# El problema de los n cuerpos y su paralelización con MPI

Universidad de Castilla-La Mancha
E. S. Informática de Ciudad Real

# Índice

- Problema de los n cuerpos
- Resolución
- Datos de entrada y salida
- Algoritmo secuencial
- Valoración del programa secuencial
- Mejoras opcionales

# Índice (cont.)

- Paralelización del algoritmo secuencial
  - Paralelización del algoritmo básico
    - Particionado
    - Comunicaciones
    - Agregación de tareas
    - Asignación de tareas a los procesadores
    - Implementación con MPI
    - Tiempo de ejecución
  - Paralelización del algoritmo rápido
    - Algoritmo paralelo del cálculo de las aceleraciones
    - Implementación del algoritmo con distribución cíclica
- Trabajo y su valoración

# Problema de los n cuerpos

- **Simular el movimiento de n cuerpos** debido a las fuerzas gravitatorias entre ellos
- Simplificaciones:
  - Mecánica clásica (gravitación newtoniana)
  - Los cuerpos son puntos de masa
  - Nos limitamos a un espacio de dos dimensiones
  - Los cuerpos no chocan entre sí

# Mecánica

- La fuerza gravitatoria que ejerce un cuerpo p sobre un cuerpo q en un instante t es:

$$\boldsymbol{F}_{pq}(t) = G\frac{m_p\, m_q}{\|\boldsymbol{s}_p(t)-\boldsymbol{s}_q(t)\|^3}\,[\boldsymbol{s}_p(t) - \boldsymbol{s}_q(t)]$$

- $\mathbf{s}_p(t)$ y $\mathbf{s}_q(t)$ son las posiciones de los cuerpos p y q, respectivamente, en el instante t
- $m_p$ y $m_q$ son las masas de los cuerpos p y q
- G es la constante de gravitación universal
  - Supondremos las unidades de masa y distancia de tal forma que **G=1**

# Mecánica

- Segunda ley del movimiento de Newton:

  Fuerza = masa × aceleración

- Por tanto, la aceleración que experimenta un cuerpo q debida a la atracción gravitatoria de p es:

$$\boldsymbol{a}_{pq}(t) = \frac{\boldsymbol{F}_{pq}(t)}{m_q} = G\frac{m_p}{\|\boldsymbol{s}_p(t)-\boldsymbol{s}_q(t)\|^3}[\boldsymbol{s}_p(t)-\boldsymbol{s}_q(t)]$$

- Y la que experimenta debido a la atracción gravitatoria de todos los n-1 cuerpos restantes es:

$$\boldsymbol{a}_q(t) = \sum_{\substack{0\le p<n \\ p\neq q}} \boldsymbol{a}_{pq}(t) = \sum_{\substack{0\le p<n \\ p\neq q}} G\frac{m_p}{\| \boldsymbol{s}_p(t)-\boldsymbol{s}_q(t) \|^3}[\boldsymbol{s}_p(t)-\boldsymbol{s}_q(t)]$$

# Sistema de ecuaciones diferenciales

- Velocidad (derivada de la posición con respecto al tiempo): $\boldsymbol{v}_q(t) = \frac{d\boldsymbol{s}_q(t)}{d\,t} = \dot{\boldsymbol{s}}_q(t)$
- Aceleración (derivada de la velocidad con respecto al tiempo):

$$\boldsymbol{a}_q(t) = \frac{d}{dt}\frac{d\boldsymbol{s}_q(t)}{d\,t} = \frac{d^2\boldsymbol{s}_q(t)}{d\,t^2} = \ddot{\boldsymbol{s}}_q(t)$$

- Por tanto, para cada q ($0 \leq q < n$) tengo la ecuación diferencial:

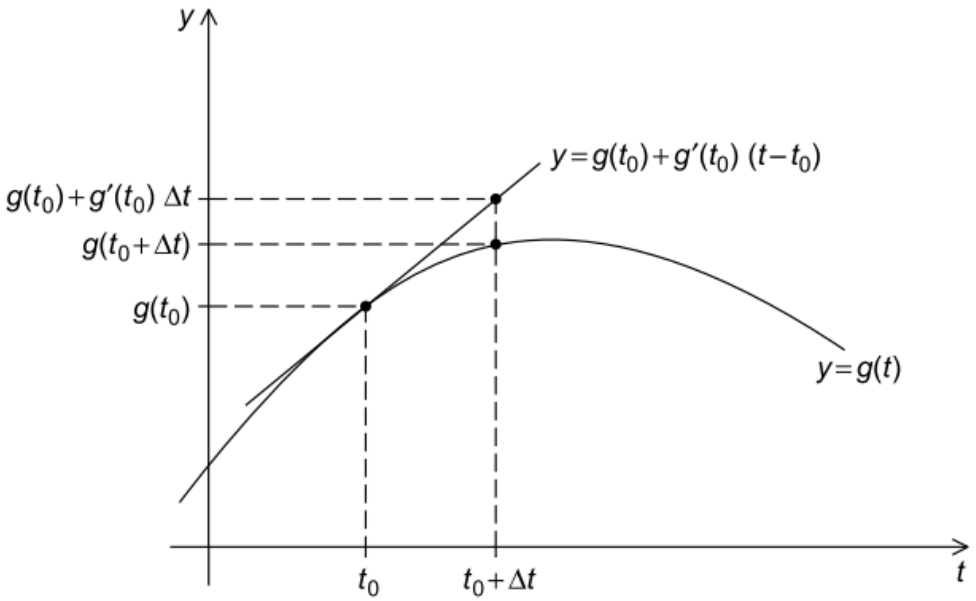
$$\ddot{\boldsymbol{s}}_q(t) = G \sum_{\substack{0 \leq p < n \\ p \neq q}} \frac{m_p}{\| \boldsymbol{s}_p(t) - \boldsymbol{s}_q(t) \|^3} [\boldsymbol{s}_p(t) - \boldsymbol{s}_q(t)]$$

# Resolución

- El problema consiste en resolver el sistema de ecuaciones anterior dadas unas condiciones iniciales
  - Condiciones iniciales: las posiciones y velocidades de cada uno de los $n$ cuerpos
- El método más eficiente conocido hoy para resolver el problema es la **integración numérica**
- Elegiremos el método numérico más simple: el de **Euler**

# Método de Euler

Si en $t_0$ conocemos el valor de una función, $g(t_0)$, y de su derivada, $g'(t_0)$, podemos aproximar el valor de la función en $t_0+\Delta t$ mediante la tangente a la curva en $t_0$:




$$g(t_0+\Delta t) \cong g(t_0) + g'(t_0)\cdot\Delta t$$

# Resolución

- A partir de las posiciones, velocidades y aceleraciones de los cuerpos en un instante t calcularemos las posiciones y velocidades en un instante posterior t+Δt

$$\boldsymbol{s}_q(t+\Delta t) = \boldsymbol{s}_q(t) + \dot{\boldsymbol{s}}_q(t) \cdot \Delta t$$

$$\dot{\boldsymbol{s}}_q(t+\Delta t) = \dot{\boldsymbol{s}}_q(t) + \ddot{\boldsymbol{s}}_q(t) \cdot \Delta t$$

En principio, como vimos, la aceleración es:

$$\ddot{\boldsymbol{s}}_q(t) = G \sum_{\substack{0 \le p < n \\ p \neq q}} \frac{m_p}{\| \boldsymbol{s}_p(t) - \boldsymbol{s}_q(t) \|^3} [\boldsymbol{s}_p(t) - \boldsymbol{s}_q(t)]$$

- Repitiendo el proceso Tp veces podremos calcular las posiciones y velocidades en un instante t+Tp·Δt
- Tp y Δt serán entradas del programa

# Distancia umbral

- Supondremos que cuando dos cuerpos están a una distancia inferior al umbral, U, no se atraen
  - Por tanto:

$$\ddot{\boldsymbol{s}}_q(t) = G \sum_{\substack{0 \le p < n \\ p \neq q \\ \|\boldsymbol{s}_p(t) - \boldsymbol{s}_q(t)\| \ge U}} \frac{m_p}{\| \boldsymbol{s}_p(t) - \boldsymbol{s}_q(t) \|^3} [\boldsymbol{s}_p(t) - \boldsymbol{s}_q(t)]$$

  - Esta simplificación evita aceleraciones muy altas o incluso infinitas
  - El umbral, U, será un dato de entrada

# Datos de entrada y salida

- Entradas:
  - n: número de cuerpos
  - Por cada uno de los cuerpos:
    - Masa
    - Posición inicial (coordenadas x e y)
    - Velocidad inicial (coordenadas x e y)
  - delta: el incremento del tiempo en cada paso
  - Tp: el número total de pasos
  - U: la distancia umbral
- Salidas:
  - Por cada cuerpo: posición, velocidad y aceleración tras cada paso de tiempo
  - Tiempo de ejecución del programa

# Algoritmo secuencial
## Esquema del programa

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "timer.h“
/* Recomendado descargar timer.h en misma carpeta que el programa */
```

*Declaraciones de constantes, tipos, variables y macros*

*Definiciones de funciones usadas*

*Programa principal*

# Esquema del programa principal

```
int main(int argc, char *argv[]) {
  Leer entradas n, delta, Tp, U;
  Reservar espacio para masas, posiciones,
    velocidades y aceleraciones;
  Leer masas, posiciones y velocidades iniciales;
  GET_TIME(inicio); // inicio y fin de tipo double
  t=0; ∀q: calcular e imprimir aceleración inicial, s̈_q(0);
  for (paso= 1; paso<=Tp; paso++) {
        for (q= 0; q<n; q++)
              Calcular e imprimir posición y velocidad
              nuevas, s_q(t + delta) y ṡ_q(t + delta);
        for (q= 0; q<n; q++)
              Calc. e impr. aceleración, s̈_q(t + delta);
        t=t+delta;
  }; GET_TIME(fin); Imprimir tiempo de ejecución;
Liberar memoria dinámica asignada;
}
```

# Mejoras que hay que realizar

- Imprimir los resultados solo cada k pasos de tiempo (k sería una nueva entrada)
  - Y, en cualquier caso, imprimir también los últimos
- Ahorrar cálculos aprovechando que la fuerza ejercida por un cuerpo p sobre q es la misma pero de sentido contrario que la ejercida por q sobre p:

$$\boldsymbol{F}_{pq}(t) = -\boldsymbol{F}_{qp}(t)$$

  - Por tanto:

$$m_q \cdot \boldsymbol{a}_{pq}(t) = -m_p \cdot \boldsymbol{a}_{qp}(t)$$

$$\boldsymbol{a}_{qp}(t) = -\frac{m_q}{m_p} \cdot \boldsymbol{a}_{pq}(t)$$

# Algoritmo para calcular las aceleraciones ahorrando cálculos

*Para cada cuerpo q hacer* $\boldsymbol{a}_q(t) = \boldsymbol{0}$;

```
for (q= 0; q<n; q++) {
        for (p= q+1; p<n; p++) {        // p>q
                Calcular a_pq(t); a_q(t) += a_pq(t);
                a_qp(t) = -(m_q / m_p) · a_pq(t);
                a_p(t) += a_qp(t);
        }
        // a_q(t) está calculado y puede imprimirse
}
```

# Valoración del programa

- Se valora por orden de importancia que:
  - El programa dé resultados correctos siempre
  - Se respeten las directrices de este documento
  - El programa sea fácil de entender:
    - Bien estructurado
    - Código sencillo
    - Nombres adecuados
    - Comentarios relevantes y sencillos
  - El tiempo de ejecución (cuanto menor, mejor)
  - El tamaño de código sea moderado o pequeño

# Mejoras opcionales

- Tomar las masas, posiciones y velocidades iniciales de un archivo
- Trabajar en un espacio finito (bidimensional)
  - Por ejemplo, fijamos $Xmax$ e $Ymax$ de modo que:
    $0 \leq x \leq Xmax, 0 \leq y \leq Ymax$
    - El espacio podría considerarse un rectángulo
      - Cuando un cuerpo llega a un borde "se refleja" en él
    - También puede considerarse toroidal
      - El borde derecho coincide con el izquierdo y el superior con el inferior
      - Cuando un cuerpo rebase el borde derecho aparece por el izquierdo, etc.
- Mostrar los cuerpos gráficamente como puntos que se mueven

# Paralelización del algoritmo secuencial básico

- Recordemos las 4 etapas de la metodología de Foster:
  - Particionado o descomposición en tareas de grano fino
  - Comunicaciones
  - Agregación o aglomeración de tareas
  - Asignación de tareas a los procesadores

# Particionado

- Las tareas de grano fino que surgen de forma natural en este problema son:
  - Por cada paso de tiempo y por cada cuerpo:
    - Cálculo de la posición $\boldsymbol{s}_q(t + delta)$
    - Cálculo de la velocidad $\boldsymbol{v}_q(t + delta)$
    - Cálculo de la aceleración $\boldsymbol{a}_q(t + delta)$
  - Teniendo en cuenta el esquema de algoritmo de la p. 14, el número de tareas resulta ser $3n \cdot Tp + n$:
    - $n \cdot Tp$ cálculos de posición
    - $n \cdot Tp$ cálculos de velocidad
    - $n + n \cdot Tp = n \cdot (Tp + 1)$ cálculos de aceleración

# Comunicaciones

- Teniendo en cuenta las ecuaciones de la página 10:
  - La tarea que calcula $\boldsymbol{s}_q(t + delta)$ necesita:
    - $\boldsymbol{s}_q(t)$ y $\boldsymbol{v}_q(t)$
  - La tarea que calcula $\boldsymbol{v}_q(t + delta)$ necesita:
    - $\boldsymbol{v}_q(t)$ y $\boldsymbol{a}_q(t)$
  - La tarea que calcula $\boldsymbol{a}_q(t + delta)$ necesita:
    - $\boldsymbol{s}_0(t + delta)$, $\boldsymbol{s}_1(t + delta)$, …, $\boldsymbol{s}_{n-1}(t + delta)$, es decir, $\boldsymbol{s}_p(t + delta)$ para los p tales que $0 \leq p < n$

# Agregación

- Hay una secuencialidad temporal, dado que:
  - El calculo de $\boldsymbol{s}_q(t + delta)$ depende de $\boldsymbol{s}_q(t)$
  - El calculo de $\boldsymbol{v}_q(t + delta)$ depende de $\boldsymbol{v}_q(t)$
  - El calculo de $\boldsymbol{a}_q(t + delta)$ depende de $\boldsymbol{s}_q(t + delta)$ el cual depende de $\boldsymbol{v}_q(t)$ que, a su vez, depende de $\boldsymbol{a}_q(t - delta)$
- Por tanto, para cada cuerpo q, agregaremos:
  - Todas las tareas que calculan su posición, en una tarea $\boldsymbol{s}_q$
  - Todas las que calculan su velocidad, en una tarea $\boldsymbol{v}_q$
  - Todas las que calculan su aceleración, en una tarea $\boldsymbol{a}_q$

# Agregación y comunicaciones

- Con la agregación anterior hemos pasado de tener $3\mathrm{nTp}+\mathrm{n}$ tareas a tener $3\mathrm{n}$
- Ahora las comunicaciones entre tareas son:
  - La tarea $\boldsymbol{s}_q$ necesita los valores de $\boldsymbol{v}_q(t)$ $\forall t \in \{0,\Delta, 2\Delta, \ldots, (\mathrm{Tp}-1)\Delta\}$
  - La tarea $\boldsymbol{v}_q$ necesita los valores de $\boldsymbol{a}_q(t)$ $\forall t \in \{0,\Delta, 2\Delta, \ldots, (\mathrm{Tp}-1)\Delta\}$
  - La tarea $\boldsymbol{a}_q$ necesita los valores de $\boldsymbol{s}_p(t)$ $\forall p \,/\, 0 \leq p < n, \forall t \in \{0,\Delta, 2\Delta, \ldots, (\mathrm{Tp}-1)\Delta, \mathrm{Tp}\cdot\Delta\}$
- Hemos ahorrado las comunicaciones de $\boldsymbol{s}_q(t)$ a $\boldsymbol{s}_q(t+delta)$ y de $\boldsymbol{v}_q(t)$ a $\boldsymbol{v}_q(t+delta)$ porque ahora quedan dentro de la misma tarea

# Agregación

- Vemos que muchas de las comunicaciones entre las tareas resultantes de la primera agregación comunican datos del mismo cuerpo q
  - Por tanto, una **segunda agregación** juntará en una las tres tareas relativas al cuerpo q
    - Para todo q ($0 \leq q < n$), $\boldsymbol{s}_q$, $\boldsymbol{v}_q$ y $\boldsymbol{a}_q$ se agregan en una sola tarea: $\boldsymbol{sva}_q$
    - El número de tareas se reduce a **n** (una por cuerpo)

# Agregación

- Tras la segunda agregación:
  - Desaparecen las comunicaciones de los $\boldsymbol{v}_q$, $\boldsymbol{a}_q$ y $\boldsymbol{s}_q$ en los distintos instantes de tiempo
  - La tarea $\boldsymbol{sva}_q$ necesita recibir los valores de $\boldsymbol{s}_p(t)$, para los p tales que $0 \leq p < n$ y $p \neq q$ y los $t = k\Delta$ para $0 \leq k \leq Tp$

# Asignación de tareas a los procesadores

- Llamemos npr al número de procesadores
- Cada tarea $\boldsymbol{sva}_q$ conlleva el mismo trabajo y comunicaciones → repartimos las tareas equitativamente entre los procesadores:
  - Por bloques: sea $r = n \; mód \; npr$
    - $r$ procesadores (p. ej., los $r$ primeros: $0, 1, \ldots, r-1$) realizan $\lceil n/npr \rceil$ tareas cada uno
    - El resto realizan $\lfloor n/npr \rfloor$ tareas cada uno
  - También es posible un reparto cíclico pero presenta más dificultades que el reparto por bloques sin ventajas significativas

# Implementación con MPI del algoritmo secuencial básico

- Elegimos un proceso (el 0) para leer los datos de entrada
- Este proceso debe difundir a los demás (mediante MPI_Bcast):
  - $n$, $delta$, $Tp$, $k$, $U$, masas iniciales y posiciones iniciales
- En cambio, cada proceso no necesita todas las velocidades sino solo las de sus cuerpos
  - Es adecuado usar MPI_Scatter si $n \; mód \; npr = 0$
  - Hacer una primera versión donde $n \; mód \; npr = 0$ y, una vez que funcione, pensar qué hacer si $n \; mód \; npr > 0$

- Para que una tarea pueda actualizar las aceleraciones de sus cuerpos necesita las posiciones de todos los cuerpos
  - Es la única comunicación entre tareas
  - Lo más adecuado es usar **MPI_Allgather**
    - Funciona como MPI_Gather (ver *Introducción a OpenMP y MPI*) pero sin el parámetro target
      - Porque todos los procesos del comunicador son destino
    - Ahorraremos memoria en cada proceso si usamos como primer parámetro MPI_IN_PLACE
    - De esta forma no hay un buffer de emisión y otro de recepción sino que el buffer de recepción deviene en un solo buffer de emisión-recepción

# Salidas

- Cada proceso puede imprimir los resultados de sus cuerpos
  - Pero esto puede dar lugar a una salida desordenada
- Para una salida ordenada:
  - Un proceso (p. ej., el 0) imprimirá todos los resultados
    - Previamente, debe recibir los datos que no tiene (velocidades y aceleraciones de los cuerpos asignados a los otros procesos)
    - Para ello se debe usar MPI_Gather

# Tiempo de ejecución y E/S

- Las capacidades de los sistemas de entrada/salida son muy variables
  - A menudo no están pensadas para procesamiento paralelo
- Para aislar la E/S del tiempo de ejecución del algoritmo:
  - Poner las operaciones de salida bajo la directiva de compilación condicional **# ifndef NO_SAL**
  - Compilar con la opción **–D NO_SAL** cuando solo interese el tiempo de ejecución
    - Sin la opción, si interesan los resultados

# Tiempo de ejecución

- No debemos usar la función clock de C porque mide el tiempo de CPU (no incluye tiempos muertos como los que puede haber esperando un mensaje)
- En MPI disponemos de la función **MPI_Wtime()** que calcula el número de segundos transcurridos desde un instante pasado (ya no necesitamos timer.h)
  - El tiempo transcurrido entre dos puntos del código se puede obtener con el siguiente esquema:

```
double inicio, fin;
inicio = MPI_Wtime();
Código cuyo tiempo queremos medir
fin = MPI_Wtime(); // Tiempo transcurrido: fin – inicio
```

# Tiempo de ejecución

- Un esquema más general, se basa en calcular el tiempo de ejecución del proceso más lento

  Barrier();

  inicio_local = MPI_Wtime();

  *Código cuyo tiempo queremos medir*

  fin_local= MPI_Wtime();

  t_local= fin_local – inicio_local;

  t_global= *Máximo de los t_local*;

  - La barrera sincroniza aproximadamente el inicio de todos los procesos

# Paralelización del algoritmo rápido

- Llamaremos **algoritmo rápido** al que ahorra cálculos aprovechando que la fuerza ejercida por un cuerpo p sobre q es la misma pero de sentido contrario que la ejercida por q sobre p (pp. 15-16)
- Supongamos que mantenemos la agregación de una tarea por cuerpo
  - En el algoritmo secuencial de la p. 16
    - Cada iteración por el primer for correspondería a una tarea
    - Una paralelización "directa" de la tarea asociada al cuerpo q, se muestra en la p. siguiente
      - En azul, los cambios sobre la versión secuencial

$\boldsymbol{a}_q(t) = \boldsymbol{0}$;

$\forall p < q$: *enviar* $\boldsymbol{s}_q(t)$ *al proceso* $p$;

$\forall p > q$: *recibir* $\boldsymbol{s}_p(t)$ *del proceso* $p$;

*// necesario para calcular* $\boldsymbol{a}_{pq}(t)$

```
for (p= q+1; p<n; p++) {      // p>q
```

*Calcular* $\boldsymbol{a}_{pq}(t)$; $\boldsymbol{a}_q(t) \mathrel{+}= \boldsymbol{a}_{pq}(t)$;

$$\boldsymbol{a}_{qp}(t) = -\frac{m_q}{m_p} \cdot \boldsymbol{a}_{pq}(t);$$

*Enviar* $\boldsymbol{a}_{qp}(t)$ *al proceso* $p$;

*// la suma* $\boldsymbol{a}_p(t) \mathrel{+}= \boldsymbol{a}_{qp}(t)$ *se hará en el*

*// proceso* $p$ *que es el que calcula* $\boldsymbol{a}_p(t)$

```
}
for (p= 0; p<q; p++)
```

{*Recibir* $\boldsymbol{a}_{pq}(t)$*del proceso* $p$ ; $\boldsymbol{a}_q(t) \mathrel{+}= \boldsymbol{a}_{pq}(t)$;}

- Con una posible mejora de la sincronización tendríamos:

$\boldsymbol{a_q(t) = 0;}$

*$\forall p < q$: enviar $\boldsymbol{s_q(t)}$ al proceso $p$;*

*$\forall p > q$: recibir $\boldsymbol{s_p(t)}$ del proceso $p$;*

**for** (p= q+1; p<n; p++) { // $p>q$

*Calcular* $\boldsymbol{a_{pq}(t)}$; $\boldsymbol{a_q(t)}$ += $\boldsymbol{a_{pq}(t)}$; $\boldsymbol{a_{qp}(t) = -\frac{m_q}{m_p} \cdot a_{pq}(t)}$;

*Enviar* $\boldsymbol{a_{qp}(t)}$ *al proceso* $p$;

$r = 2q - p$; // $\boldsymbol{q - r = p - q}, r < q$

**if** $(r \geq 0)$ *Recibir* $\boldsymbol{a_{rq}(t)}$ *del proceso* $r$; $\boldsymbol{a_q(t)}$ += $\boldsymbol{a_{rq}(t)}$;

} // Recibidos los $\boldsymbol{a_{rq}(t)}$ tales que $1 \leq q - r = p - q < n - q$

// Hay que recibir los $\boldsymbol{a_{rq}(t)}$ tales que $0 \leq r < q$,
// es decir, tales que $1 \leq q - r \leq q$

// Faltan por recibir los $\boldsymbol{a_{rq}(t)}$ tales que $n - q \leq q - r \leq q$

**for** (r= $2q - n$; $r \geq 0$; r − −) // ordenados por el valor de $q - r$

{*Recibir* $\boldsymbol{a_{rq}(t)}$*del proceso* $r$; $\boldsymbol{a_q(t)}$ += $\boldsymbol{a_{rq}(t)}$;}

# Paralelización del algoritmo rápido

- Pero el pseudocódigo de las páginas anteriores es difícil de implementar:
  - Implementación lenta y propensa a fallos
- Hay una implementación más sencilla y eficiente
  - Los npr procesos se comunican como si formasen un anillo, es decir,
    - Cada proceso **envía datos al proceso anterior y los recibe del siguiente**
      - Se entiende que el proceso cero es el que sigue al último
    - Este envío y recepción de datos sucede npr-1 veces

# Algoritmo paralelo del cálculo de las aceleraciones aprovechando simetrías

- Cada proceso tendrá:
  - Un vector p_local con las posiciones de sus cuerpos
  - Un vector a_local con las aceleraciones de sus cuerpos que queremos calcular
  - Un vector p_anillo con las posiciones de otros cuerpos que van circulando por los procesos
  - Un vector a_anillo con las aceleraciones de otros cuerpos que se están calculando y van circulando por los procesos
- Un esquema del algoritmo empieza en la p. sgte.

# Esquema del algoritmo

fte = (idpr+1) mód npr; //idpr: identificador proceso
dest = (idpr−1+npr) mód npr;
*p_anillo* ← *p_local*; *a_local* ← *a_anillo* ← *0*;
*Actualizar* a_local y a_anillo *sumando las aceleraciones debidas a fuerzas entre cuerpos locales (si* $p > q$*, la aceleración* $\boldsymbol{a}_{pq}$ *se suma a* a_local *y* $\boldsymbol{a}_{qp}$ *se suma a* a_anillo*)*;
// Recordemos que $\boldsymbol{a}_{qp}(t) = -(m_q/m_p) \cdot \boldsymbol{a}_{pq}(t)$
**for** (fase=1; fase<npr; fase++) {
    *Enviar* p_anillo *y* a_anillo *a* dest;
    *Recibir* p_anillo *y* a_anillo *de* fte;
    // He recibido las posiciones y aceleraciones de los
    // cuerpos propios del proceso (idpr+fase) mód npr

# Esquema del algoritmo (cont.)

*Actualizar* a_local y a_anillo *sumando las aceleraciones debidas a fuerzas entre cada par de cuerpos* (p, q) *tales que* p *es propio del proceso* (idpr+fase) mód npr, q *es local y* $p > q$;

} // fin del for

// En a_local ya se han sumado todas las aceleraciones
// debidas a fuerzas entre cada par de cuerpos (p, q)
// tal que q *es* local, y $p > q$;
*Enviar* a_anillo *a* dest; *Recibir* a_anillo *de* fte;
// a_anillo ha dado la vuelta completa
// a_anillo == suma aceleraciones debidas a fuerzas
// entre cada par (p, q), con q local y $p < q$
a_local = a_local + a_anillo;

# Ejemplo de aplicación del algoritmo

- 3 procesos y 6 cuerpos
- Distribución cíclica

| | Variable | Proceso 0 | | Proceso 1 | | Proceso 2 | |
|---|---|---|---|---|---|---|---|
| Inicialización | p_local | $s_0$ | $s_3$ | $s_1$ | $s_4$ | $s_2$ | $s_5$ |
| | p_anillo | $s_0$ | $s_3$ | $s_1$ | $s_4$ | $s_2$ | $s_5$ |
| | a_local | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |
| | a_anillo | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ |
| Fase 0 | a_local | $a_{30}$ | $\mathbf{0}$ | $a_{41}$ | $\mathbf{0}$ | $a_{52}$ | $\mathbf{0}$ |
| | a_anillo | $\mathbf{0}$ | $a_{03}$ | $\mathbf{0}$ | $a_{14}$ | $\mathbf{0}$ | $a_{25}$ |

| | Vble. | Proceso 0 | | Proceso 1 | | Proceso 2 | |
|---|---|---|---|---|---|---|---|
| | p_local | $s_0$ | $s_3$ | $s_1$ | $s_4$ | $s_2$ | $s_5$ |
| Fase 1 | p_anillo | $s_1$ | $s_4$ | $s_2$ | $s_5$ | $s_0$ | $s_3$ |
| | a_anillo | $\mathbf{0}$ | $\boldsymbol{a_{14}}$ | $\mathbf{0}$ | $a_{25}$ | $\mathbf{0}$ | $a_{03}$ |
| | a_local | $a_{30} + a_{10} + a_{40}$ | $a_{43}$ | $a_{41} + a_{21} + a_{51}$ | $a_{54}$ | $a_{52} + a_{32}$ | $\mathbf{0}$ |
| | a_anillo | $a_{01}$ | $a_{14} + a_{04} + a_{34}$ | $a_{12}$ | $a_{25} + a_{15} + a_{45}$ | $\mathbf{0}$ | $a_{03} + a_{23}$ |
| Fase 2 | p_anillo | $s_2$ | $s_5$ | $s_0$ | $s_3$ | $s_1$ | $s_4$ |
| | a_anillo | $a_{12}$ | $a_{25} + a_{15} + a_{45}$ | $\mathbf{0}$ | $a_{03} + a_{23}$ | $a_{01}$ | $a_{14} + a_{04} + a_{34}$ |
| | a_local | $a_{30} + a_{10} + a_{40} + a_{20} + a_{50}$ | $a_{43} + a_{53}$ | $a_{41} + a_{21} + a_{51} + a_{31}$ | $a_{54}$ | $a_{52} + a_{32} + a_{42}$ | $\mathbf{0}$ |
| | a_anillo | $a_{12} + a_{02}$ | $a_{25} + a_{15} + a_{45} + a_{05} + a_{35}$ | $\mathbf{0}$ | $a_{03} + a_{23} + a_{13}$ | $a_{01}$ | $a_{14} + a_{04} + a_{34} + a_{24}$ |

| De p. anterior | Vble. | Proceso 0 | | Proceso 1 | | Proceso 2 | |
|---|---|---|---|---|---|---|---|
| | p_local | $s_0$ | $s_3$ | $s_1$ | $s_4$ | $s_2$ | $s_5$ |
| | a_local | $\mathbf{a_{30} + a_{10} + a_{40} + a_{20} + a_{50}}$ | $\mathbf{a_{43} + a_{53}}$ | $\mathbf{a_{41} + a_{21} + a_{51} + a_{31}}$ | $\mathbf{a_{54}}$ | $\mathbf{a_{52} + a_{32} + a_{42}}$ | $\mathbf{0}$ |
| Rotar | a_anillo | $\mathbf{0}$ | $\boldsymbol{a_{03} + a_{23} + a_{13}}$ | $\boldsymbol{a_{01}}$ | $\boldsymbol{a_{14} + a_{04} + a_{34} + a_{24}}$ | $\boldsymbol{a_{12} + a_{02}}$ | $\boldsymbol{a_{25} + a_{15} + a_{45} + a_{05} + a_{35}}$ |
| Sumar | a_local | $\mathbf{a_{30} + a_{10} + a_{40} + a_{20} + a_{50}}$ | $\mathbf{a_{43} + a_{53} + a_{03} + a_{23} + a_{13}}$ | $\mathbf{a_{41} + a_{21} + a_{51} + a_{31} + a_{01}}$ | $\mathbf{a_{54} + a_{14} + a_{04} + a_{34} + a_{24}}$ | $\mathbf{a_{52} + a_{32} + a_{42} + a_{12} + a_{02}}$ | $\boldsymbol{a_{25} + a_{15} + a_{45} + a_{05} + a_{35}}$ |

# Implementación del algoritmo

- Téngase en cuenta que:
  - Para enviar y recibir p_anillo *y* a_anillo lo mejor es usar MPI_Sendrecv_replace
  - La principal dificultad es:
    - Implementar las actualizaciones de a_local y a_anillo sumando en el lugar adecuado las aceleraciones $\boldsymbol{a}_{pq}$ *y* $\boldsymbol{a}_{qp}$, con q local y $p > q$
    - La dificultad se debe a que se requiere saber a qué cuerpo corresponde cada dato de los vectores a_local, a_anillo, p_local y p_anillo para saber si $p > q$ y dónde almacenar $\boldsymbol{a}_{pq}$ *y* $\boldsymbol{a}_{qp}$ en cada caso
    - La forma de resolverlo depende del tipo de distribución de cuerpos en procesos que escojamos

# Implementación del algoritmo con distribución cíclica

- Distribución cíclica de cuerpos a procesos:
  - Mejor equilibrio de carga
  - Algunas dificultades:
    - ¿Cómo difunde el proceso 0 a cada proceso las posiciones (y velocidades) de sus cuerpos?
      - El proceso 0 almacena las posiciones de todos los cuerpos ordenadamente → las que envía a un proceso no son consecutivas: cada una está a una distancia npr posiciones de la anterior
      - Podemos usar MPI_Scatter pero tenemos que preparar el tipo de datos del proceso fuente

# Implementación del algoritmo con distribución cíclica

- MPI_Type_vector permite crear un tipo de datos con datos no consecutivos en memoria
- **MPI_Type_vector (count, blocklength, stride, oldtype, &newtype)**
  - Crea el tipo newtype con count bloques
  - Cada bloque tiene blocklength elementos contiguos del tipo oldtype
  - La distancia entre el comienzo de un bloque y el siguiente es de stride elementos de tipo oldtype

# Implementación del algoritmo con distribución cíclica

- Si cada posición (o velocidad) es de un tipo coord, y a cada proceso se asocian miscu cuerpos, podemos crear un tipo, cnc, mediante:

  MPI_Type_vector (miscu, 1, npr, coord, &cnc)

  - cnc consta de miscu bloques de 1 elemento de tipo coord
  - La distancia entre el comienzo de un elemento y el del siguiente es la de npr elementos de tipo coord
  - Si el desplazamiento de un elemento es d, el del siguiente será $d + npr \times ext\_coord$, siendo ext_coord la extensión del tipo de datos coord

- Ej.: para 6 cuerpos y 3 procesos, miscu=2, npr=3 la estructura cnc sería como la de la fig.


- MPI_Scatter difunde elementos consecutivos del tipo elegido
  - Pero para repartir 6 posiciones no queremos algo así

porque las 6 posiciones están contiguas en el proceso fuente

- Queremos algo así

  - Anaranjado: proceso 0; verde: proceso 1; azul: proceso 2
- Es decir, queremos un tipo, cncr, como cnc pero de forma que al tener varios elementos consecutivos del tipo cncr cada uno comience solo una posición después del anterior


- Así la primera figura correspondería a tres datos consecutivos de tipo cncr

- Para conseguir el tipo cncr disponemos de MPI_Type_create_resized
- **MPI_Type_create_resized (oldtype, lb, extent, &newtype)**
  - El nuevo tipo es igual que el antiguo salvo que el límite inferior (*lower bound*) es lb y el superior es lb+extent
  - En nuestro caso necesitamos el límite inferior y la extensión del tipo coord
  - Esto lo obtenemos con MPI_Type_get_extent (coord, &lb, &extent)

# Trabajo y su valoración

- El trabajo constará de tres partes:
  1. El **programa** paralelo para el algoritmo básico
  2. El **programa** paralelo para el algoritmo rápido
  3. Un **documento** en formato Word o pdf con:
     1. Una explicación de la forma en que habéis resuelto la dificultad que se explica en la p. 43 de este documento
     2. Cualquier otra explicación que consideréis esencial para entender vuestro código
- Los programas se valorarán por:
  - Dar resultados correctos
  - Ser fácil de entender
  - El tiempo de ejecución
    - Que esté bien medido y que sea moderado o pequeño
  - Respetar las directrices de este documento
  - Tamaño de código moderado o pequeño