SOAD-Laboratorio 4

Preparación práctica

Análisis del paralelismo de omp_mm.c y propuesta de mejora

El código proporcionado implementa la multiplicación de matrices mediante el uso de OpenMP. A continuación, haré un comentario sobre el grado de paralelismo e intentaré proponer una mejora para éste.

El código actual distribuye los cálculos de la multiplicación de matrices en varios hilos de ejecución. El programa utiliza una región paralela que comparte tres matrices: a, b y c, así como variables que almacenan el número de hilos que se utilizan y el tamaño del bloque de iteraciones a realizar por cada hilo. Las iteraciones de los bucles se distribuyen con la directiva "#pragma omp for".

Considero que el grado de paralelismo podría mejorarse tanto reduciendo el tamaño de chunk, para que cada hilo realizara menos iteraciones de los bucles externos y pudiesen haber más trabajando en paralelo, como paralelizando los bucles más internos en vez de los externos. Yo me voy a decantar por realizar la primera opción, ya que la matriz tampoco es tan grande y reduciendo el chunk ya tendremos la posibilidad de tener una cantidad considerable de threads trabajando y nos será más cómodo que los threads se encarguen de líneas completas a la hora del cálculo de la multiplicación de matrices y no de columnas concretas. Además, si usáramos la segunda opción para el cálculo, tendríamos problemas de sincronización dado que varios threads querrían escribir en la misma posición y tendríamos que solucionar problemas de carrera. Lo que voy a añadir, además, es la directiva #pragma omp barrier al final de las inicializaciones para asegurarme de que al iniciar el cálculo de la multiplicación de matrices las inicializaciones ya estén todas hechas.

```c
int main (int argc, char *argv[])
{
int     tid, nthreads, i, j, k, chunk;
double  a[NRA][NCA],           /* matrix A to be multiplied */
        b[NCA][NCB],           /* matrix B to be multiplied */
        c[NRA][NCB];           /* result matrix C */

chunk = 5;                    /* set loop iteration chunk size */

/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");
    }
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;

  #pragma omp barrier
  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for(j=0; j<NCB; j++)       {
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  }   /*** End of parallel region ***/
```

Fig.1: omp_mm_mejorado.c

Versión de los programas con redirección de la salida a un fichero de texto

Antes de empezar a implementar las diferentes versiones de la práctica, he probado de añadir el código necesario para escribir el resultado de la multiplicación de matrices tanto en la versión serial como en la versión paralela que he propuesto.

```
fp = fopen("salidaSERIAL", "w");
fprintf(fp, "Result Matrix:\n");
for (i=0; i<NRA; i++) {
  for(j=0; j<NCB; j++) fprintf(fp,"%6.2f ",c[i][j]);
  fprintf(fp,"\n");
}
```

```
fp = fopen("salidaOMP", "w");
fprintf(fp, "Result Matrix:\n");
for (i=0; i<NRA; i++) {
  for(j=0; j<NCB; j++) fprintf(fp,"%6.2f ",c[i][j]);
  fprintf(fp,"\n");
}
```

Lo que hacemos es declarar una variable fp de tipo FILE que viene a ser un puntero a archivo para al final del código, cuando ya tenemos el resultado de la matriz, abrir el archivo de texto donde queremos escribir el resultado en modo escritura y a continuación usar fprintf especificando como primer argumento fp para poder ir escribiendo los elementos de la matriz resultado c en el archivo de texto.

Guardar números en un fichero de texto en ASCII vs guardarlos en formato int

Tanto el guardar números en un fichero de texto en ASCII como guardarlos en formato int tiene ventajas y desventajas. Si nos decantamos por la primera opción, nos tendremos que tomar la molestia de convertir los enteros a ASCII antes de escribirlos en el fichero de texto, y en cambio luego si queremos leerlos y escribirlos por pantalla ya los tendremos en el formato correcto. Por otro lado, si escogemos la segunda opción, al guardar no tendremos que hacer ninguna conversión pero en cambio al leerlos y posteriormente mostrarlos por pantalla sí que tendremos que convertir los enteros a ASCII. Yo me voy a decantar por la primera opción dado que me gusta que los datos sean leíbles por el humano también en el fichero de texto, algo que no es posible si los guardamos en formato int. Gracias a la función fprintf mencionada anteriormente, tenemos esta necesidad cubierta.

Práctica

Versión 1

Es la versión implementada en la segunda sección de la preparación de la práctica:

```
int main (int argc, char *argv[])
{
int     tid, nthreads, i, j, k, chunk;
double  a[NRA][NCA],           /* matrix A to be multiplied */
        b[NCA][NCB],           /* matrix B to be multiplied */
        c[NRA][NCB];           /* result matrix C */
FILE *fp;
chunk = 5;                    /* set loop iteration chunk size */

/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");
    }
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;
  #pragma omp barrier

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for(j=0; j<NCB; j++)       {
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  }   /*** End of parallel region ***/

  fp = fopen("salidaOMP", "w");
  fprintf(fp, "Result Matrix:\n");
  for (i=0; i<NRA; i++) {
    for(j=0; j<NCB; j++) fprintf(fp,"%6.2f ",c[i][j]);
    fprintf(fp,"\n");
  }
  printf ("Done.\n");
```

FIg. 2: omp_mm_salidaFichero.c

Como ya he dicho, se declara la variable fp y cuando la multiplicación de matrices ya está calculada, se abre el fichero de texto salidaOMP en modo escritura para ir escribiendo las columnas y filas del resultado con los elementos separados gracias a fprintf que además nos pasa los números a ASCII.

Versión 2

En esta versión lo que hacemos es aprovechar que cada thread trata un conjunto de líneas para ir escribiendo en una cadena de caracteres, simulando ya el fichero, los elementos de la matriz resultado que calcula. Esto de guardar los elementos en una cadena de caracteres nos permite almacenarla en una matriz de cadenas de caracteres llamada resultado donde cada indice de fila de esta matriz corresponde con el mismo indice de fila de la matriz resultado de multiplicar calculada. Esto nos permite llevar a cabo una sincronización, que consiste en permitir al thread que gestionaba la primera línea escribir la suya ya en el fichero, y a los otros hacerlos esperar mientras la fila que va antes que ellos aún no haya sido escrita en el fichero. De este modo, logramos escribir los números ya en el momento de calcularlos y pasarlos al fichero de texto estableciendo una sincronización que nos vemos obligados a asumir.

```
/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk,resultado) private(tid,i,j,k)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");
    }
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;
  #pragma omp barrier

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  fp = fopen("salidaSHARED", "w");
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++) {
    char linia[100] = "";

    for(j=0; j<NCB; j++) {

      for (k=0; k<NCA; k++) c[i][j] += a[i][k] * b[k][j];
      char cadena[20];
      sprintf(cadena, "%6.2f ", c[i][j]);
      strcat(linia, cadena);
    }

    while (i != 0 && resultado[i-1][0] == '\0');
    strcat(resultado[i], linia);
    fprintf(fp, "%s\n", linia);
  }
  fclose(fp);
   /*** End of parallel region ***/
  }

  printf ("Done.\n");
}
```

Fig. 3: omp_mm_SHARED.c

Como podemos ver, cadena hace referencia a cada número calculado, y se van concatenando en “linia”, que cuando acaba el while de sincronización (while(i != 0 && resultado[i-1][0] == ‘\0’)) se escribe en resultado[i] que es la matriz que nos gestiona las líneas escritas hasta el momento y replica el resultado que se verá en el fichero.

Versión 3

Para esta versión no necesitaremos sincronizaciones. Seguiremos el mismo proceso que antes de ir concatenando los números calculados en una cadena de caracteres “linia”, para que cuando cada thread haya acabado de calcular una línea de la matriz resultado, la escriba en un fichero llamado ROW”num_línea”, aprovechando la función sprintf que nos permite concatenar una cadena de caracteres (“ROW”) con un int (“iteración del bucle i”).

```
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++) {
    char linia[100] = "";

    for(j=0; j<NCB; j++) {

      for (k=0; k<NCA; k++) c[i][j] += a[i][k] * b[k][j];
      char cadena[20];
      sprintf(cadena, "%6.2f ", c[i][j]);
      strcat(linia, cadena);
    }

    char archivo[20];
    int num = i;
    sprintf(archivo, "ROW%d", num);
    fp = fopen(archivo, "w");
    fprintf(fp, "%s", linia);
  }
   /*** End of parallel region ***/
  }
  fclose(fp);
  printf ("Done.\n");
}
```

Fig. 4: omp_mm_ROW.c

Como antes, cada thread trata varias líneas, pero al acabar una ya genera un fichero con el resultado.

Versión 4

Para esta versión, lo único que teníamos que hacer era adaptar los bucles de manera que tuviésemos una distribución de elementos por threads que nos fuese también conveniente. Concretamente, hemos intercambiado el bucle j por el i. De esta manera, cada thread trata un conjunto de columnas y es mucho más fácil escribirlas de manera correcta en archivos distintos.

```
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (j=0; j<NCB; j++) {
    char columna[1000] = "";

    for(i=0; i<NRA; i++) {

      for (k=0; k<NCA; k++) c[i][j] += a[i][k] * b[k][j];
      char cadena[20];
      sprintf(cadena, "%6.2f\n", c[i][j]);
      strcat(columna, cadena);
    }

    char archivo[20];
    int num = j;
    sprintf(archivo, "COL%d", num);
    fp = fopen(archivo, "w");
    fprintf(fp, "%s", columna);
  }
   /*** End of parallel region ***/
  }
  fclose(fp);
  printf ("Done.\n");
}
```

Fig. 5: omp_mm_COL.c

En este caso, se sigue el mismo procedimiento con la concatenación de los números, pero como estamos hablando de columnas, tenemos que introducir un salto de línea para cada elemento. Ahora los archivos se llaman COL”num_columna”, donde num_columna es la iteración del bucle j en cuestión.

Vayamos a comparar el tamaño de dos ficheros generados con las dos últimas versiones. Concretamente, compararemos la última fila con la última columna.

Fig. 6: tamany en bytes de ROW61 i COL6

Como podemos ver, COL6 pesa considerablemente más (551 bytes) que ROW61 (60 bytes). Esto es lógico, es debido a que el fichero de texto COL6 contiene 62 saltos de línea (uno por fila) y cada salto de línea cuenta como un caracter, que contribuye al peso del archivo, algo que los archivos ROW no contienen.

Si vamos comparando los ficheros generados en las dos versiones con la versión BASE, podemos comprobar que todos los resultados son correctos, y lo mismo con la segunda versión.

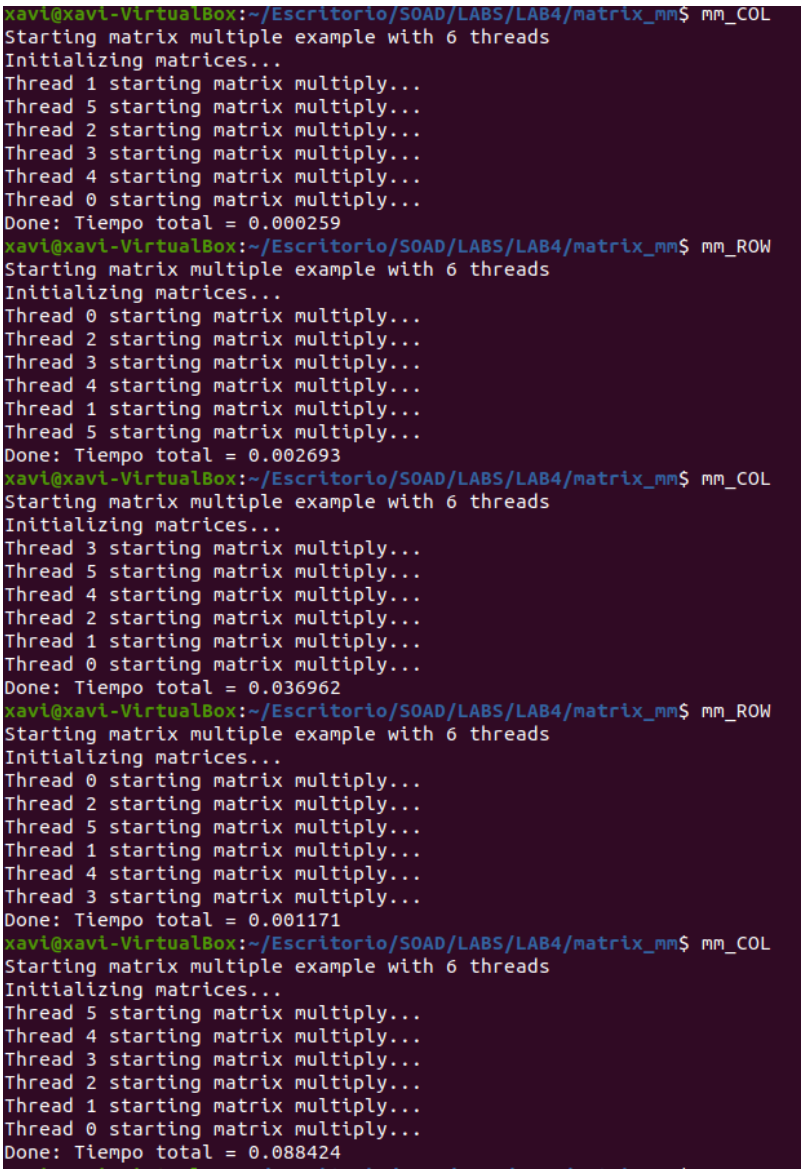

Fig. 7: tiempo de ejecución ROW vs COL

Si realizamos seguidas ejecuciones de las versiones ROW y COL, nos damos cuenta de que no hay una que tarde significativamente más que la otra. Ya lo hemos visto en el código, con ambos hemos alcanzado una técnica de paralelización para el objetivo que teníamos idéntica y buena y estamos hablando de dos versiones que no tenían sincronizaciones y por tanto ausencia de overhead.