Conceptos y Métodos de la Computación Paralela (CMCP)

Trabajos

José E. Román

Índice

Instrucciones

El trabajo se realizará de forma individual o en grupos de dos alumnos. Cada grupo debe elegir uno de los trabajos, correspondientes a cada una de las entradas del índice que aparece en esta página (por ejemplo, 2.1). Una vez elegido el trabajo hay que comunicarlo al profesor (por correo electrónico) para recibir la información adicional que sea necesaria.

Todos los trabajos deben presentarse públicamente al resto de la clase. Las presentaciones tendrán lugar durante la última clase programada. La duración recomendada de cada presentación es de 15 minutos. Si el trabajo ha sido realizado por dos alumnos, ambos deben participar en la presentación, y ambos deben ser capaces de contestar a cualquier pregunta relacionada con el desarrollo del trabajo.

La entrega de la tarea consiste en el fichero de las transparencias de la presentación (en PDF), junto con el código fuente de los programas desarrollados. No es necesario escribir una memoria formal. Una posible estructura para la presentación sería:

1. Descripción del problema.
2. Descripción de la paralelización realizada.
3. Detalles de implementación.
4. Validación (comprobación de que la paralelización es correcta).
5. Estudio de prestaciones paralelas.

Para el estudio de prestaciones, hay que tener en cuenta lo siguiente:

- Hay que medir el tiempo del fragmento de código relevante, no de todo el programa. La medición de tiempos se debe hacer con primitivas estándar de OpenMP, MPI o CUDA, según sea el caso.
- Es recomendable realizar el estudio de prestaciones con una compilación optimizada (opción `-O` al compilar). Indicar las opciones de compilación utilizadas.
- Se recomienda llevar a cabo estudios de escalabilidad fuerte y débil.
- Para la escalabilidad fuerte, hay que elegir el tamaño de problema suficientemente grande para que los tiempos sean significativos. También se puede optar por repetir el estudio de escalabilidad fuerte para varias tallas de problema.

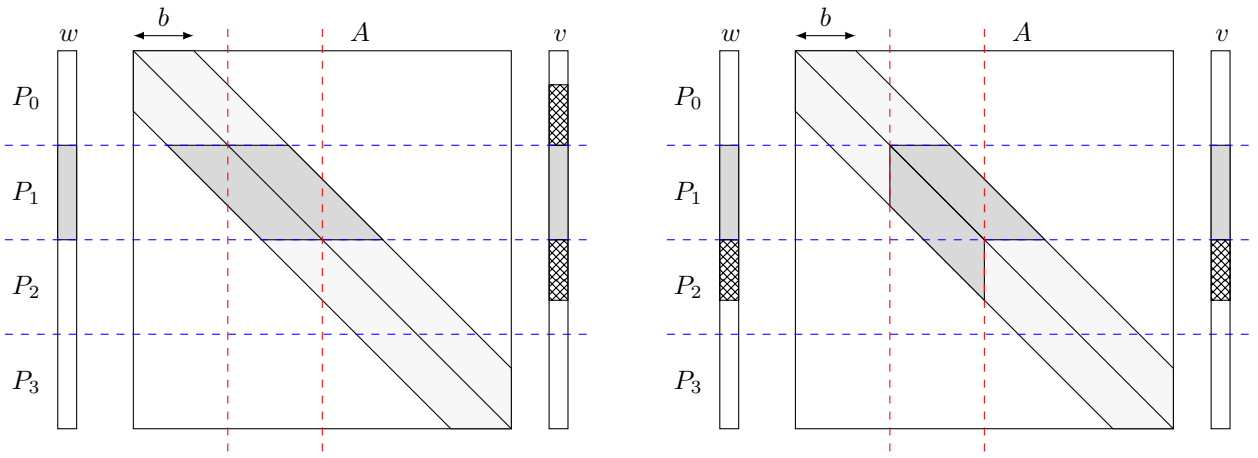

Figura 1: Esquema del producto matriz-vector $w = Av$, donde A es una matriz banda con ancho de banda b. Se ilustra el caso de 4 procesos. La parte sombreada indica los datos almacenados localmente en el proceso P_1, y las zonas rayadas indican datos pertenecientes a otros procesos que implican comunicación con P_1. El esquema de la derecha es una modificación para el caso de almacenamiento simétrico.

- Para la escalabilidad débil, según el algoritmo hay que ver cómo se debe incrementar el tamaño del problema al incrementar el número de procesadores utilizados. El objetivo es que el trabajo por proceso se mantenga constante.
- En el caso de paralelización con MPI, en principio se debe utilizar el cluster `kahan`. Si se lanza un proceso por nodo, las ejecuciones están limitadas a 4 procesos (4 nodos). Para escalar a más procesos, hay que lanzar varios procesos en cada nodo.
- Las ejecuciones en `kahan` deben realizarse a través del sistema de colas. Esto garantiza un uso exclusivo de los nodos reservados. Sin embargo, en `knights` y `gpu2` no hay sistema de colas, por lo que al tomar tiempos hay que asegurarse de que no hay otros usuarios realizando ejecuciones. Se recomienda no esperar al último momento para llevar a cabo las ejecuciones.

1. Producto matriz-vector con matriz banda

Para este proyecto se proporciona la versión secuencial `matvec.c` del producto matriz-vector $w = Av$ de una matriz banda cuadrada $A \in \mathbb{R}^{N \times N}$ con ancho de banda b. La estructura banda de la matriz se refiere a que el elemento a_{ij} es cero si $|i - j| > b$.

En una implementación real, únicamente se almacenaría la parte de la matriz correspondiente a la banda (elementos no nulos). Sin embargo, para simplificar, el programa `matvec.c` utiliza un almacenamiento denso por filas, es decir, se reserva espacio también para los ceros.

El producto matriz-vector es la operación básica utilizada en muchos métodos iterativos para diferentes problemas de álgebra lineal. En estos métodos se repite la operación de producto matriz-vector muchas veces. Para simplificar, en nuestro caso vamos a analizar las prestaciones de un bucle simple que ejecute el producto matriz-vector (secuencial o paralelo) un número determinado de veces, por ejemplo 1000 o 10000.

Versión paralela básica Habría que desarrollar una versión paralela `matvecmpi.c`, en la que la distribución de la matriz se realice por bloques de filas contiguas, siendo el número de filas locales a cada proceso, n, aproximadamente el mismo pero no necesariamente igual en todos los procesos. Para simplificar la paralelización, se puede asumir el caso sencillo en que el ancho de banda es siempre menor que el tamaño local de la matriz, n.

En la Figura 1 (izquierda) se muestra el esquema que debe seguir el código paralelo. Básicamente, antes de realizar el cálculo de su porción local del vector w, cada proceso debe recibir b elementos del vector v del vecino de arriba y b elementos del vecino de abajo. Asimismo, cada proceso deberá enviar b elementos de su vector v al proceso de arriba y b elementos al proceso de abajo. Esta comunicación debe realizarse de forma

que se evite la secuencialización y/o el interbloqueo, por ejemplo, utilizando primitivas de comunicación combinada (`MPI_Sendrecv`). Habitualmente, la implementación paralela se haría de forma que cada proceso almacena n filas completas de la matriz y n elementos del vector w. En el caso del vector v, se reservaría memoria para $n + 2b$ elementos, es decir, los n elementos locales más b recibidos de cada uno de los vecinos.

En el estudio de prestaciones, se debería considerar tanto el tamaño de la matriz N como el ancho de banda b, ya que este último parámetro influye tanto en el coste aritmético como en la longitud de los mensajes.

1.1. Versión MPI con solapamiento

Además de la versión paralela básica descrita antes, el objetivo de este proyecto es hacer una versión que intente mejorar las prestaciones paralelas solapando la comunicación con parte de la computación. Esto es posible porque hay parte del cálculo (correspondiente a las columnas que coinciden con el rango local de filas) que no depende de los datos de otros procesos. Para ello, hay que utilizar primitivas de comunicación no bloqueantes. El algoritmo quedaría así:

```
matvec_parallel(N,n,b,A,v,w):
  start send to neighbour above
  start send to neighbour below
  start receive to neighbour above
  start receive to neighbour below
  compute partially the local vector w, using only local columns of A and local elements of v
  finish all pending communication operations (MPI_Waitall)
  complete computation of w for remaining columns of A
```

Realizar un estudio de prestaciones de la versión básica y de la versión mejorada, para ver si en este caso se nota una mejora sustancial o no.

1.2. Versión MPI para almacenamiento simétrico

Además de la versión paralela básica descrita antes, el objetivo de este proyecto es implementar una versión simétrica. En matrices simétricas es posible almacenar solo la parte triangular superior, por ejemplo. Cada proceso posee localmente un rango de filas (parte superior) y le corresponden también los elementos simétricos, por lo que el esquema de comunicación cambia (ver Figura 1, derecha). Adaptar el código paralelo a este caso.

2. Métodos estacionarios para la ecuación de Poisson

Para este proyecto partimos del programa `poisson.c`, ya utilizado en las prácticas, que 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.

Durante las prácticas se realizó una versión paralela basada en el particionado unidimensional vertical del dominio (Figura 2), y otra versión con particionado unidimensional horizontal. Se puede utilizar cualquiera de ellas como punto de partida para este proyecto.

En el problema de Poisson, para la validación puede ser apropiado realizar una visualización gráfica de la solución calculada.

Nota: En todos los proyectos relacionados con la ecuación de Poisson, hay que tener en cuenta que al aumentar el tamaño del problema las iteraciones necesarias para la convergencia (bucle while) crecen. Esto puede ser un problema a la hora de realizar el estudio de escalabilidad, especialmente la escalabilidad débil. En este sentido, se recomienda que el estudio de prestaciones se lleve a cabo limitando el número de iteraciones de ese bucle, y que sea constante en todas las pruebas realizadas.

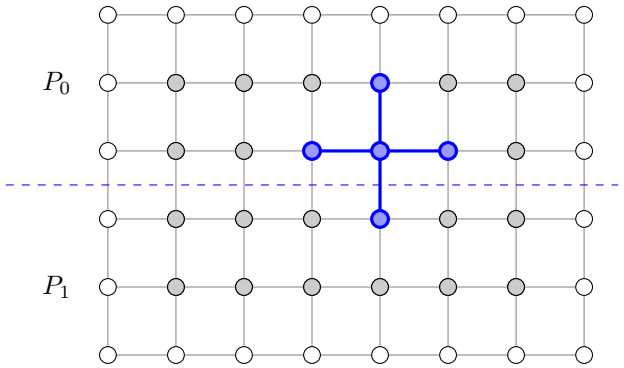

Figura 2: Malla repartida entre dos procesos, donde se indica un caso en que es necesaria la comunicación.

2.1. Versión MPI con solapamiento y comunicación persistente

De forma similar al apartado 1.1, el objetivo de este proyecto es modificar el código paralelo para poder realizar un solapamiento de la comunicación y el cálculo. En el caso de Poisson, se inician las comunicaciones con los dos procesos vecinos y, mientras se transmiten los mensajes, el proceso debe realizar el cálculo asociado a los puntos interiores de la malla, es decir, todos menos la primera y última fila (o columna). Después de este cálculo se espera a que lleguen los mensajes pendientes y se completa el cálculo con los valores recibidos.

Además, en este trabajo se van a realizar las comunicaciones mediante primitivas persistentes, como `MPI_Startall` y otras. Dado que en cada iteración las comunicaciones a realizar son similares, la definición de las comunicaciones (`MPI_Send_init`, etc.) se debe hacer una única vez, fuera del bucle. Realizar una comparativa de la versión modificada con la original, para determinar si el solapamiento y el uso de primitivas persistentes ofrecen una mejora significativa en este caso.

2.2. Versión híbrida MPI-OpenMP

Partiendo de la versión paralela MPI, añadir la paralelización del "cálculo local" mediante OpenMP. De esa forma, tendremos dos niveles de paralelismo, un paralelismo de grano más grueso a nivel de subdominios (MPI) y otro paralelismo de grano más fino a nivel de bucles (OpenMP). Realizar un estudio de prestaciones que incluya los siguientes aspectos:

- Comparar la versión OpenMP pura con la MPI pura ejecutándose en memoria compartida en la misma máquina.
- Estudiar la escalabilidad de la versión híbrida MPI-OpenMP usando varios nodos del cluster `kahan`. En este caso, el tamaño del problema debe ser suficientemente grande para que el tamaño local justifique la paralelización con hilos.

Hay que tener en cuenta que para obtener un buen rendimiento en la parte de OpenMP es necesario minimizar el número de veces que se activan y desactivan los hilos. Esto implica usar una única región paralela, que además debería abarcar la mayor cantidad de código posible (ley de Amdahl).

Nota: si al hacer la paralelización con hilos, resulta que hay alguna primitiva MPI dentro de la región paralela, entonces hay que realizar la inicialización de MPI mediante `MPI_Init_thread`.

2.3. Versión MPI con particionado bidimensional y topología cartesiana

El objetivo es particionar el dominio tanto en vertical como en horizontal, por lo que cada proceso podrá tener hasta 4 vecinos. Para simplificar la programación, utilizar la utilidad de topologías cartesianas de MPI, con `MPI_Cart_shift` para obtener el identificador de los vecinos.

Nota: en este caso, para la recogida de resultados en el proceso 0 no es apropiado utilizar ninguna primitiva de comunicación colectiva, sino simplemente comunicaciones punto a punto de forma que el proceso receptor determine en qué posición de la matriz global hay que almacenar una submatriz a partir de las coordenadas del proceso que la ha enviado.

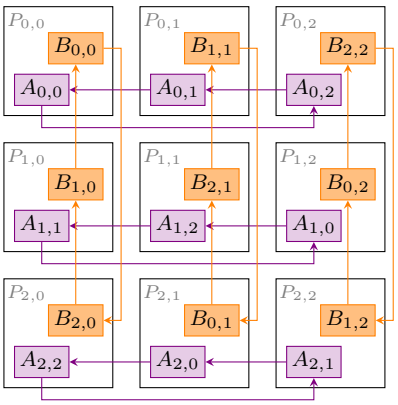

Figura 3: Esquema del algoritmo de Cannon para multiplicar dos matrices.

2.4. Versión CUDA

El objetivo de este proyecto es implementar un versión CUDA partiendo del código secuencial `poisson.c`. Se puede hacer una paralelización progresiva, en la que cada versión mejora la anterior, realizando un estudio comparativo de las prestaciones de cada versión.

1. Una primera versión básica en la que se paraleliza únicamente la función `jacobi_step`, que se implementaría con un *kernel* de CUDA sencillo, que accediera a la memoria global únicamente.
2. La segunda versión mejoraría el *kernel* de la versión anterior, de forma que el acceso a los elementos del vector se acelerara mediante el uso de variables `__shared__`. Pista: para evitar que el *kernel* tenga muchos `if`'s, es recomendable crear hilos también asociados a los elementos del borde de la matriz.
3. Las versiones anteriores tendrían que copiar el vector en cada iteración a la CPU (necesario para el cálculo del criterio de parada). La tercera versión debería realizar el cálculo completo de `jacobi_poisson` en la GPU, incluyendo el criterio de parada, sin tener que copiar el vector (únicamente al final). Pista: el cálculo del criterio de parada requiere de una reducción, que en el caso de hilos CUDA puede implementarse mediante dos *kernels*, uno que haga la reducción a nivel de bloque de hilos, y otro que posteriormente haga la suma de las sumas parciales.

3. Producto matriz-matriz

El producto de dos matrices, $C = AB$, es una operación importante, especialmente en algoritmos de álgebra lineal con matrices densas. En nuestro caso, suponemos que las matrices son cuadradas y de dimensión $N \times N$. El producto matricial tiene en este caso un coste $\mathcal{O}(N^3)$, por lo que es interesante su paralelización para el caso de matrices grandes. Sin embargo, la paralelización de dicha operación es difícil debido a las dependencias de datos. Diferentes autores han propuesto algoritmos paralelos más o menos complejos.

3.1. Algoritmo de Cannon

La Figura 3 ilustra el funcionamiento del algoritmo de Cannon para multiplicar dos matrices. Las matrices se reparten según una distribución 2-D en una malla bidimensional de procesos MPI. En cada fase del algoritmo se realiza un desplazamiento en anillo de A a lo largo de cada fila de procesos, y de B a lo largo de cada columna de procesos, seguido del cálculo local. Por tanto, es necesario crear comunicadores nuevos para las filas y columnas de procesos. La forma más fácil de hacer esto es con la funcionalidad ofrecida por la topología cartesiana de MPI.

Para este proyecto se permite utilizar alguna implementación de método que esté disponible en Internet, por ejemplo `https://github.com/anicolaspp/Parallel-Computing-MPI-Matrix-Multiplication`. Se debe analizar en detalle el código, y explicar su funcionamiento. En el caso de las primitivas de MPI que no se hayan mencionado en clase, se deberá incluir una descripción didáctica durante la presentación.