchura de los rectángulos, y $x_i = a + h \cdot (i + 0{,}5)$ es el punto medio de la base de cada rectángulo. Cuanto mayor sea el número de rectángulos, mejor será la aproximación obtenida.

El código del programa secuencial que realiza el cálculo se encuentra en el fichero `integral.c`, donde el bucle de la función `calcula_integral` es el que se corresponde al sumatorio que aparece en la ecuación (1). Hay que paralelizar este bucle mediante OpenMP, comprobando que el resultado obtenido es correcto.

En este caso, para compilar el programa hay que añadir la librería matemática:

```
$ gcc -fopenmp -Wall -o integral integral.c -lm
```

En el programa paralelo, se recomienda imprimir el número de hilos utilizado para asegurarse de que tiene los esperados (y lo mismo en todos los ejercicios de esta sección). Lo mejor es hacerlo mediante una región paralela específica para ello en la función `main`.

E3 **Producto matricial**

Dadas dos matrices cuadradas, $A, B \in \mathbb{R}^{n \times n}$, su producto se define como $C = AB$, de forma que $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. El archivo `matmat.c` contiene un programa en lenguaje C que crea dos matrices aleatorias e invoca a la función `matmat` que las multiplica.

Tomando como base la versión secuencial, paralelizar el programa del producto de matrices utilizando directivas OpenMP. Incluir en el programa instrucciones para mostrar el tiempo empleado en la realización del cálculo, mediante `omp_get_wtime()`.

En este ejercicio, se pretende paralelizar a nivel de bucle. Dado que hay tres bucles anidados, probar diferentes posibilidades. Ejecutar para diferente número de hilos y observar la mejora obtenida. Utilizar tallas del problema suficientemente grandes para que los tiempos sean significativos, por ejemplo

```
$ OMP_NUM_THREADS=4 ./pmatmat 1000
```

También es conveniente hacer varias repeticiones del cálculo. Hay que tener en cuenta que `knights` no dispone de colas de ejecución, por lo que si hay varios usuarios ejecutando a la vez los tiempos dejarán de tener validez.

Nota: para comprobar que el resultado de la paralelización es correcto, utilizar tamaños de matriz pequeños, ya que para tamaño 1000 o mayor no se realiza la comprobación.

1.3. Regiones paralelas

E4 **Fractales–Paralelización con Regiones Paralelas**

El programa contenido en el directorio `fractal` genera fractales de Mandelbrot a partir de los datos proporcionados. Para que el cálculo no sea excesivamente rápido, hay que escoger los parámetros adecuadamente, por ejemplo:

```
$ ./mandel -width 2048 -height 1536 -ulx -1.37 -uly 0.04 -lly -0.01
$ ./mandel -width 2048 -height 1536 -ulx -1.37 -uly 0.014 -lly 0.0135
$ ./mandel -width 2048 -height 1536 -ulx -1.369538 -uly 0.013797 -lly 0.01368
```

Si fuera necesario, se puede incrementar la resolución de la imagen generada.

Paralelizar el programa utilizando regiones paralelas y construcciones de reparto de trabajo. Realizar pruebas de prestaciones y probar diferentes estrategias de planificación (cláusula `schedule`). Se recomienda utilizar `schedule(runtime)` en combinación con la variable de entorno `OMP_SCHEDULE`.

Nota: comprobar que la figura generada en paralelo coincide con la del código original.

---

2. CUDA

Para las prácticas del bloque de CUDA se dispone de la máquina `gpu.dsic.upv.es`, que tiene instaladas dos GPUs de NVIDIA.

2.1. Uso básico

El objetivo inicial es familiarizarse con el entorno. Para ello, compilaremos y ejecutaremos un programa que proporciona información sobre la(s) GPU(s) del sistema. Posteriormente, pasaremos a implementar diferentes programas que usan *kernels* para realizar cálculos en la GPU, con complejidad creciente.

Para compilar un programa CUDA se utiliza el compilador de NVIDIA, `nvcc`:

```
$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Wed_Oct_23_19:24:38_PDT_2019
Cuda compilation tools, release 10.2, V10.2.89
```

Estrictamente, el compilador `nvcc` se utiliza para compilar los ficheros con extensión `.cu`. En un programa más avanzado se combinan ficheros `.cu` con ficheros de C o C++ normales, aunque para simplificar, utilizaremos la extensión `.cu` en todos los ficheros empleados en esta sección.

La ejecución de programas que contienen código CUDA se realiza como un programa convencional.

Nota: El comando `nvidia-smi` proporciona información sobre el estado de cada GPU, y los procesos que se están ejecutando en ese momento.

E5 Información de la GPU

En el directorio `dquery`, compila y ejecuta el programa.

```
$ nvcc -o dquery dquery.cu
```

La salida del programa proporciona información acerca de las GPUs instaladas y sus características. Este programa es una versión simplificada de `deviceQuery.cpp`, disponible en el SDK de CUDA, concretamente en el directorio `/usr/local/cuda/samples/1_Utilities/deviceQuery`. Modifica el programa para que muestre también otra información, como la anchura del bus de memoria (consulta la documentación online de NVIDIA).

Nota: En este caso, el programa no incluye ningún *kernel* sino que únicamente se utilizan funciones del *runtime* de CUDA, por lo que no sería necesario que tuviera la extensión `.cu` y podría compilarse con el compilador de C++ (enlazando las librerías adecuadas, en este caso `-lcudart`).

2.2. Ejecución de kernels

E6 Suma de vectores

El objetivo es hacer un programa sencillo `vecadd` que sume dos vectores en la GPU. Se proporciona el esqueleto del programa, y hay que completar lo siguiente:

1. Código del *kernel* para sumar los vectores.
2. Reservar la memoria en la GPU.
3. Copiar datos de la CPU a la GPU.
4. Lanzar el *kernel*.
5. Copiar resultado de la GPU a la CPU.
6. Liberar la memoria reservada en la GPU.

Se proporciona un `makefile` para facilitar la compilación.

E7 Suma de matrices

El programa `matadd` es similar a `vecadd`, pero para sumar matrices en lugar de vectores. Vamos a hacer pruebas con diferentes tipos de bloques de hilos, comparando el rendimiento. Realiza los siguientes cambios:

1. Completa el código del kernel para realizar la suma de matrices. Esta primera versión usa una malla bidimensional de bloques de hilos bidimensionales
2. Prueba con diferentes tamaños de matriz y de bloque de hilos. Ten en cuenta que no se puede superar el límite de hilos por bloque, por ejemplo, prueba 256×256 y verás que da error al lanzar el kernel.
3. Modifica el programa para que se haga un particionado unidimensional, es decir, tanto el grid como los bloques de hilos pasan a ser unidimensionales. En este caso, en el kernel, en lugar de eliminar los dos bucles, eliminamos uno de ellos y mantenemos el otro (cada hilo suma todos los elementos de su fila). ¿Cómo es el rendimiento en comparación a la versión anterior?
4. (OPCIONAL) En la última versión, vamos a combinar un grid bidimensional con bloques de hilos unidimensionales. Este caso es similar a la primera versión, pero los bloques de hilos van a tener 1 en una de las dimensiones.

E8 Producto matriz por vector

Vamos a comprobar cómo el uso de memoria compartida puede acelerar los programas, concretamente con el ejemplo de producto matriz por vector explicado en clase. Se proporciona una versión básica del kernel en el directorio `matvec`.

1. Implementa una versión modificada del kernel basada en el código analizado en clase.
2. Implementa una versión de simple precisión para comparar las prestaciones con respecto de la versión de doble precisión. Para ello, sustituir `double` por `float`, y en la función `verify` también `fabs` por `fabsf`, y la tolerancia `1e-13` por `1e-5`. *Nota*: Al trabajar en simple precisión las variables de coma flotante ocupan la mitad de espacio en memoria, por lo que se puede incrementar más el tamaño del problema. Para tamaños ya muy grandes, como `n=50000`, el programa deja de funcionar. La explicación es que en ese caso el acceso `A[i+j*n]` es incorrecto porque el índice `i+j*n` desborda (excede la capacidad de los enteros con signo de 32 bits). La solución sería forzar a calcular este índice con 64 bits, por ejemplo con `A[i+j*(size_t)n]`.

E9 Multiplicación de matrices

En este ejercicio vamos a hacer pruebas con el producto de matrices. En lugar de crear un nuevo programa para ello, vamos a hacer uso de una implementación ya disponible en los ejemplos de CUDA. En `/usr/local/cuda/samples/0_Simple` hay dos versiones del producto de matrices, una similar a la estudiada en clase usando memoria compartida (`matrixMul`) y otra haciendo una llamada a la librería cuBLAS (`matrixMulCUBLAS`). El objetivo será comparar las prestaciones de las dos versiones para diferentes tamaños de matriz.

Nota: Para compilar estos programas los tendréis que copiar a vuestro directorio. Los programas incluyen archivos de `/usr/local/cuda/samples/common`, pero no es necesario copiar este directorio si se compila de la siguiente forma:

```
$ cp -r /usr/local/cuda/samples/0_Simple/matrixMul $HOME/cmcp/cuda
$ cd $HOME/cmcp/cuda/matrixMul
$ make INCLUDES=-I/usr/local/cuda/samples/common/inc
```

Nota: Para que la comparación sea significativa, habría que usar matrices del mismo tamaño en ambos ejemplos. Esto puede requerir modificar el código fuente de alguno de ellos.

3. MPI

Para las prácticas del bloque de MPI se dispone del cluster `kahan.dsic.upv.es`.

3.1. Uso básico

La compilación de un programa MPI se realiza mediante unos comandos especiales (`mpicc`, `mpif90`, etc.) que son simplemente *wrappers* del compilador correspondiente que añaden las opciones y/o librerías necesarias para MPI. Por ejemplo,

```
$ mpicc -Wall -o hellow hellow.c
```

Para consultar qué opciones y/o librerías se están usando, se puede ejecutar con esta opción

```
$ mpicc -show
```

Para ejecutar el programa también tenemos un comando específico, `mpiexec`, con el que indicamos cuántos procesos MPI se deben crear. Por ejemplo,

```
$ mpiexec -n 2 ./hellow
```

Importante: el comando anterior lanza dos procesos en el *front-end* de `kahan`, no en las colas de trabajos. Para ejecutar en las colas, seguir las instrucciones en `http://personales.upv.es/jroman/kahan.html`.

E10 **Ejecución de programas**

Ejecutar el programa `hellow.c` en `kahan`, con diferente número de procesos, tanto en el *front-end* como en las colas. El objetivo es familiarizarse con el uso del cluster.

3.2. Comunicación punto a punto

E11 **Comunicación en anillo**

El programa proporcionado `ring_c.c` realiza una comunicación punto a punto entre varios procesos dispuestos en anillo, es decir, cada proceso envía un mensaje al siguiente proceso (cuyo índice de proceso es uno más que el suyo), y el último proceso lo envía nuevamente al proceso 0. Analiza el código para entender cómo funciona, y ejecútalo en el cluster con diferente número de procesos.

Modifica el programa para que el proceso 0 imprima el tiempo transcurrido entre el primer envío y la última recepción, utilizando para ello `MPI_Wtime`.

E12 **Cálculo de Pi**

El programa proporcionado `cpi.c` calcula el valor de π mediante integración numérica de la función $f(x) = \frac{4}{1+x^2}$ en el intervalo $[0, 1]$, de forma similar a lo explicado anteriormente (Figura 1). La paralelización se basa en que cada proceso MPI suma únicamente una parte de los rectángulos, y finalmente se realiza una comunicación para obtener la suma total.

Nota: el programa lee el valor de n de la consola, por lo que no es conveniente ejecutar este programa en las colas.

El código del ejemplo realiza la recepción "por orden", es decir, el proceso 0 ejecuta un bucle desde `p` igual a 1 hasta el número de procesos menos uno, y hace una operación `MPI_Recv` para ese proceso concreto. Dado que los mensajes no tienen por qué llegar ordenados, el objetivo de este ejercicio es cambiar el código para que se reciba de cualquier proceso. Incluir también una llamada a `printf` para imprimir por cada recepción qué valor se ha recibido y de qué proceso.

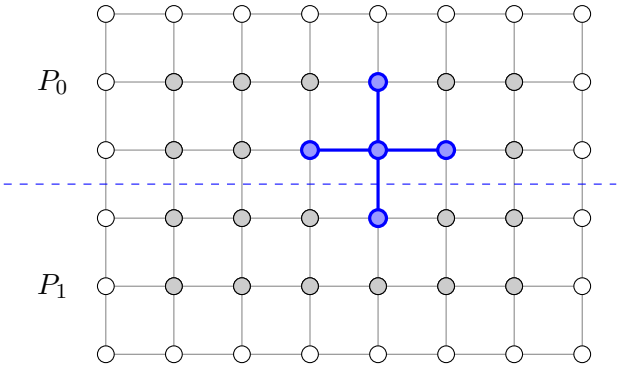

Figura 2: Malla repartida entre dos procesos, donde se indica un caso en que es necesaria la comunicación, ya que P_0 necesita valores que están en P_1. Los puntos grises representan las incógnitas del problema, y los puntos blancos son las condiciones de frontera (cuyo valor es constante, p.e. cero).

E13 Ecuación de Poisson - Particionado Unidimensional Vertical

El programa proporcionado `poisson.c` resuelve la ecuación de Poisson $\nabla^2 u = f$ en un dominio rectangular mediante el método de diferencias finitas centradas, utilizando una malla de discretización en la que se divide el eje horizontal en $M + 1$ subintervalos y el vertical en $N + 1$ subintervalos, resultando en un total de $M \times N$ puntos interiores (incógnitas). El sistema de ecuaciones resultante, $Ax = b$, se resuelve mediante el método iterativo estacionario de Jacobi sin construir explícitamente la matriz A. En cada iteración de Jacobi, se actualiza el valor en cada uno de los puntos de la malla haciendo una media ponderada con el valor de los cuatro vecinos.

El objetivo de este ejercicio es realizar la paralelización con MPI. Paralelizar el cálculo dividiendo el dominio en bandas horizontales, una por proceso (ver Figura 2). Cada subdominio deberá tener también una orla de valores alrededor (*ghost values*) para almacenar los valores de la frontera real o bien los valores recibidos de otro proceso (frontera ficticia), ver Figura 3.

La paralelización con MPI normalmente implica la distribución explícita de las estructuras de datos entre los procesos. En este caso, las variables asociadas a la malla (`x`, `b`, `t`) están distribuidas, es decir, cada proceso únicamente almacenará una parte. En este ejercicio, cada proceso tiene únicamente $n_{\text{local}} + 2$ filas, a las cuales se accederá utilizando índices locales (no índices globales). En este algoritmo no se utiliza el índice de fila explícitamente en el cálculo, por lo que no es necesario calcular los índices globales en ningún momento.

En este ejercicio vamos a paralelizar únicamente la función `jacobi_step`, por lo que habría que llamar directamente a esta función desde el programa principal, o bien anular provisionalmente el bucle que hay en la función `jacobi_poisson`. La implementación paralela debería seguir el siguiente esquema:

```
jacobi_step_parallel(N,M,x,b,t):
  send line n_local of my x to neighbour below
  receive line from my neighbour above and store it in line 0 of my x
  send line 1 of my x to neighbour above
  receive line from my neighbour below and store it in line (n_local+1) of my x
  for i=1:n_local
    for j=1:M
      compute t(i,j) as in the sequential algorithm
    end
  end
```

Nótese que las variables `x`, `b`, `t` son arrays unidimensionales que representan arrays bidimensionales almacenados por filas (en memoria se almacena la primera fila, seguida de la segunda

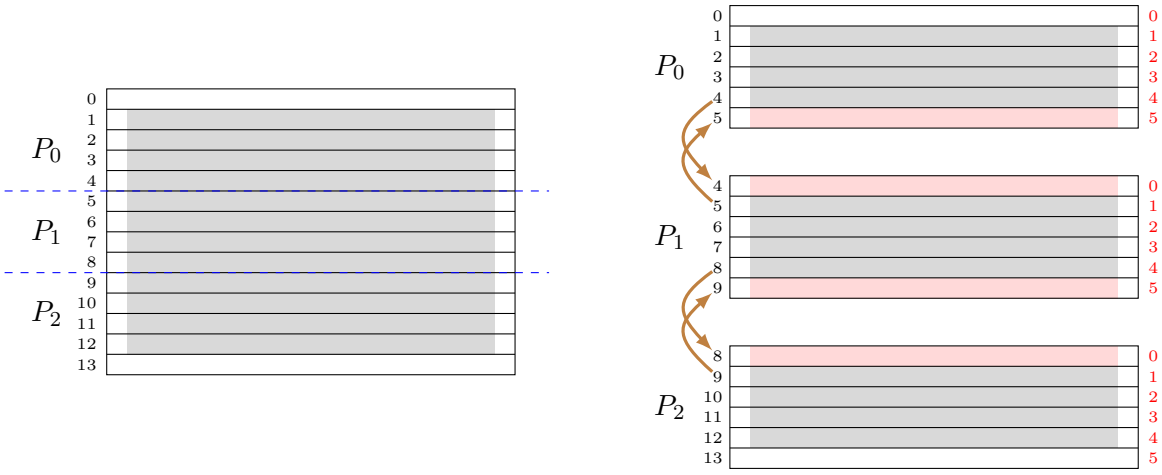

Figura 3: Ejemplo de reparto de la malla en la versión paralela para tres procesos MPI. A la izquierda se muestra la malla original (secuencial), y a la derecha el array local con que trabaja cada proceso MPI, de dimensión $(n_{\text{local}} + 2) \times (M + 2)$, donde $n_{\text{local}} = N/p$. Las dos filas extra contienen o bien las condiciones de frontera (en blanco) o bien los valores recibidos del proceso vecino (*ghost values*, en color rosado). La numeración en negro corresponde a índices globales, mientras que la numeración en rojo son índices locales.

fila, etc). La variable `ld` contiene el *leading dimension*, es decir, la separación entre un elemento de la malla y el elemento que ocupa la misma posición en la fila siguiente.

Nota: Para simplificar, se puede hacer que N sea un múltiplo del número de procesos, de forma que el tamaño local n_{local} sea igual en todos los procesos.

Nota: Usar comunicación punto a punto para intercambio de valores de la malla, intentando resolver el problema de la secuencialización.

3.3. Comunicación colectiva

E14 Cálculo de Pi con colectivas

Vamos a modificar el programa `cpi.c` utilizado anteriormente. En este caso, el ejercicio consiste en sustituir los fragmentos de código marcados como comunicación, que están implementados mediante primitivas de envío y recepción punto a punto, por primitivas de comunicación colectiva de MPI, de forma que una sola llamada sea equivalente al código reemplazado.

E15 Ecuación de Poisson - Programa Completo

Vamos a modificar el programa de Poisson paralelo realizado anteriormente, en este caso paralelizando el programa completo, es decir, la función `jacobi_poisson` incluyendo el bucle. En este bucle se realiza un cálculo para el criterio de parada, para el que en el caso paralelo habría que utilizar una primitiva de comunicación colectiva.

Además de la comunicación colectiva en el criterio de parada, añadir el código necesario para que al final de la ejecución (tras la llamada a la función `jacobi_poisson`) todos los procesos envíen al proceso 0 su fragmento de `x`, de forma que el proceso 0 tenga la solución completa y pueda imprimirla. Esto se debe hacer también mediante una primitiva de comunicación colectiva.

3.4. Tipos de datos derivados

E16 Ecuación de Poisson - Particionado Unidimensional Horizontal

En este ejercicio se pretende paralelizar nuevamente el programa de Poisson, pero este vez

utilizando un particionado por columnas en lugar de filas. Es decir, en este caso cada proceso tendrá un array local de dimensiones $(N+2) \times (m_{\text{local}}+2)$, y tendrá que realizar comunicaciones con los procesos vecinos de su izquierda y su derecha.

Los arrays deben seguir estando almacenados por filas, siendo en este caso el *leading dimension* igual a $(m_{\text{local}}+2)$. La diferencia fundamental en este caso, con respecto del caso anterior, es que los elementos que se intercambian con los procesos vecinos no están almacenados de forma contigua en memoria, por lo que será necesario definir un tipo de datos derivado con `MPI_Type_vector`.

Nota: La definición de los tipos de datos derivados deben realizarse una sola vez, no cada vez que se van a usar, y deben liberarse cuando ya no sean necesarios.

Pista: En el caso de definir un tipo de datos derivado para enviar columnas, si se quiere enviar más de una columna (por ejemplo, `MPI_Send(A,2,coltype,...)`), es necesario modificar la *extensión* del tipo mediante `MPI_Type_create_resized`, de forma que tras enviar una columna se posicione en el primer elemento de la siguiente columna y no en el último elemento de la columna anterior. Esto no es necesario en este caso para paralelizar el cálculo, pero sí para que al final el proceso 0 reciba todas las columnas calculadas por los demás procesos.