Presentación

Objetivos

Esta práctica tiene como objetivo el diseño e implementación de una aplicación concurrente multi-hilo utilizando phtreads y los mecanismos de concurrencia de java. Para su realización se pondrán en práctica muchos de los conceptos presentados en esta asignatura.

Presentación de la práctica

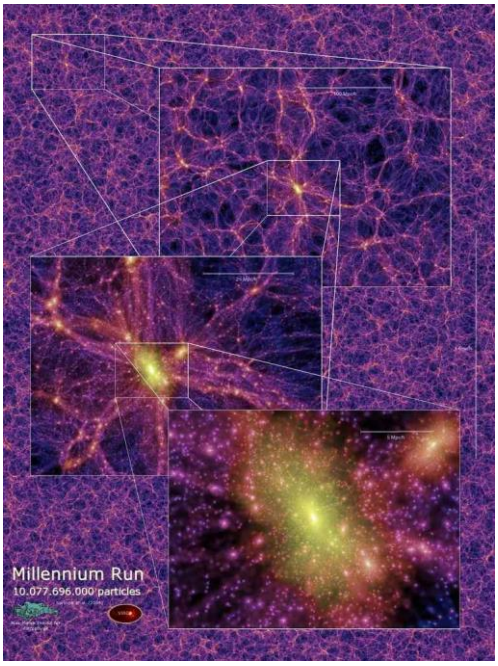
Hay que presentar los archivos con el código fuente realizado. Toda la codificación se hará exclusivamente en lenguaje C, C++ y Java.

Enunciado de la práctica

El objetivo de la práctica consiste en solventar el problema de la simulación NBody mediante una aplicación concurrente multi-hilo.

Simulación de partículas (problema N-Body)

El problema de N-Body es el nombre genérico que se le da a un conjunto de problemas cuya meta es determinar en cualquier instante las posiciones y velocidades de N partículas, de cualquier masa, sometidos a su atracción mutua y partiendo de unas posiciones y velocidades dadas.



Este problema que define los fundamentos de la teoría de los sistemas dinámicos puede ser enunciado como:

- Se define un conjunto de N partículas en el espacio.
- La influencia de las partículas en movimiento sobre otras partículas es modelada mediante la fuerza de atracción (tradicionalmente, la gravedad).
- Si las posiciones y velocidades iniciales son conocidas, el objetivo es calcular las posiciones y velocidades de los cuerpos en el tiempo T.

Una implementación sencilla de este problema implica el cálculo de las velocidades y posiciones de cada partícula, teniendo en cuenta la influencia de otras partículas. Esta implementación implica una complejidad algorítmica del orden de $O(n^2)$ que solo es practicable para un número pequeño de partículas.

El algoritmo básico para la simulación de partículas sería el siguiente:

```
t = 0
while t < t_final
   for i = 1 to n                                   /* n = número de cuerpos */
      Calcular f(i) = fuerza ejercida sobre el cuero i
      Mover cuerpo i en función fuerza f(i) para el intervalo dt
   end for
   Calcular propiedades interesantes de las partículas (colisiones, es opcional)
   t = t + dt                                       /* dt = Incremento Tiempo */
end while
```

La suma de fuerzas que afecta a una partícula se descompone en 3 fuerzas básicas:

Fuerza = fuerza_propia + fuerzas_partículas_cercanas + fuerza_partículas_lejanas

Las fuerzas propias pueden ser calculadas para cada partícula independientemente, de forma embarazosamente paralela. Las fuerzas de las partículas cercanas solo requieren interacción con los vecinos cercanos para calcularse y es relativamente fácil de ejecutar concurrentemente. Los campos de fuerza más lejanos son más costosos de calcular debido a que la fuerza de cada partícula depende del resto de partículas.

La expresión más sencilla para calcular el campo de fuerzas lejanas para la partícula i seria:

```
for i = 1 to n
      f(i) = sumar[ j=1,...,n,  j != i ] f(i,j)
end for
```

donde f(i,j) es la fuerza que incide sobre la partícula i proveniente de la partícula j. El problema de esta alternativa es que la complejidad alcanza $O(n^2)$, mientras que las otras dos fuerzas la complejidad es del orden O(n). Incluso utilizando paralelismo, el cálculo de los campos de fuerza lejanos resulta muy costoso.

La simplificación del centro de masas

Para reducir la complejidad del cálculo de los campos de fuerza lejanos, nos vamos a basar en una simple intuición física. Por ejemplo, si queremos calcular la fuerza gravitacional que afecta a la tierra, deberíamos tener en cuenta las fuerzas de gravedad de todas las estrellas y como contribuyen cada una de ellas a la suma total de fuerzas. Sin embargo, estas partículas no solo consisten en estrellas individuales si no también galaxias formadas por billones de estrellas, por ejemplo, Andrómeda. Estas galaxias si son lo suficientemente compactas y están lo suficientemente lejos, tiene sentido tratarlas como una única partícula localizado en el centro de masas de la galaxia y con una masa igual a la masa total de la galaxia.

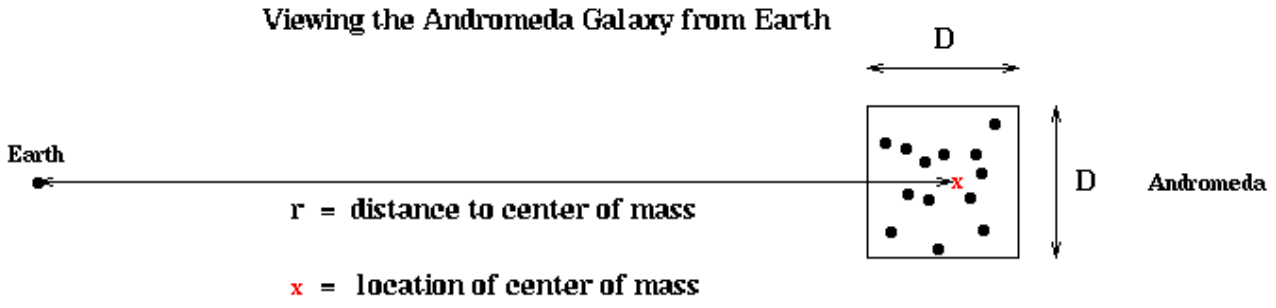


Figura 1. Simplificación de cálculo de campos de fuerza lejanos mediante el centro de masas.

La decisión sobre el conjunto de partículas es lo suficientemente compacto y está lo suficientemente lejos se realizará en base a una ratio entre la dimensión del conjunto de partículas y su distancia:

$$Ratio = \frac{Tamaño\ caja\ que\ contiene\ el\ grupo\ cuerpos}{distancia\ al\ centro\ de\ masas} = \frac{D \times D}{r} \quad (1)$$

si esta ratio es lo suficientemente pequeña, se puede remplazar de forma segura y precisa el grupo de partículas por un único termino (o partícula) ubicado en su centro de masas.

Esta técnica se puede aplicar recursivamente, siempre que se cumpla la condición anteriormente descrita.

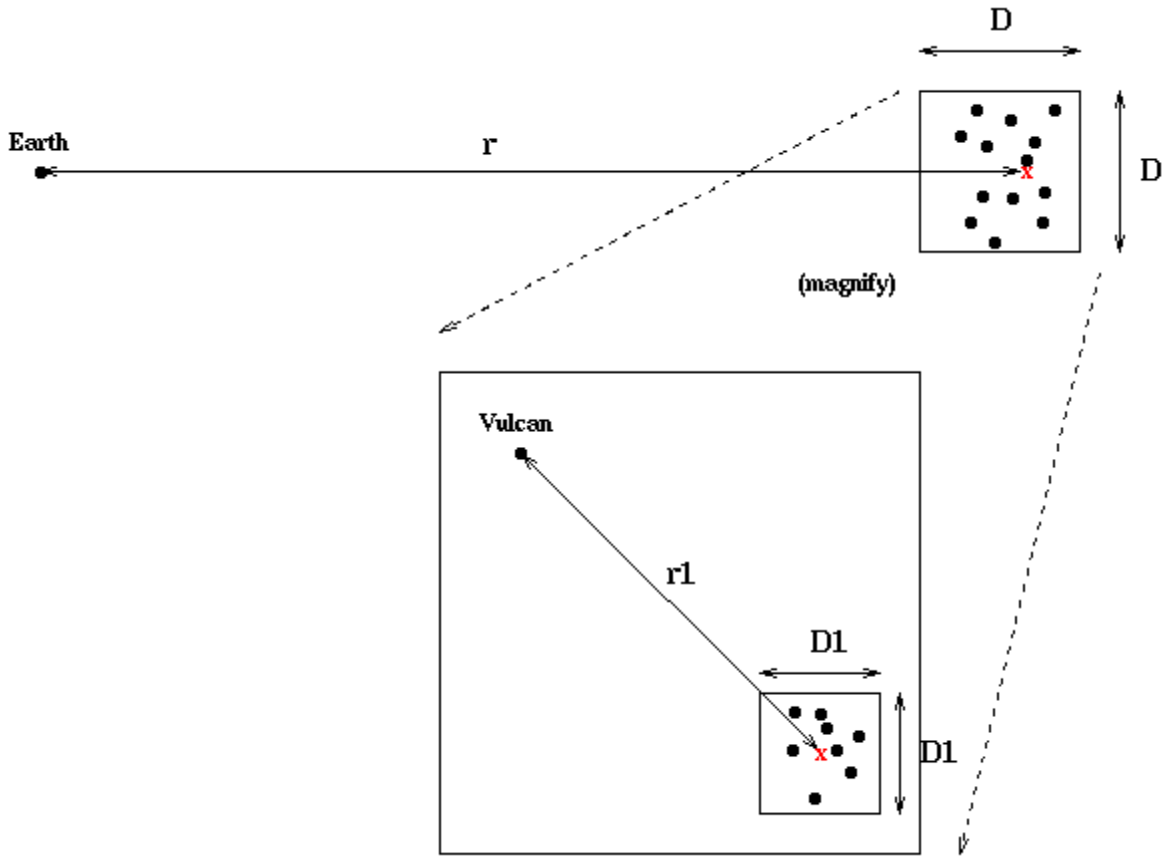


Figura 2. Reemplazo de clusters de partículas por sus respectivos centros de masas.

Representación del espacio mediante árboles cuaternarios

Para implementar de forma recursiva la simplificación basada en el centro de masas, se necesita una estructura de datos para subdividir el espacio de forma fácil. Esta estructura de datos es los Octrees para representar espacios 3D y los Quadtrees (árboles cuaternarios) para espacios con dos dimensiones. Nosotros nos vamos a centra en los árboles cuaternarios.

Para construir el árbol cuaternario empezamos definiendo la raíz del árbol con un cuadrado que incluya todas las partículas. A continuación, se define el siguiente nivel dividiendo la superficie original en cuatro cuadrados más pequeños (líneas azules) con la mitad del perímetro y una cuarta parte del área. Estos cuatros cuadrados son los cuatros hijos de la raíz del árbol. Cada

hijo puede a su vez ser dividido en 4 sub-cuadrados para crear sus hijos, y de esta forma se crea todo el árbol recursivamente.

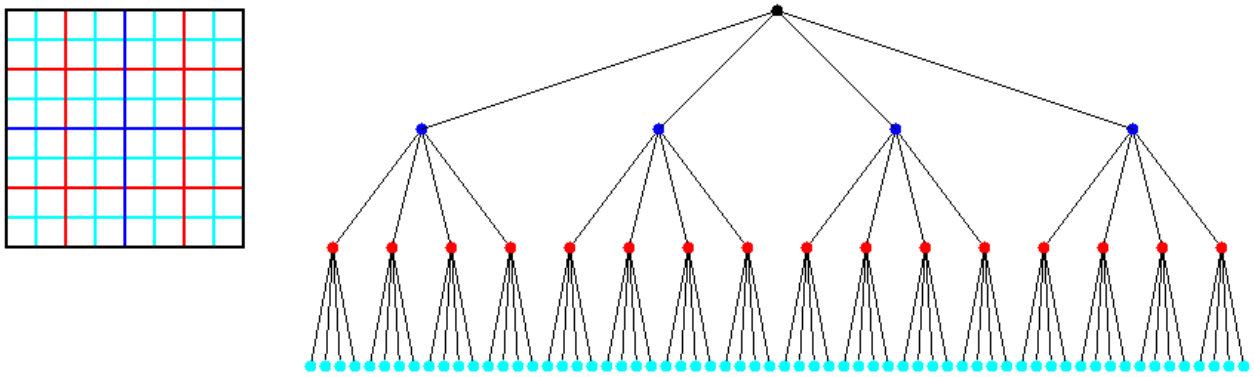


Figura 3. Construcción de un Árbol cuaternario con 4 niveles.

El objetivo de utilizar esta representación es almacenar las partículas de forma jerárquica, manteniendo información de las distancias entre ellos y de esta forma facilitar el cálculo del centro de masas. De esta forma, las hojas del árbol contendrán as posiciones y las masas de las partículas en el cuadrante correspondiente.

A la hora de construir el árbol también hay que tener en cuenta que las partículas no están distribuidas uniformemente en el espacio, por lo cual alguno de los cuadrantes en que se divide el espacio pueden estar vacío. En este caso, tiene sentido subdividir recursivamente solo aquellos cuadrantes que tengan un mínimo número de partículas (al menos una o un número pequeño). Esto conlleva, la creación de árboles adaptados al número de partículas, como el mostrado en la figura 4.

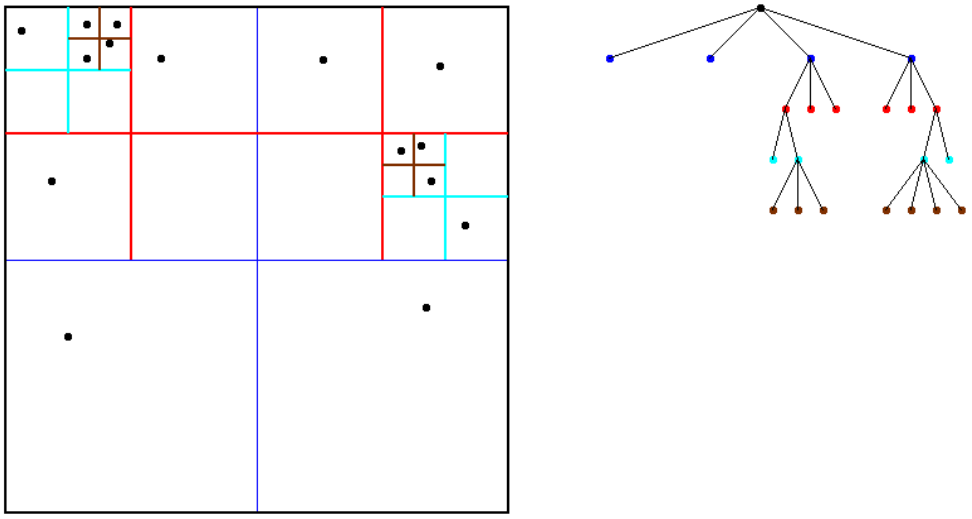


Figura 4. Árbol cuaternario adaptado, donde ningún cuadrante contiene más de una partícula.

Dado un conjunto de partículas, el árbol cuaternario se puede construir recursivamente utilizando el siguiente algoritmo:

```
procedure QuadtreeBuild
   Quadtree = {vacio}
   For i = 1 to n                          /* bucle sobre todas las partículas */
      QuadInsert(i, root)                  /* Insertar cuerpo i en el árbol */
   end for
   /* En este momento, el árbol puede contener algunas hojas vacías, cuyos nodos hermanos no estén
      vacíos */
   Recorrer el árbol eliminando las hojas vacías.

procedure QuadInsert(i,n)
   /* Intentar insertar el cuerpo i en el nodo n del quadtree*/
  /* Por definición, cada hoja contendrá 1 o 0 cuerpos */
   if el subárbol cuya raíz es n contiene más de 1 cuerpo
      Determinar el hijo c del nodo n al cual pertenece el cuerpo i (en función de su posición)
      QuadInsert(i,c)
   else if el subárbol cuya raíz es n contiene solo un cuerpo
      /* n es una hoja */
      Añadir los 4 hijos de n al árbol Quadtree
      Mover el cuerpo que está en n hacia el hijo al cual pertenece (en función de su posición)
      Determinar el hijo/cuadrante c correspondiente al cuerpo i */
      QuadInsert(i,c)                      /* Insertar cuerpo en su cuadrante/hoja */
   else if el subárbol cuya raíz es n está vació
       / n es una hoja */
      Almacenar el cuerpo i en el nodo n
   endif
```

La complejidad de algoritmo de construcción del árbol cuaternario depende de la distribución de las partículas dentro de los distintos cuadrantes. El coste para insertar una partícula es proporcional a la distancia desde la raíz del árbol hasta la hoja en donde la partícula está ubicada.

El algoritmo de Barnes-Hut.

Liu and Bhatt han propuesto en 1986 el algoritmo de Barnes-Hut para la resolución del problema de N-Body[1], basado en la simplificación del centro de masas y en los árboles cuaternarios. El resultado es un algoritmo eficiente que se adapta muy bien para su ejecución en ordenadores multi-procesador.

A alto nivel, el algoritmo de Barnes-Hut se divide en 3 pasos principales:

1) Construir el árbol cuaternario, utilizando el algoritmo visto anteriormente.
2) Para cada cuadrante en el árbol, calcular su centro de masas y la masa total, teniendo en cuenta todas las partículas que contiene.
3) Para cada partícula, recorrer el árbol y calcular la fuerza que incide sobre él.

El primer paso del algoritmo de Barnes-Hut ya se ha presentado previamente. El segundo paso del algoritmo se logra recorriendo el árbol en post-orden (los nodos hijos son visitadas/procesadas antes que el nodo padre) y almacenando la masa y el centro de masas calculados para subárbol n en el nodo n. El algoritmo para este paso sería el siguiente:

[1] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. Nature, 324(4):446-449, December 1986.

```
/* Calcular el centro de masas y la masa total para las partículas de cada cuadrante */
(masa, cm) = CalcularMasa(root)

function (masa, cm) = CalcularMasa(n)
   /* Calcular la masa y el centro de masas (cm) de todas las partículas en el subárbol n */
   if n contiene 1 cuerpo
      /* La masa y cm de n son idénticas a la posición y la masa del cuerpo */
      Almacenar (masa, cm) en n
      return (masa, cm)
   else
      for los cuatros hijos c(i) de n (i=1,2,3,4)
         (masa(i), cm(i)) = CalcularMasa(c(i))
      end for
      /* La masa del nodo n es la suma de las masas de los hijos. */
      masa = masa(1) + masa(2) + masa(3) + masa(4)
      /* El centro de masas del nodo n es la suma ponderada de los centros de masas de los hijos. */
      cm = (masa(1)*cm(1) + masa(2)*cm(2)
           + masa(3)*cm(3) + masa(4)*cm(4)) / masa
      Almacenar (masa, cm) en n
      return (masa, cm)
   end
```

El coste de este algoritmo es proporcional al número de nodos en el árbol, el cual no es mayor que O(n) modulo la distribución de las partículas (es decir O(n log n)).

Finalmente, en el tercer paso del algoritmo de Barnes-Hut, se utiliza la simplificación del centro de masas en función de la ratio presentada en la ecuación 1. De esta forma si la ratio es lo suficientemente pequeño, se puede calcular la influencia de todo un cuadrante de partículas, utilizando su centro de masas y su masa.

De esta forma si la ratio es menor que β (normalmente un poco menor que 1), podemos calcular la fuerza gravitacional de la partícula de la siguiente forma.

Siendo:

- (x, y, z) la posición en el espacio de la partícula (3D).
- m el centro de masas de la partícula.
- (x_{cm}, y_{cm}, z_{cm}) la posición del centro de masas de las partículas del cuadrante.
- m_{cm} la masa total de las partículas del cuadrante.
- G la constante gravitacional.

Entonces, la fuerza gravitacional que atrae la partícula se puede aproximar por:

$$Fuerza = G \times m \times m_{cm} \times \left[\frac{x_{cm} - x}{r^3}, \frac{y_{cm} - y}{r^3}, \frac{z_{cm} - z}{r^3}\right] \quad (2)$$

Donde:

$$r = \text{Sqrt}((x_{cm}-x)^2 + (y_{cm}-y)^2 + (z_{cm}-z)^2) \quad (3)$$

es la distancia desde la partícula hasta el centro de masas de las partículas del cuadrante.

Por lo tanto, el tercer paso del algoritmo Barnes-Hut seria:

```
/* Para cada partícula, recorrer el árbol para calcular la fuerza que le afecta */
For i = 1 to n
   f(i) = TreeForce(i,root)
end for

function f = TreeForce(i,n)
   /* Calcular la fuerza gravitacional sobre la partícula i proveniente de todas las partículas que están
     dentro del cuadrante n */
   f = 0
   if n contiene una partícula
      f = Fuerza calculada según la expresión 2
   else
      r = Distancia desde la partícula i hasta el centro de masas del cuadrante n
      D = Tamaño del cuadrante n
      Ratio = D/r
      if Ratio < β
         Calcular f utilizando la expresión 2
      else
         for todos los hijos c de n
            f = f + TreeForce(i,c)
         end for
      end if
   end if
```

Vamos a implementar el NBody de forma concurrente de forma que permita realizar la simulación de forma más rápida.

Habrá que implementar dos versiones de la práctica: una en C utilizando pthreads y otra en java utilizando los threads de java. Implementar la aplicación para que soporte a una longitud variable de partículas y en el número de hilos a utilizar para realizar la simulación. Por defecto, se utilizarán 4 hilos de ejecución.

El código secuencial proporcionado para la versión C y la versión java son diferentes y tienen diferentes características y parámetros. Por lo tanto, los requisitos y el diseño de la versión concurrente variarán entre las dos versiones.

NBody Concurrente/Paralelo.

La resolución de este problema se puede abordar distribuyendo el cálculo de las partículas del árbol cuaternario entre varios hilos de ejecución de forma que el cálculo del centro de masas y de las nuevas posiciones y velocidades se realicen en paralelo por parte de los distintos hilos de la aplicación.

Para abordar la realización de la práctica habrá que efectuar todos los pasos de diseño e implementación requeridos por una aplicación paralela:

1.- **Entender el problema planteado**. Analizar el funcionamiento de la aplicación que se quiere implementar: puntos calientes, posibles cuellos de botella, dependencias, etc.

2.- **Particionado del problema**. Analizar cómo se puede descomponer el problema entre los diferentes procesos/hilos.

3.- **Implementación y depuración**. Una vez realizado el diseño de la aplicación paralela habrá que implementarla, teniendo en cuenta las dependencias que pueden existir entre los diferentes pasos de la simulación.

4.- **Análisis de las prestaciones de la aplicación**. La última fase del desarrollo de la práctica consistirá en analizar y mejorar sus prestaciones. Habrá que analizar speed-up obtenido por la aplicación. Una parte importante de la evaluación de la práctica dependerá de su eficiencia de esta.

La aplicación concurrente debe ser diseñada e implementada para que se pueda ejecutar con cualquier número de threads (M), especificado por el usuario mediante parámetros.

En ambas versiones, la aplicación puede mostrar gráficamente por pantalla la evolución de las partículas en el espacio. La versión concurrente en C deberá disponer de un hilo específico (a más a más de los M definidos por el usuario) para el dibujo de las partículas, de forma que la salida gráfica se pueda hacer de forma independiente del cálculo de la siguiente iteración de simulación. La versión java, ya tiene implementada la visualización gráfica mediante un hilo.

En el diseño de la práctica debéis analizar las posibles dependencias entre los diferentes pasos de algoritmo. Para gestionar esas dependencias se deberá coordinar/sincronizar la ejecución de los hilos. Sin embargo, esta práctica todavía no disponemos de mecanismos de sincronización específicos (semáforos, etc..). Por ello, la coordinación de los hilos se implementará mediante *joins,* y por lo tanto, esperando a que un determinado hilo finalice. Esto puede implicar que se tenga que crear hilos de forma dinámica en diversos puntos del algoritmo. Esto se admite, siempre que de forma simultánea solo haya un máximo de M+1 hilos en ejecución al mismo tiempo.

También debéis tener en cuenta que, al no disponer en esta práctica, de mecanismos de sincronización de grado fino, no se puede (para evitar errores e incoherencias) permitir que dos hilos puedan escribir simultáneamente en la misma variable compartida. Si que se permite leer la misma variable compartida o escribir en diferentes variables compartidas.

Ante cualquier error que se produzca en la aplicación concurrente, se debe finalizar la misma de forma controlada, finalizando todos los hilos que estén en ejecución y liberando los recursos solicitados (memoria, ficheros, etc.).

Funciones involucradas.

En la práctica podéis utilizar las siguientes funciones de c:

- ```
  int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_routine) (void *), void *arg);
  ```
- ```
  int pthread_join(pthread_t thread, void **retval);
  ```
- ```
  void pthread_exit(void *retval);
  ```
- ```
  int pthread_cancel(pthread_t thread);
  ```

En la práctica podéis utilizar las siguientes clases y métodos de java:

- *clase java.lang.Thread*
  - *run()*
  - *start()*
  - *join()*

- *interrupt()*
- *isAlive()*
- *currentThread()*

Análisis prestaciones

Calcular el tiempo de ejecución de la versión concurrente y compararlo con el de la versión secuencial y la versión secuencial con un solo hilo. Discutir los resultados obtenidos. El objetivo de la práctica es que **la versión concurrente sea más rápida que la versión secuencial**.

Analizar también, cómo evoluciona el tiempo de ejecución en función del número de hilos utilizados en la aplicación concurrente, tanto para la versión C, como para la versión java. Entregar un pequeño informe en donde se muestren (preferiblemente de forma gráfica) y expliquen los resultados obtenidos.

Indicar en el informe las características del hardware en donde habéis obtenido los resultados (especialmente el número de procesadores/núcleos).

Evaluación

La evaluación de la práctica se realizará en función del grado de eficiencia/concurrencia lograda.

Se utilizarán los siguientes criterios a la hora de evaluar las dos implementaciones de la práctica, partiendo de una nota base de 5:

- Aspectos que se evalúan para cada una de las versiones:
  - Ejecución Secuencial (Susp)
  - Ejecución no determinista ([-1.0p,-5.0p])
  - Prestaciones (-1.0p, +0.5p)
  - No mejora el tiempo de la versión secuencial (-2.0p)
  - Niveles de concurrencia implementados (-0.75p, cada uno que falte)
  - Número incorrecto de hilos (-0.5p)
  - Join hilos (-0.5p)
  - Control Errores: cancelación hilos (-0.5p)
  - Condiciones de carrera (-1.0p)
  - No liberar memoria: (-0.25p,-0.5p)
  - Descomposición desbalanceada (Versión óptima) (-0.5p)
  - Descomposición parcial: falta asignar tareas a los hilos (-0.5p)

- Evaluación del informe de la práctica:

  + Informe completo, con gráficos y conclusiones correctas (+0.25p,+1.0p)

  - Sin informe (-1.0p)
  - Informe con incoherencias en los tiempos sin justificar (-0.5p)
  - Informe: análisis poco riguroso (-0.5p)
  - No se explica el diseño de la solución propuesta (-0.5p)
  - Descripción de los problemas encontrados y como se han solventado.

- Estilo del código.
  - Comentarios
  - Utilizar una correcta identación
  - Descomposición Funcional (Código modular y estructurado)
  - Control de errores.
  - Test unitarios para verificar la correcta ejecución de la lógica de la aplicación.

Se puede tener en cuenta criterios adicionales en función de la implementación entregada.

Versiones Secuenciales

Junto con el enunciado se os proporciona varias versiones secuenciales (en C y Java) para la simulación de las partículas. Podéis utilizar dichas versiones para implementar las versiones concurrentes que se os pide en esta práctica. En principio, no se permite utilizar otra versión secuencial.

- **Versión NBody en C.**

La versión secuencial en C se denomina NBody_BarnesHut y es una versión adaptada del algoritmo publicado en la siguiente URL: https://github.com/lang22/MPI-NBody. La versión adaptada para la práctica la tenéis disponible en el archivo comprimido NBody.zip que os pasamos en la actividad.

Está versión implementa el algoritmo de Barnes Hut basado en árboles cuaternarios comentados en la práctica.

Esta versión utiliza la libraría de OpenGL GLFW para la visualización gráfica de la simulación. Debido a que esta librería no está instalada en los ordenadores del laboratorio, se ha adaptado el código para que mediante la definición de una constante (**D_GLFW_SUPPORT**) active/desactive la compilación de la interfaz gráfica. Por ello, el makefile de compilación soporta dos modos:

- Compilar incluyendo la salida gráfica de la simulación:
  - make all BUILD=gui

    que crea el ejecutable NBody_GUI

- Compilar sin incluir la salida gráfica de la simulación:
  - make all BUILD=no_gui

    que crea el ejecutable NBody

Para ejecutar la versión secuencial, únicamente tenéis que especificar obligatoriamente el número de partículas a simular y el número de iteraciones que se simularán. Opcionalmente se puede especificar un fichero con los valores iniciales de las partículas a simular y un cuarto parámetro (con cualquier valor) que activa la visualización grafica (en el caso de estar soportada). En el caso que el número de partículas especificado en el primer parámetro difiera del que hay en el fichero de entrada, entonces se tienen en cuenta el del fichero.

Sintaxis: NBody* <NumParticulas> <NumIteraciones> [FicheroParticulas] [ActivarSalidaGráfica]

  - `NBody* 100 1000 ./galaxy_1000B_initial.out 1`

La ejecución genera, en el directorio res, un fichero con el estado de la simulación (posición, velocidad y masa de todas las partículas) al inicio, final y a ciertos intervalos de la simulación.

En esta versión se debe implementar la ejecución concurrente de la simulación con M hilos, tanto para la versión texto, como gráfica. Además, se añadirá un hilo adicional para que la versión gráfica visualicé el estado de simulación de forma concurrente con el cómputo de la siguiente iteración de simulación.

Con el enunciado se incluye varios ficheros de entrada de diferentes tamaños de simulación y duración (100, 1000, 5000, 10000 y 25000 partículas), junto a los ficheros con el estado obtenido después de simularlos durante 1000 iteraciones. Podéis utilizar estos ficheros para validar la práctica y realizar el análisis de prestaciones.

Para soportar la visualización gráfica de la simulación se necesita instalar los siguientes paquetes (requisitos para instalar la librería GLFW):

- Instalar dependencias GLFW:
  - sudo apt install xorg-dev
  - (opcional) sudo apt install libXcursor-devel libXi-devel libXinerama-devel libXrandr-devel
  - sudo apt-get install libglfw3-dev
- Descargar y compilar GLFW (https://www.glfw.org/docs/3.3/compile.html)
  - cmake -S path/to/glfw -B path/to/build
  - cd path/to/build
  - make

- **Versión NBody en Java.**

La versión secuencial en java se denomina NBody_jos y es una versión adaptada del algoritmo publicado en la siguiente URL: https://sourceforge.net/projects/jos-n-body/. La versión adaptada para la práctica la tenéis disponible en el archivo comprimido NBody_jos.zip que os pasamos en la actividad.

Está versión implementa tres variantes para la simulación NBody, una versión basada en GPUs utilizando la librería Aparapi, otra versión secuencial y otra hibrida (modo AUTO). La aplicación admite tres tipos de precisión numérica flotante (precisión simple), precisión doble y precisión arbitraria. Para ello la aplicación introduce una abstracción de números que permite elegir qué implementación utilizar: tipos primitivos de Java float y double, o ApFloat de precisión arbitraria.

Estas versiones están soportadas en el mismo código y se utiliza una u otra en función de los parámetros seleccionados en la interfaz gráfica para escoger el modo de funcionamiento y la precisión utilizada.

La versión basada en CPU utiliza el procesador principal y admite las tres precisiones numéricas. La versión GPU se ejecuta en la tarjeta de video y puede estar deshabilitado dependiendo del hardware disponible en vuestro ordenador, en concreto de que la tarjeta gráfica soporte OpenCL. La precisión de la versión GPU también depende del hardware. La mayoría de las tarjetas de video admiten OpenCL y al menos precisión basada en floats. En el modo AUTO cambia de GPU a CPU cuando la cantidad de objetos cae (al fusionarse) por debajo de un umbral particular, sin embargo, este modo ha sido modificado para que solo utilice exclusivamente la CPU.

Nosotros vamos a trabajar con la versión secuencial (modo CPU[2]). Esta versión es más sencilla ya que no implementa la optimización del algoritmo de Barnes Hut basado en árboles cuaternarios. El objetivo es que esta versión utilice N hilos de ejecución para ejecutar concurrentemente la simulación para las versiones de precisión simple y doble. Esta versión ya implementa la gestión de la interfaz gráfica mediante un hilo de ejecución independiente por lo que no es necesario implementarlo en la práctica.

En esta versión se debe implementar la ejecución concurrente de la simulación con M hilos cuando de activa el modo CPU, tanto para la precisión simple como la doble.

La implementación proporcionada consiste en un proyecto *Intellij*. Los parámetros de simulación, incluyendo el modo de ejecución, como el número de hilos utilizado en la versión concurrente se especificarán mediante la interfaz gráfica de la aplicación. Estos parámetros de simulación (salvo el número de hilos) se pueden salvar en un fichero json. Con el código de la aplicación, en el directorio Test, os adjuntamos 6 ficheros de configuración (200, 1000 y 2000 partículas para las dos precisiones y simulados con 2000 iteraciones) con los correspondientes ficheros de salida (en el directorio Test/Output).

Formato de entrega

MUY IMPORTANTE: La entrega de código que no compile correctamente, implicará suspender TODA la práctica.

No se aceptarán prácticas entregadas fuera de plazo (salvo por razones muy justificadas).

La entrega presencial de esta práctica es obligatoria para todos los miembros del grupo.

Comenzar vuestros programas con los comentarios:

```
/* ------------------------------------------------------------
Práctica 1.
Código fuente : gestor.c
Grau Informàtica
NIF i Nombre completo autor1.
NIF i Nombre completo autor2.

------------------------------------------------------------ */
```

[2] `simulationLogic.getExecutionMode()==CPU`

Para presentar la práctica dirigiros al apartado de Actividades del Campus Virtual de la asignatura de Sistemas Concurrentes y Paralelos, ir a la actividad "Práctica 1" y seguid las instrucciones allí indicadas.

Se creará un fichero tar con todos los ficheros fuente de la práctica, con el siguiente comando:

```
$ tar -zcvf prac1.tgz fichero1 fichero2 ...
```

se creará el fichero "prac1.tgz" en donde se habrán empaquetado y comprimido los ficheros "fichero1", fichero2, y ...

Para extraer la información de un fichero tar se puede utilizar el comando:

```
$ tar -zxvf tot.tgz
```

El nombre del fichero tar tendrá el siguiente formato: "Apellido1Apellido2PRA2.tgz". Los apellidos se escribirán sin acentos. Si hay dos autores, indicar los dos apellidos de cada uno de los autores separados por "_". Por ejemplo, el estudiante "Perico Pirulo Palotes" utilizará el nombre de fichero: `PiruloPalotesPRA1.tgz`

Fecha de entrega

Entrega a través de Sakai el 24 de octubre, entrega presencial en el grupo de laboratorio del 25 de octubre de 2022.