Escuela Superior de Informática

Universidad de Castilla-La Mancha
Escuela Superior de Informática

Computadores Avanzados.

4º Grado en Ingeniería Informática.

Práctica NCuerpos.

Autor:
Marcos López Sobrino y
Alberto Salas Seguín.

Fecha:
29 de diciembre de 2018

Índice

1. Instrucciones para la ejecución de los programas.

Para la compilación y ejecución de ambos programas, se ha decidido realizar un *Makefile*, donde para compilar tanto el programa paralelo rápido es suficiente con abrir un terminal e introducir el comando:

```
make all
```

Esta instrucción, como vemos a continuación, primero compila el programa y posteriormente lo ejecuta.

```
all: compile-par-rapido run-par-rapido
```

Donde ***compile-par-rapido*** es:

```
compile-par-rapido:
        mpicc NCuerposParalelo_AlgoritmoRapido.c -o
           NCuerposParalelo_AlgoritmoRapido -lm -Wall
```

Y por su parte, ***run-par-rapido***

```
run-par-rapido:
        mpirun -np 2 NCuerposParalelo_AlgoritmoRapido
```

Siguiendo la transparencia 30 del documento, si solo interesa el tiempo el tiempo de ejecución, se ha incluido la directiva de compilación condicional ***# ifndef NO_SAL*** en el programa, de modo que para únicamente obtener el tiempo de ejecución, en el terminal tenemos que introducir la instrucción ***make compile-par-rapido-nosal***, esta instrucción es la siguiente:

```
compile-par-rapido-nosal:
        mpicc NCuerposParalelo_AlgoritmoRapido.c -o
           NCuerposParalelo_AlgoritmoRapido -lm -Wall -D NO_SAL
```

Posteriormente, ejecutar con ***make run-par-rapido***.

Si en lugar de querer ejecutar el programa paralelo rápido, quisiésemos ejecutar el programa paralelo básico, sería suficiente con introducir el comando ***make compile-par*** y ***make run-par***. Al igual que en el caso del algoritmo paralelo rápido, si solo nos interesa el tiempo de ejecución y no los resultados, lo que tenemos que hacer es compilar el programa mediante ***make compile-par-nosal*** y después ejecutar con ***make run-par***.

2. Explicaciones de diseño.

En primer lugar, se han implementado 3 estructuras:

- **struct Datos**. Dicha estructura contiene las variables **n, tp, k, delta, u** necesarias para el algoritmo.
- **struct Masas**. En esta estructura se encuentra el **id** y la masa (**m**) de cada cuerpo.
- **struct Coord**. Donde almacenamos el **id**, la posición **x** y la posición **y** de cada uno de los cuerpos.

Para poder trabajar con estas estructuras se ha utilizado ***MPI_Datatype***.

```
MPI_Datatype MPI_Datos;
MPI_Datatype MPI_Masas;
MPI_Datatype MPI_Coord;
MPI_Datatype MPI_CNCR;
```

MPI_Datatype MPI_CNCR se ha utilizado para la implementación del algoritmo con distribución cíclica como se dice en el documento a partir de la diapositiva 44. De este modo, el proceso 0 va a almacenar las posiciones de todos los cuerpos y se las va a enviar a todos los procesos mediante la primitiva ***MPI_Scatter***.

Las operaciones que se han utilizado han sido las mencionadas en el enunciado y explicadas en el documento *Introducción a MPI*.

- Para el envío y recepción de las masas se ha utilizado la operación ***MPI_Bcast***.
- Para el envío y recepción de las posiciones y velocidades desde el rank 0 a los demás, se ha optado por ***MPI_Scatter*** como hemos mencionado anteriormente.
- Para sincronizar a todos los procesos ***MPI_Barrier***.
- Para que el proceso 0, pueda imprimir los valores de los cuerpos y que se produzca una salida ordenada, primero debe recibir todos los datos, que son las velocidades y aceleraciones de los cuerpos asignados a otros procesos. Para este fin, se ha usado ***MPI_Gather***.

3. Solución a la dificultad de la pagina 43.

```
1  for(int fase = 1; fase < npr; fase++){
2
3    \* ---- Enviamos a destino/Recibimos desde fuente: p_anillo y
        a_anillo ----*\
4
5     MPI_Sendrecv_replace(p_anillo, ncu, MPI_Coord, dst, 1, src, 1,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
6     MPI_Sendrecv_replace(a_anillo, ncu, MPI_Coord, dst, 1, src, 1,
         MPI_COMM_WORLD, MPI_STATUS_IGNORE);
7
8    \* ---- Calculamos aceleraciones ---- *\
9
10    calcularAceleracion(fase);
11
12 }
13
14    \* ---- Enviamos a_anillo a destino y recibimos a_anillo de
         fuente ----*\
15
16 MPI_Sendrecv_replace(a_anillo, ncu, MPI_Coord, dst, 1, src, 1,
      MPI_COMM_WORLD, MPI_STATUS_IGNORE);
17
18    \* ---- Sumamos aceleraciones ----*\
19
20 for(int i = 0; i < ncu; i++){
21         a_local[i].x += a_anillo[i].x;
22         a_local[i].y += a_anillo[i].y;
23 }
```

En el bucle de la línea 1 del código anterior, lo que se hace es que por cada fase, enviamos y recibimos *p_anillo* y *a_anillo*, es decir, cada proceso en su variable p_anillo y a_anillo está guardando las variables p_anillo y a_anillo de todos los demás, de este modo, todos los procesos tienen todos los datos.

Cuando termina el bucle, como hay que sumar todas las aceleraciones, en la última iteración para tener todas esas iteraciones, en la línea 16, se ejecuta la primitiva ***MPI_Sendrecv_replace*** de a_anillo, de modo que todos los procesos tienen las aceleraciones de todos los procesos.

Posteriormente, en las aceleraciones locales del proceso, se suma su propia aceleración local (a_local[i]) más la aceleración del anillo (a_anillo[i]) en las posiciones x e y.