DESARROLLO DE SOFTWARE EN ARQUITECTURAS PARALELAS

PRÁCTICA 5: MMmalla

La práctica se ENTREGARÁ en la ficha `Evaluación` de UACloud:

Nombre de la entrega de prácticas en UACloud: MMmalla

Solo hay entregar los ficheros explícitamente indicados en la práctica, junto con aquellos que sean imprescindibles para su compilación y/o ejecución.

En esta práctica se desea que se realice la implementación paralela, sobre MPI, del producto matriz por matriz, utilizando el algoritmo desarrollado para una arquitectura de malla (algoritmo de Cannon). Las matrices `A` y `B` deben estar particionadas en `r` $\times$ `r` bloques (`r` variable y no superior a `rmax=4`). Todos los bloques deben ser del mismo tamaño dado por la variable `bloqtam` que no debe ser superior a `maxbloqtam=100`. Todos los bloques de matrices que son necesarios definir: `A,B,C,ATMP` se considerarán almacenados en un vector. Por ejemplo, si `A`$= [a_{ij}]_{0 \leq i,j \leq m-1}$, los elementos de esta matriz estarán almacenados en el vector:

$$a = (a_{0,0}, a_{0,1}, \ldots, a_{0,m-1}, a_{1,0}, a_{1,1}, \ldots, a_{1,m-1}, \ldots, a_{m-1,0}, a_{m-1,1}, \ldots, a_{m-1,m-1}).$$

Esto permite trabajar más fácilmente con las comunicaciones.

Podemos establecer el producto `c = a·b`, cuando las matrices implicadas están almacenadas en forma de vector y de orden `m`. La siguiente función obtiene este producto y adicionalmente lo acumula en `c` (`c = c + a·b`, operación que se necesita en el algoritmo):

```
void mult(double a[], double b[], double *c, int m) {
int i,j,k;
for (i=0; i<m; i++)
    for (j=0; j<m; j++)
       for (k=0; k<m; k++)
          c[i*m+j]=c[i*m+j]+a[i*m+k]*b[k*m+j];
return; }
```

Esta función puede ser utilizada para obtener los distintos productos por bloques que calculan los procesos.

Vamos a dar ahora unas nociones sobre cómo programar este producto matriz por matriz sobre MPI sin utilizar topologías de procesos. Con MPI no hay restricciones sobre qué tareas pueden comunicarse con otras. Sin embargo, para este algoritmo deseamos ver los procesos como situados en una malla abierta. Utilizaremos para ello la numeración que cada proceso obtiene cuando llama a `MPI_Comm_rank(MPI_COMM_WORLD, &myrank)`. El proceso 0 (proceso `root`) se encargará de obtener por el input estándar el único dato necesario: el orden de cada bloque, `bloqtam`. Además, comprobará que el número de procesos, `nproc`, sea un cuadrado perfecto de manera que se ajuste a la estructura de una malla cuadrada. Posteriormente enviará el parámetro necesario a los otros procesos, `bloqtam`. Las tareas con número de orden distinto de

cero recibirán dicho parámetro. Notar que ésta es la única diferencia que hay entre lo que debe hacer el proceso 0 y el resto.

Ahora, cada tarea necesita conocer la `fila` y la `columna` en la que se encuentra localizada en la malla. Esto es sencillo, ya que la división entera `myrank/r` nos da la `fila` de la tarea y el resto de esa división entera nos da la `columna` (`fila, columna` = $0, 1, 2, \ldots, r-1$). Ahora, usando las variables `myrank`, `nproc` y `r` podemos conocer sin demasiada dificultad los números de orden de los procesos situadas `arriba` y `abajo` en la misma columna. Estos números de orden serán utilizados para la posterior rotación de los bloques de `B` entre las columnas.

En cada etapa del algoritmo se envía un bloque de matriz `A` a todas las tareas de una misma fila. Así pues, es conveniente almacenar en un array, que llamaremos `mifila[]`, los números de orden de las tareas situadas en la fila de la tarea en cuestión.

Como este algoritmo supone que cada bloque está almacenado ya en cada tarea, cada proceso definirá su bloque `a` como:

`a[i]=(i+1)*(float)(fila*columna+1)`^{2}`/bloqtam`^{2}`, i=0,1,2,...,bloqtam*bloqtam-1`,

y los bloques de `b` de manera que `B` sea la matriz identidad. Esto permitirá chequear al final del algoritmo si `C=A`.

Finalmente, como ya conocemos todas las variables necesarias se procede con el bucle principal del algoritmo.

Se aconseja que se usen distintas etiquetas de mensaje en cada iteración del algoritmo y que se especifique el número de orden en `MPI_Recv()`, es decir, que no se use como número de orden el valor `MPI_ANY_SOURCE`.

Cuando los cálculos estén terminados se comprobará que `A=C` para verificar que la multiplicación se ha efectuado correctamente.

Cada proceso debe contar el número de errores que ha cometido (si todo funciona correctamente serán cero errores) y enviarle la información al proceso `root`, el cual escribirá por la salida estándar el número de errores para cada proceso identificado por su número de orden. Para evitar posibles errores de redondeo, se recomienda verificar los errores con un nivel de precisión:

```
numerror=0;
for (int i=0; i<bloqtam*bloqtam; i++)
    if (fabs(a[i]-c[i]) > 0.0000001) numerror++;
printf("Proceso: %d. Numero de errores: %d\n",myrank,numerror);
```

Ficheros a entregar:

matriz.c Contendrá la unidad principal y la función `mult` que calcula el producto de dos matrices almacenadas como vectores.

makefile Makefile utilizado para la compilación.