eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Paralelización de códigos científicos de simulación de partículas

Autor

Julen Galarza

2022

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Paralelización de códigos científicos de simulación de partículas

Autor

Julen Galarza

Director

Jose A. Pascual y Javier Navaridas

Resumen

El trabajo realizado ha consistido en mejorar y optimizar un programa de simulación de partículas para tratar de aprovechar al máximo la capacidad de cómputo del supercomputador ATLAS del DIPC (Centro de Supercomputación Donostia International Physics Center). Para ello, se partía de un programa inicial **mpc_run** creado por la empresa ESS Bilbao, el cual simulaba el recorrido y las colisiones de electrones dentro de una pieza durante un suceso llamado "Multipacting".

El programa inicial estaba hecho en python y se ejecutaba en serie, aunque tenía también una versión multithreading, pero el motor de python evitaba que se ejecutaran en paralelo, por lo que el objetivo principal del proyecto era lograr que funcionase dicha versión logrando el mayor speed-up posible respecto a la versión en serie.

Por lo tanto, las tres partes principales del trabajo realizado han sido las siguientes: Por un lado, analizar el programa inicial, su funcionamiento y las mejores opciones para paralelizar la ejecución. Después, implementar los cambios adecuados para los distintos modos de ejecución del programa, para tratar de sacar el máximo partido a cada uno de ellos. Por último, analizar todos los resultados obtenidos tras los cambios realizados y el funcionamiento del programa, pensando también en posibles mejoras que se podrían realizar de cara al futuro.

Índice general

Índice de figuras

Índice de tablas

1. CAPÍTULO

Introducción

ESS Bilbao[1] es un centro estratégico de referencia internacional en Tecnologías Neutrónicas que aporta conocimiento y valor añadido a través de la contribución en especie al proyecto europeo de ESS, infraestructura de científica que se está construyendo en la ciudad sueca de Lund. Además de esto, también trabajan en proyectos propios en el mundo de la aceleración de partículas y los neutrones [Pérez et al., 2020].

Estos aceleradores de partículas están compuestos por distintas piezas y una parte importante a la hora de diseñar y construir estas piezas es simular el comportamiento de las partículas dentro de las mismas. En este TFG, se ha trabajado con uno de los programas que tienen para estas simulaciones, el cual se utiliza principalmente para analizar un suceso llamado Multipacting que puede suceder dentro de las mencionadas piezas.

El Multipacting o Efecto Multipactor [Vaughan, 1988] es una descarga en avalancha de electrones generada por la sincronización entre un campo eléctrico alternativo intenso y la emisión de electrones secundarios (SEY) en las superficies expuestas a los electrones acelerados por el campo en condiciones de vacío. En la Figura 1.1 se puede ver como el proceso comienza cuando un electrón que entra dentro de una de las mencionadas piezas colisiona con la superficie de la propia pieza, y por consecuencia de la colisión se generan otro/s electrón/es. Estos electrones a su vez pueden colisionar y si este suceso se va repitiendo puede crearse una nube de electrones y generarse lo que se llama multipacting o efecto multipactor.

[1] https://www.essbilbao.org/es/

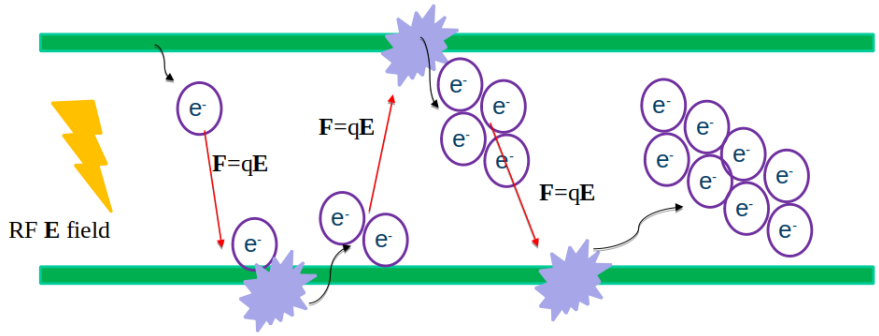

Figura 1.1: Multipacting o Efecto Multipactor

Analizar en qué condiciones es más probable que se forme esta avalancha de electrones en cada pieza puede ser importante, ya que, normalmente, se trata de evitar que suceda, ya que puede influir en las partículas del acelerador y, en el peor de los casos, incluso dañar el equipamiento. En algunas ocasiones, también es interesante crear la avalancha a propósito bajo condiciones controladas.

En cualquier caso, para asegurar la fiabilidad de este análisis, necesitamos simular cantidades muy grandes de electrones y además realizar un gran número de ejecuciones, por lo que el tiempo de ejecución de un programa de estas características que simule el recorrido de cada uno de los electrones en serie puede ser muy alto, del orden de horas, o incluso días.

Con el objetivo de mejorar este tipo de simulaciones, en este TFG se ha tratado de optimizar al máximo el programa mediante la paralelización de su ejecución para ejecutarlo en el supercomputador ATLAS del DIPC[2], tratando de conseguir el mejor speed-up posible reduciendo así los tiempos de ejecución y permitiendo a su vez ejecutar el mismo programa con mayores cantidades de electrones.

Durante el desarrollo del proyecto, se han implementado tres versiones distintas del programa. Para la primera se ha utilizado el nombre **MultiProcessing** y se ha paralelizado uno de los modos de ejecución del programa, comenzando con un único electrón inicial y una sola potencia.

Después, para otro de los modos de ejecución del programa, donde se simulan varios electrones iniciales en un rango de potencias, se implementó una primera versión llamada **2Pools**. Más adelante, para mejorar algunos aspectos, se implementó la versión final del programa, llamada **OnePool**.

[2] http://dipc.ehu.es/

2. CAPÍTULO

Objetivos del Proyecto

El objetivo principal de este proyecto consiste en mejorar un programa de simulación de partículas ya existente mediante la paralelización del código, para poder aprovechar mejor las capacidades del supercomputador ATLAS. Para tratar de lograr esto, primero era importante también analizar el programa original y el impacto de algunas de las opciones del mismo para mejorarlo en todos los aspectos posibles.

En el estado inicial del programa, únicamente se podía ejecutar en un PC personal y había una versión multithreading, pero el motor de python evitaba que se ejecutaran en paralelo. Por esta razón, el primer objetivo que se marcó fue tratar de obtener el mayor speed-up posible en la versión paralela respecto a la versión en serie al ejecutarse en el supercomputador ATLAS.

Sin embargo, antes de recibir el código del programa, no se sabía con exactitud que aspectos iban a ser necesarios modificar ni hasta que punto iba a ser posible realizar mejoras significantes, por lo que durante la propia elaboración del proyecto fueron surgiendo también otros objetivos colaterales o secundarios que han servido para mejorar el programa.

Teniendo esto en cuenta, los siguientes son todos los objetivos relacionados con la elaboración del proyecto, incluyendo algunos que surgieron durante la propia elaboración del mismo:

- Preparar el programa informático para ejecutarse en el supercomputador ATLAS.
- Analizar el funcionamiento de la versión original y estudiar el impacto de los parámetros en el tiempo de ejecución.
- Mejorar los tiempos de ejecución del programa mediante la paralelización del código.
- Analizar la versión paralela y proponer mejoras a la escalabilidad del programa.
- Mejorar el sistema para recopilar datos de las simulaciones.

3. CAPÍTULO

Contexto del Proyecto

El programa utilizado en el TFG sirve para simular el recorrido de electrones en una pieza durante un suceso llamado Multipacting o Efecto Multipactor, el cual se ha explicado en el Capítulo 1.

Para entender mejor todos los cambios realizados, en este capítulo se van a explicar los archivos, parámetros y el funcionamiento del programa original desarrollado por ESS Bilbao, así como las entidades involucradas en el proyecto y las herramientas fundamentales utilizadas durante la realización del mismo.

3.1. Archivos y parámetros del programa original

El programa inicial del que se ha partido en el proyecto, es un programa desarrollado por ESS Bilbao que sirve principalmente para simular el recorrido de los electrones dentro de una pieza. Para poder simular la forma de las distintas piezas se utilizan unos archivos con un formato determinado que representan unos mallados utilizando triángulos o tetraedros.

Antes de explicar el funcionamiento del programa, vamos a ver las carpetas y archivos que lo componen:

- **Carpeta bin:** Es donde se encuentran los ficheros de python del programa. En concreto son 3 archivos:
  - mpc_dolfin.py: Es un fichero en el que se encuentra definida la clase "multipacting" junto a todas las funciones que se utilizan en el programa.
  - mpc_run.py: Es el fichero principal del programa, que se encarga básicamente de recoger los parámetros desde un fichero externo para inicializar la clase "multipacting" y ejecutar las funciones de la misma.
  - parameter_list.py: Es un archivo con la lista de parámetros del programa y sus respectivos tipos (booleano, entero, carácter, etc). Se utiliza en "mpc_run.py" a la hora de leer los parámetros desde el archivo de entrada.
- **Carpeta data:** En esta carpeta se guardan los datos de las piezas. Por cada pieza hay un fichero con extensión ".h5", que son los mallados en formato HDF5, un estándar similar a XML, pero binario y que lee directamente el programa. La información de los mallados 3D es compleja porque no solo tienen información sobre los nodos y los elementos del mallado, sino también sobre la topología y sobre medidas interiores. Además de esto, por cada fichero ".h5" hay otro fichero de mismo nombre, pero con terminación ".txt" que utiliza el programa para leer información sobre los datos que contiene el mallado.
- **Carpeta generated_files:** En realidad esta carpeta no estaba en el programa original, pero fue una de las primeras modificaciones, para almacenar dentro todos los archivos que crea el programa y que no molesten en otros directorios. Principalmente, se crean ficheros de log o archivos que almacenan datos sobre las simulaciones.
- **Archivos .mpc:** Es el archivo que, mediante un formato determinado, contiene la información que necesita el programa para leer los parámetros de entrada.
- **mpc_run.py:** Es un enlace simbólico al archivo que se encuentra dentro de la carpeta bin y sirve para ejecutar el programa desde el directorio en el que se encuentra el enlace. Algunos directorios/ficheros están referenciados dentro del código del programa, por lo que es necesario ejecutarlo de esta manera para que pueda acceder a los mismos y funcione correctamente.
- **setup_atlas_example.sh:** Era un script de ejemplo que utilizaban en ESS Bilbao para mandar a ejecutar programas en el superordenador ATLAS del DIPC. Sin embargo, no estaba preparado para este programa en concreto, por lo que más adelante se modificó.

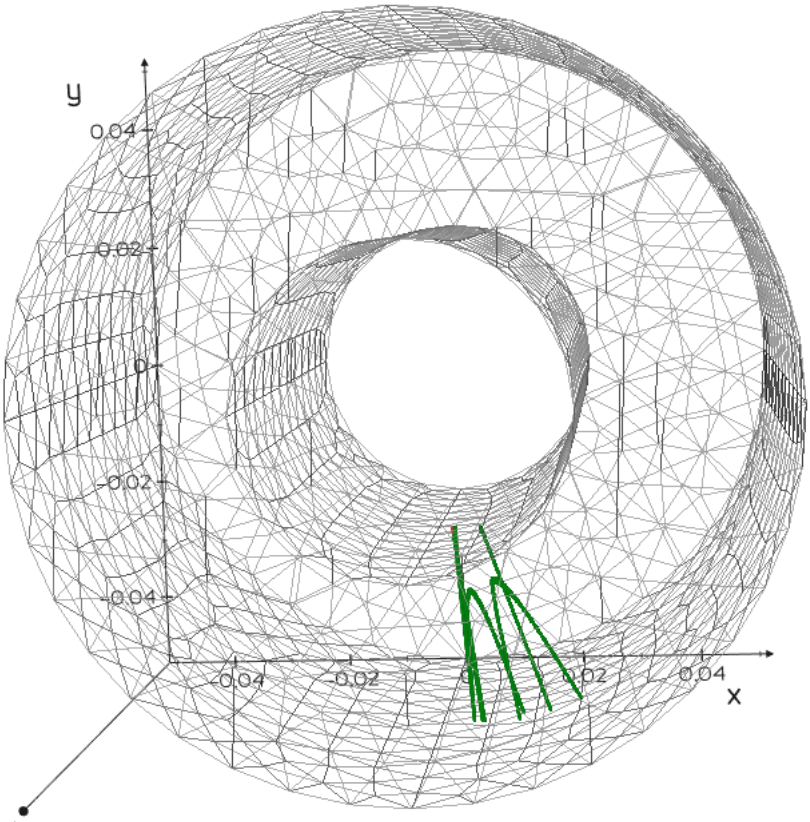

Figura 3.1: Recurso visual generado al utilizar la opción plot.

El programa tiene tres modos principales distintos de ejecución:

1. **Single electron trayectory:** Se simula la trayectoria de un único electrón.
2. **Single electron multipacting:** Se simula comenzando con un único electrón en una única potencia y los electrones secundarios creados por este.
3. **Power range:** Se simula un comienzo con varios electrones y todos los electrones secundarios creados por los mismos dentro de un rango de potencias.

Además de esto, se utilizan varios parámetros que se introducen mediante la lectura de un archivo de entrada con un formato determinado. El archivo (terminación .mpc) tiene un total de 17 parámetros, que se pueden ver en la Tabla 3.1.

Parámetros	Función/Descripción
version	Número de versión del programa, es únicamente informativo e irrelevante para el proyecto.
parallel	Número de hilos/procesos con el que queremos ejecutar el programa.
log	True/False, Si es True se creará un fichero log de nombre "log_mpc_py_fecha.txt" que almacena información sobre los electrones simulados.
debug	True/False, en un principio este parámetro no tenía ninguna función.
plot	True/False, Si es True mostrará al final de la ejecución del programa el recorrido de los electrones en la pieza de manera visual (ver la Figura 3.1). También generará un archivo por cada electrón simulado de nombre "N-2022-fecha-..." que contendrá el recorrido del mismo mediante las distintas posiciones dentro de la malla en las que se ha encontrado. Estos archivos son necesarios para mostrar el recorrido de manera visual.
verbose	True/False, si es True mostrará por la salida información sobre la ejecución al inicio de la misma.
electrons_seed	Número inicial de electrones por cada potencia (para el modo de ejecución power range).
max secondary runs	Número máximo de generaciones de electrones.
delta_t	Un tiempo en segundos que se utiliza dentro de los cálculos del campo magnético.
RF_frequency	Frecuencia en hercios que se utiliza en los cálculos del campo de Radiofrecuencia (RF).
N_cycles	Número de ciclos de radiofrecuencia por cada electrón, utilizada dentro de los cálculos para simular el comportamiento de un electrón.
RF_power	La potencia o el conjunto de potencias (para el modo power range) en vatios con el que queremos simular los electrones. Puede ser un valor único, un rango de potencias o una lista de potencias.
simulation_type	El modo de simulación anteriormente explicado. 1 para "power range", 2 para "single electron trajectory" y 3 para "single electron multipacting".
data_file	Es la ruta de un fichero con un formato determinado que sirve para que el programa pueda simular el mallado de la pieza. Este archivo tiene una terminación .txt.
mesh_file	Es la ruta de un fichero con terminación .h5 correspondiente al fichero .txt que tiene la misma función. Ambos ficheros son necesarios.
boundaries excluded boolean	Pueden ponerse varias líneas de manera acumulativa y sirve para indicar algunos límites de la pieza simulada.
plot_title	El título de la figura que se crea si el parámetro plot es True.

Tabla 3.1: Función y/o descripción de los parámetros del programa original

3.2. Funcionamiento del programa original

En cualquier modo de ejecución, lo primero que hace el programa es leer los parámetros de entrada y crear la clase multipacting en base a esos datos, con la información del mallado y del campo RF de la pieza. Después, la función más básica y principal es la que simula el recorrido de un electrón, a la que nos referiremos como "track_1_e".

En el modo de ejecución de **Single electron trayectory** es la única función necesaria para obtener la trayectoria del electrón. A continuación veremos una breve explicación de la función, junto a un pseudocódigo de la misma (Listing 3.1)

Listing 3.1: Pseudocódigo de track_1_e.

```
input: Electron, Power, Δt
output: Electron
begin
  while not collision(Electron.Pos)
    Electron.Acc = calculate_acceleration(Electron, Power)
    Electron.Pos = calculate_new_position(Electron, Δt)
  end
end
```

- **track_1_e:** Se le pasan por parámetro la posición inicial, potencia y fase del electrón.
  - Bucle principal, hasta que el electrón colisione con una de las caras de la pieza.
    - Se calcula la aceleración del electrón.
    - Con la aceleración se calcula la nueva posición del electrón.
    - Si la nueva posición está dentro de la pieza, sigue con el bucle, si no, se calcula la colisión.
  - Una vez el electrón ha colisionado y sale del bucle principal, la función devuelve los datos de la colisión (energía, lugar, fase, etc) y la trayectoria total del electrón.

En el modo de ejecución de **Single electron multipacting**, la función principal que utiliza se llama Run_n_electrons_parallel, la cual a su vez utiliza la función mencionada para simular los electrones. En este caso el pesudocódigo sería el siguiente: (Listing 3.2)

Listing 3.2: Pseudocódigo de Run_n_electrons_parallel.

```
input: Electron, Generations
output: Electrons_per_generation
begin
  electron_list[0] = {electron}
  for g in [0, Generations)
    while not empty(electron_list[g])
      Electrons_per_generation[g]++
      e = first_element (electron_list[g])
      e' = track_1_e(e)
      gen_new_electrons(electron_list[g+1], e')
    end
  end
end
```

- **Run_n_electrons_parallel:** Se le pasan por parámetro el número de generaciones que se quieren calcular y la información del electrón inicial.
  - Bucle principal, hasta que se llegue al número de generaciones indicado.
    - Se crean listas con los datos de todos los electrones a simular (al comenzar será solo 1).
    - Un segundo bucle por cada electrón a simular.
      - Se llama a la función track_1_e y se obtienen los datos de la colisión.
    - Una vez simulados todos los electrones, se calcula la cantidad de electrones que se crean debido a las colisiones y se almacenan sus datos en nuevas listas para simularlos en la siguiente generación.
    - Se almacena/muestra la información de la generación y, si no se ha llegado al máximo número de simulaciones indicado, continúa con el bucle.
  - Una vez alcanzado en número máximo de generaciones a simular, la función devuelve el número total de electrones simulados y la cantidad de electrones generados en la última generación.

Por último, en el modo de ejecución más complejo, **Power Range**, el funcionamiento es muy similar al de Single electron multipacting, pero añadiendo un bucle exterior que repite la función con cada una de las potencias a simular, con la particularidad también de que en lugar de haber un único electrón inicial, hay varios. A continuación el pseudocódigo (Listing 3.3) y una breve explicación del mismo:

Listing 3.3: Pseudocódigo de Power Range.

```
input: Power_range
output: Electrons_per_power
begin
  for p in Power_range
    Electrons_per_power[p] = Run_n_electrons_parallel(p)
  end
end
```

- **Run:** El parámetro de entrada sería el rango de potencias a simular.
  - Bucle principal, por cada potencia a simular:
    - Se calculan los electrones iniciales en dicha potencia y se almacenan sus datos en listas.
    - Se llama a la función Run_n_electrons_parallel, pasando como parámetro las listas con los datos de los electrones iniciales.
    - Se almacenan los resultados de la función y, si quedan potencias a simular, continua con el bucle.
  - Una vez simuladas todas las potencias, la función muestra el número total de electrones simulados en cada una de ellas.

3.3. Entidades Involucradas

- **DIPC**

  El DIPC (Donostia International Physics Center) nace como un centro intelectual cuyo principal objetivo es el de promocionar y catalizar el desarrollo al más alto nivel de la investigación básica y básica-orientada en ciencia de materiales. Desde su creación, el DIPC ha sido una institución abierta y ligada a la Universidad del País Vasco, que sirve como plataforma de internacionalización de la ciencia básica en el País Vasco en el campo de los materiales.

  En el DIPC se encuentra el supercomputador ATLAS, el cual es utilizado tanto por trabajadores internos como por empresas y organizaciones externas, a las que se les presta un servicio de asesoramiento y ayuda a la hora de utilizar y aprovechar las características del supercomputador. Este servicio es el que originó la realización del TFG, como ayuda a una de las entidades externas (ESS Bilbao) que utiliza el supercomputador ATLAS.

- **ESS Bilbao**

  ESS Bilbao (European Spallation Source Bilbao) es un centro estratégico de referencia internacional en Tecnologías Neutrónicas que forma parte del proyecto europeo ESS [1], que tiene su sede principal en la ciudad sueca de Lund. Desde el centro de Bilbao, principalmente se aporta conocimiento y valor añadido a través de la contribución en especie al proyecto europeo, pero además de eso también hay proyectos propios en el ámbito de la aceleración de partículas y las Tecnologías Neutrónicas.

  En cuanto a la relación de ESS Bilbao con este proyecto, es una de las empresas externas que utiliza el supercomputador ATLAS para ejecutar sus programas, y al cual le pertenece el programa de simulación de partículas original que se ha utilizado en el proyecto.

[1] https://europeanspallationsource.se/

3.4. Herramientas de trabajo

- **Librería Concurrent.Futures**

  El programa inicial que se ha utilizado estaba hecho con el lenguaje de programación python, y para la versión paralela se trataba de utilizar la librería concurrent.futures[2], aunque más adelante veremos que esta versión no mejoraba los tiempos de ejecución de la versión en serie.

  Esta librería provee una interfaz de alto nivel para poder ejecutar programas de manera asíncrona y simultánea. En la misma, hay dos módulos distintos: ThreadPoolExecutor y ProcessPoolExecutor. La diferencia entre estos dos módulos es que en el primero de ellos, ThreadPoolExecutor, la ejecución se puede repartir en distintos hilos de ejecución, mientras que en el segundo, ProcessPoolExecutor, la ejecución se divide en procesos distintos.

  La definición de los términos *hilos* y *procesos* puede variar dependiendo del contexto, pero en el caso del intérprete de python la diferencia principal es que los hilos no se pueden ejecutar en paralelo al mismo tiempo debido al GIL [3] (Global Interpreter Lock) de python, el cual no permite que distintos hilos de un mismo proceso se ejecuten de manera simultánea para evitar problemas de atomicidad y coherencia. Sin embargo, en el caso de los procesos que se crean mediante el módulo ProcessPoolExecutor, el GIL no tiene ningún impacto debido a que se consideran procesos totalmente independientes y no comparten espacio de memoria.

  A la hora de utilizar cualquiera de los dos módulos, las opciones son las mismas. Existe una clase abstracta que se llama “Executor” y que proporciona una serie de métodos para ejecutar las llamadas de manera asíncrona. Estos métodos son los siguientes:

  - **map:** Este método sirve para programar la ejecución de una función en paralelo con distintos parámetros de entrada y devolviendo al final un objeto de tipo “Future” que representa los resultados de dicha función. Por ejemplo, se incluyen los distintos parámetros con los que se quiere ejecutar la función en una lista y se pasa la lista como parámetro, para que se ejecute dicha función con todos los parámetros de la lista de manera simultánea. El número máximo

[2] https://docs.python.org/3/library/concurrent.futures.html
[3] https://realpython.com/python-gil/

de procesos/hilos distintos que pueden crear en un mismo momento se puede modificar con la variable “max_workers”.

- **submit:** Este ”método” tiene la misma función que el método *map*, pero se utiliza de una manera más “manual”. Es decir, en lugar de utilizar una única vez el método *map* con una lista de parámetros, hay que utilizar el método *submit* una vez por cada uno de los parámetros.
- **shutdown** Indica al ejecutor que tiene que cancelar todos los ejecutables cuando terminen los que actualmente están en ejecución.

Por último, la librería también incluye el mencionado objeto de tipo “Future”, que es una clase que sirve para obtener la información de las ejecuciones de la función. Se puede saber si alguna de las ejecuciones ha fallado y en caso de que la función devuelva algo, se almacenan los resultados en listas con el mismo orden que tenían los parámetros para poder saber a qué parámetros de entrada corresponde cada resultado de la función.

- **Librería Fenicsproject**

  Tanto para representar las piezas mediante mallados como para realizar algunos de los cálculos para determinar el recorrido de los electrones, se utiliza una librería de python que se llama Fenicsproject[4]. FEniCS es una plataforma informática de código abierto que sirve principalmente para resolver ecuaciones diferenciales parciales (PDE). Esta permite a los usuarios traducir rápidamente modelos científicos en código de elementos finitos eficiente y además está diseñado en su totalidad para el procesamiento paralelo al proporcionar soporte para la ejecución con MPI. Esto está particularmente optimizado para resolver las ecuaciones diferenciales electromagnéticas por el método de elementos finitos.

  Sin embargo, una vez calculada, la distribución del campo eléctrico se usa para calcular la trayectoria de los electrones, que es un cálculo muy sencillo, por lo que usar las capacidades MPI de la biblioteca no supondría ningún beneficio, ya que es un cálculo demasiado fino.

  De hecho, la mayor parte del trabajo pesado de cálculo se realiza fuera de la biblioteca y, por lo tanto, se requiere un enfoque diferente. Emplear paralelismo a nivel de electrones parecía el más adecuado y, por lo tanto, es el nivel de granularidad que se usa en nuestro código paralelo.

[4] https://fenicsproject.org/

Dado que no se necesita comunicación entre electrones para el tipo de simulaciones que estamos llevando a cabo, se consideró innecesario un modelo de programación *communication-oriented* (orientado a la comunicación) rico en características como MPI.

A pesar de que actualmente existe alguna versión más actual de la librería, se ha utilizado "FEniCS Legacy 2019.1.0", ya que es la versión instalada en el supercomputador ATLAS.

4. CAPÍTULO

Desarrollo del proyecto

En este capítulo, se van a explicar todos los cambios y mejoras que se han ido implementando a lo largo del desarrollo del proyecto. El primer paso, una vez recibido el código del programa por parte de ESS Bilbao, era analizar la versión inicial del programa para comprobar el impacto de algunas de las opciones y parámetros del mismo en el tiempo de ejecución.

Después de comprobar el funcionamiento del programa original y de realizar algunos cambios menores, se modificó el script setup_atlas_example.sh para poder ejecutar el programa en el supercomputador ATLAS.

Una vez que el programa ya se podía ejecutar en el supercomputador, el siguiente paso era analizar la opción **MultiThreading** del programa original, para comprobar las diferencias con la versión en serie y los cambios que se podían hacer para mejorar los tiempos de ejecución.

Después de comprobar el funcionamiento de la versión MultiThreading del programa original, se comenzó a implementar la primera versión modificada del proyecto, para la que se ha utilizado el nombre **MultiProcessing**. En esta primera versión el objetivo era paralelizar la ejecución en el modo de Single electron multipacting.

Una vez terminada la primera versión, el siguiente objetivo era paralelizar el código correspondiente al modo de Power Range, donde esta primera versión realizada no lograba obtener buenos resultados. En este caso, para la primera solución que se implementó se ha utilizado el nombre **2Pools**, pero más tarde se implementó otra versión distinta que a priori parecía mejor, a la que nos referiremos como versión **OnePool**.

A continuación, se van a explicar todos estos cambios y las características de las distintas versiones que se han realizado.

4.1. Análisis y modificaciones del programa original

Antes de comenzar a analizar el funcionamiento del programa en paralelo, se realizaron algunos análisis y cambios menores en el programa original:

4.1.1. Asegurar que la ejecución es determinista

El primer cambio realizado consistió en añadir un parámetro extra al programa de nombre "random_seed", que se utiliza para decidir cuál será la semilla que se utilizará para la generación de números aleatorios dentro del programa y así asegurar una ejecución determinista independientemente del número de procesos.

Esto es importante debido a que dentro de la simulación se utilizan probabilidades para determinar el comportamiento de los electrones. Estas están implementadas mediante la generación de números aleatorios, por lo que si ponemos una semilla fija, podremos repetir la misma simulación varias veces. Sin esta opción, no podríamos comparar de manera justa los tiempos de ejecución porque el programa crearía distintos números de electrones y con trayectorias diferentes en cada ejecución.

Para mantener también la opción de que la semilla sea aleatoria, se ha implementado de manera que si el parámetro random_seed es igual a -1 la semilla sea aleatoria, y si, por lo contrario, es un número mayor que 0, la semilla será ese mismo número.

Además de esto, la semilla dentro del programa principal se inicializa tanto al principio del programa como dentro de la simulación de un electrón, debido a que si se ejecuta en paralelo con varios cores, el orden de ejecución varía entre ejecuciones y, por lo tanto, a pesar de tener la misma semilla, las simulaciones serían distintas y no podríamos comparar los tiempos de ejecución.

Después de tener la semilla en funcionamiento, se podía repetir la misma simulación varias veces, por lo que se añadió una función para ver los tiempos de ejecución, tanto del programa entero como de cada una de las generaciones de electrones (llamadas en el programa secondary_runs).

También se añadió al programa la función de crear un fichero más, de nombre "exec_time_fecha.txt" y con un funcionamiento muy similar al fichero de log, pero en lugar de almacenar datos sobre la simulación y trayectoria de los electrones, almacena tanto los tiempos de ejecución del programa entero como el de las generaciones e incluso el de cada uno de los electrones.

De esta manera, se pueden empezar a sacar algunas conclusiones mediante la simulación del programa con distintos parámetros. Lo primero que se ha hecho ha sido buscar la semilla adecuada para generar multipacting (la cual dependerá de la pieza con la que se simule, en este caso se utilizó una de nombre "coaxial_704_01") para que se genere una cantidad grande de electrones.

Tras esto, se procedió a analizar el impacto de los parámetros **log** y **plot** en el tiempo de ejecución del programa, para comprobar si era necesario modificar las funciones de alguno de estos parámetros.

4.1.2. Comprobar impacto del parámetro **log** en el tiempo de ejecución

Una de las primeras pruebas que se hicieron fue ejecutar la misma simulación con el parámetro log como True y después como False, para ver si generar los archivos de log tenía algún impacto en cuanto al tiempo de ejecución en el programa.

Como se puede ver en la Tabla 4.1, la diferencia es prácticamente inexistente, por lo que podemos concluir que este parámetro no tiene ningún impacto en cuanto al tiempo de ejecución del programa. Para esta prueba se calculó la media de los tiempos de tres simulaciones con el parámetro log como True y otras tres como False.

Ejecución con 1 hilo			
Run	nº electrones	Sin Log	Con Log
0	1	0,001	0,001
1	2	0,005	0,004
2	3	0,007	0,007
3	5	0,005	0,005
4	7	0,014	0,014
5	11	0,058	0,066
6	14	0,072	0,077
7	20	0,098	0,108
8	32	0,135	0,131
9	50	0,152	0,140
10	75	0,383	0,425
11	108	0,454	0,483
12	168	0,639	0,681
13	251	1,093	1,136
14	375	1,582	1,614
15	567	2,189	2,236
16	857	3,738	3,786
17	1305	5,742	5,706
18	1964	8,783	8,556
Total	5815	25,150	25,176

Tabla 4.1: Impacto del parámetro *log* en tiempo de ejecución

4.1.3. Comprobar impacto del parámetro **plot** en el tiempo de ejecución

En este caso se quería hacer un análisis muy similar al anterior pero en este caso con el parámetro plot. En la Tabla 4.2 podemos observar que en este caso el impacto en el tiempo de ejecución de la variable plot es bastante grande.

Además, como se genera un archivo por cada electrón que se crea en la simulación, la escalabilidad empeorará mucho si se utiliza el programa con esta opción activada.

Ejecución con 1 hilo			
Run	nº electrones	Sin Plot	Con Plot
0	1	0,001	0,001
1	2	0,005	0,005
2	3	0,007	0,008
3	5	0,005	0,012
4	7	0,014	0,019
5	11	0,058	0,072
6	14	0,072	0,091
7	20	0,098	0,126
8	32	0,135	0,160
9	50	0,152	0,186
10	75	0,383	0,498
11	108	0,454	0,564
12	168	0,639	0,841
13	251	1,093	1,438
14	375	1,582	2,065
15	567	2,189	2,935
16	857	3,738	4,952
17	1305	5,742	7,571
18	1964	8,783	11,614
Total	5815	25,150	33,158

Tabla 4.2: Impacto del parámetro *plot* en tiempo de ejecución

4.2. Script para ejecutar el programa en ATLAS

Antes de comenzar con los cambios para mejorar los tiempos de ejecución del programa, uno de los aspectos más importantes del proyecto era preparar el programa para ejecutarse en el supercomputador ATLAS. Para esto, primero era necesario preparar un poco el entorno, por lo que se utilizó la herramienta Anaconda [1], creando un entorno con las librerías necesarias (principalmente, Fenicsproject) para poder utilizar el programa.

En el supercomputador ATLAS, se utiliza el sistema de administración de colas SLURM[2] para decidir el orden en el que se ejecutan los programas de los distintos usuarios del clúster en los nodos del sistema, por lo que a la hora de ejecutar el programa es necesario mandarlo a la cola de ejecución mediante el comando *sbatch*.

Desde ESS Bilbao se envió junto al código del programa un script a modo de ejemplo en el que se determinan los parámetros con los que se va a enviar a ejecutar el programa al

[1] https://www.anaconda.com/
[2] https://slurm.schedmd.com/documentation.html

clúster, pero era necesario modificarlo para este programa en concreto. En Listing 4.1 se puede observar un ejemplo de este script, en el que se lanza una ejecución con 4 cores de la pieza "pikachu_testbox_900k.mpc".

Listing 4.1: Ejemplo del script para mandar ejecuciones al supercomputador ATLAS.

```
#!/bin/bash
#SBATCH --partition=regular
#SBATCH --job-name=mpc_run
#SBATCH --cpus-per-task=4
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=32G
#SBATCH --time=1:00:00:00

#SBATCH --array=1-8

# modules
conda activate /scratch/jgalarza/conda-env/fenicsproject

BINDIR=$HOME/datos/fenix/bin
export DIJITSO_CACHE_DIR="./cache"

srun python mpc_dolfin_run_mp.py pikachu_testbox_900k.mpc $SLURM_ARRAY_TASK_ID
```

Para modificar la cantidad de cores con las que se pretende ejecutar el programa, basta con cambiar el valor del parámetro *–cpus-per-task*, donde el número máximo de cores será 56 (Aunque hay algunos nodos del supercomputador que solo tienen 48 cores). Debido a las características de la librería concurrent.futures utilizada para paralelizar el código y del propio lenguaje python, no se puede dividir la ejecución del programa en distintos nodos, por lo que la opción *–nodes* (Número de nodos con los que se va a ejecutar el programa) deberá ser siempre 1.

Por otro lado, el argumento $SLURM_ARRAY_TASK_ID que se utiliza al ejecutar el programa sirve para poder simular los electrones utilizando distintas semillas de números aleatorios, las cuales se tienen que incluir en el parámetro *–array*. De esta manera, ejecutando una sola vez el comando *sbatch* con el script Listing 4.1 por ejemplo, se pondrían a la cola 8 ejecuciones del programa, cada una de ellas con una semilla distinta del 1 al 8.

4.3. Versión MultiThreading del programa original

Tras preparar el programa para ejecutarse en el supercomputador ATLAS, se comenzó con uno de los objetivos principales del proyecto: Analizar el funcionamiento en paralelo del programa y comprobar posibles mejoras.

Cuando se utiliza un número diferente a 1 en el parámetro de entrada *parallel*, el funcionamiento de la función Run_n_electrons_parallel cambia. En este caso, se utiliza el módulo ThreadPoolExecutor de la librería Concurrent.Futures de python, y el pseudocódigo de esta función utilizándola en paralelo sería el siguiente (Listing 4.2):

Listing 4.2: Pseudocódigo de Run_n_electrons_parallel en la versión original en paralelo.

```
1 input: Power, Generations, Threads
2 output: Electrons_per_generation
3 begin
4   electron_list[] = generate_initial_electrons(Power)
5   for g in [0, generations]
6     new_electrons = ThreadPoolExecutor(Single_electron_trajectory(electron_list), workers=
      Threads)
7     electron_list = new_electrons
8   end
9 end
```

- **Run_n_electrons_parallel:** Se le pasan por parámetro el número de generaciones que se quieren calcular, la potencia y el número de hilos que se quiere utilizar.
  - Bucle principal, hasta que se llegue al número de generaciones indicado.
    - Se crean listas con los datos de todos electrones a simular (al comenzar será solo 1)
    - Mediante la librería se crea una "pool" de ejecuciones con todos electrones a simular y un "Ejecutor" por cada hilo con el que quiere ejecutar el programa (parámetro parallel)
    - Los "Ejecutores" ejecutan la función track_1_e con los datos de los electrones del "pool" en paralelo, hasta que no quede ninguno por simular.
    - Una vez simulados todos los electrones, se calcula la cantidad de electrones que se crean debido a las colisiones y se almacenan sus datos en nuevas listas para simularlos en la siguiente generación.

- Se almacena/muestra la información de la generación y, si no se ha llegado al máximo número de simulaciones indicado, continúa con el bucle.

- Una vez alcanzado el número máximo de generaciones a simular, la función devuelve el número total de electrones simulados y la cantidad de electrones generados en la última generación.

Para probar el funcionamiento en paralelo se simuló el programa tres veces con cada número de hilos para obtener tiempos medios de simulación. En la Tabla 4.3 se puede ver que al ejecutar el programa en paralelo no solo no se obtiene speed-up, sino que tarda bastante más tiempo en ejecutarse.

		Minutos ejecución			
Run	nº electrones	1 Hilo	4 Hilos	8 Hilos	32 Hilos
0	1	0,001	0,001	0,001	0,001
1	2	0,005	0,006	0,006	0,005
2	3	0,007	0,009	0,008	0,008
3	5	0,005	0,006	0,008	0,005
4	7	0,014	0,018	0,021	0,015
5	11	0,058	0,069	0,075	0,069
6	14	0,072	0,089	0,092	0,093
7	20	0,098	0,122	0,129	0,012
8	32	0,135	0,165	0,176	0,155
9	50	0,152	0,174	0,169	0,17
10	75	0,383	0,486	0,484	0,515
11	108	0,454	0,608	0,557	0,554
12	168	0,639	0,981	0,801	0,867
13	251	1,093	1,570	1,374	1,544
14	375	1,582	2,142	1,971	2,187
15	567	2,189	2,773	2,721	3,093
16	857	3,738	4,756	4,671	5,234
17	1305	5,742	7,829	7,046	7,669
18	1964	8,783	11,085	10,590	11,467
Total	5815	25,150	32,889	30,900	33,663

Tabla 4.3: Tiempos de ejecución en paralelo de la versión original del programa

Gracias al comando de linux *htop*, el cual es una versión mejorada del comando por defecto *top* para ver los procesos que se están ejecutando en el ordenador y el uso de los recursos del sistema por estos, se pudo comprobar que los hilos se creaban correctamente, pero, sin embargo, no se ejecutaban de manera simultánea en ningún momento.

Después de investigar un poco, se llegó a la conclusión de que esto era debido al GIL (Global Interpreter Lock) de Python. El GIL es una herramienta que utiliza Python para asegurar la atomicidad en sus programas, y el cual no permite que dos hilos generados por el mismo proceso accedan al intérprete de Python al mismo tiempo.

Debido a estas conclusiones, y tras valorar distintas alternativas para paralelizar el código, se decidió realizar una nueva versión del programa que en lugar de crear hilos distintos de un mismo proceso de Python, cree procesos distintos utilizando ProcessPoolExecutor. A esta nueva versión se le puso el nombre de “MultiProcessing” y en la Sección 4.4 se explica con mayor detalle.

4.4. Versión MultiProcessing

En el apartado anterior se ha podido comprobar que la versión original funcionaba peor en paralelo que en serie. Para entender mejor los cambios que se han realizado para mejorar este aspecto, primero es necesario entender la librería Concurrent.Futures de Python, que es la que se utiliza para crear en el programa distintos hilos de ejecución. Como se ha explicado en la Sección 3.4, esta librería contiene principalmente una clase llamada Executor y dos módulos: ThreadPoolExecutor y ProcessPoolExecutor.

La diferencia entre estos dos módulos es la clave para el funcionamiento en paralelo del programa, ya que en el programa se utiliza ThreadPoolExecutor, el cual crea lo que para Python son hilos distintos de un mismo programa (que se ven limitados debido al GIL de Python mencionado anteriormente). En cambio, el módulo ProcessPoolExecutor, tiene un funcionamiento similar, pero en vez de crear distintos hilos de un mismo proceso de python, crea procesos totalmente independientes. Sin embargo, si únicamente se cambia el módulo ThreadPoolExecutor por el ProcessPoolExecutor, este no funcionará, debido a que para este segundo módulo el programa no puede estar orientado a objetos.

Por esta razón, para poder implementar el módulo ProcessPoolExecutor, se decidió crear un nuevo archivo que sería una fusión entre los ficheros mpc_dolfin.py y mpc_run.py, el cual se llamó **“mpc_dolfin_run_mp”**. Esto era necesario, ya que si el programa no es orientado a objetos, no se puede crear una clase a la que se acceda desde un archivo externo como se hacía anteriormente, por lo que ahora la lectura de parámetros, las funciones de la librería para simular los electrones y las ejecuciones de estas mismas funciones, se encuentran en un mismo fichero.

Además de esto, se eliminó la clase "Multipacting"que estaba en mpc_dolfin.py y los argumentos de esta clase se convirtieron ahora en variables globales a las que se podrá acceder tanto para leer como para modificarlas desde las distintas funciones del programa. Una vez implementados todos estos cambios, se cambió la función que utilizaba el módulo ThreadPoolExecutor por el módulo ProcessPoolExecutor.

De esta manera, el pseudocódigo de esta versión, MultiProcessing, se puede ver en Listing 4.3 y es muy similar al de la versión original (Listing 4.2), con las únicas diferencias mencionadas de que ahora el programa ya no está orientado a objetos y que se utiliza el módulo ProcessPoolExecutor. Para este último cambio únicamente ha sido necesario modificar en el código el nombre del módulo a utilizar, debido a que la manera de utilizar ambos módulos de la librería es idéntica.

Listing 4.3: Pseudocódigo de Run_n_electrons_parallel en la versión MultiProcessing en paralelo.

```
1  input: Power, Generations, Threads
2  output: Electrons_per_generation
3  begin
4    electron_list[] = generate_initial_electrons(Power)
5    for g in [0, generations]
6      new_electrons = ThreadPoolExecutor(Single_electron_trajectory(electron_list), workers=
       Threads)
7      electron_list = new_electrons
8    end
9  end
```

4.5. Versión 2Pools

Después de ver los resultados obtenidos al ejecutar la versión MultiProcessing (Sección 5.1), la primera idea que surgió fue paralelizar la ejecución de las distintas potencias. Esto se debe a que analizando un poco el funcionamiento del programa, las generaciones de electrones mantienen dependencias entre sí: hasta que una generación no termina de simularse, no se sabe cuántos nuevos electrones se van a generar para la siguiente.

En cuanto a los electrones dentro de una misma generación, en este caso sí se pueden paralelizar debido a que no hay dependencias entre ellos. Con el caso de las potencias ocurre algo similar, dentro de cada potencia se mantiene la dependencia que hay entre las generaciones de electrones, pero entre los electrones de distintas potencias no existe ninguna dependencia.

Por esta razón, la decisión que se tomó fue realizar una paralelización muy similar a la realizada con los electrones, utilizando el mismo método y la misma librería (Concurrent.Futures), pero en este caso con las potencias. Por lo tanto, el programa tendría dos "pools" distintas, una de potencias y otra de electrones dentro de cada generación de cada potencia. Esta fue la razón por la que se le puso el nombre de 2Pools a esta nueva versión.

Por lo tanto, el pseudocódigo de esta versión quedaría de la siguiente manera (Listing 4.4):

Listing 4.4: Pseudocódigo de la versión 2Pools.

```
1 input: Generations, Power_range, Threads
2 output: Electrons_per_power
3 begin
4   Electrons_per_power = ProcessPoolExecutor(Run_n_electrons_parallel(Power_range, Generations),
      workers=Threads)
5 end
```

- **Run:**

  - Mediante la librería se crea una "pool" de ejecuciones con las distintas potencias a simular y un "Ejecutor" por cada core con el que se quiere ejecutar el programa (parámetro parallel). Estos ejecutarán una nueva función llamada run_1_power.
  - Dentro de esta función, el funcionamiento es muy similar al de la versión MultiProcessing: Bucle principal, hasta que se llegue al número de generaciones indicado.
    - Se crean listas con los datos de todos electrones a simular (al comenzar será solo 1).
    - Mediante la librería se crea una segunda "pool" de ejecuciones con todos los electrones a simular y un "Ejecutor" por cada core con el que quiere ejecutar el programa (parámetro parallel).
    - Los "Ejecutores" ejecutan la función track_1_e con los datos de los electrones del "pool" en paralelo, hasta que no quede ninguno por simular.
    - Una vez simulados todos los electrones, se calcula la cantidad de electrones que se crean debido a las colisiones y se almacenan sus datos en nuevas listas para simularlos en la siguiente generación.
    - Se almacena/muestra la información de la generación y, si no se ha llegado al máximo número de simulaciones indicado, continúa con el bucle.
  - Una vez alcanzado el número máximo de generaciones a simular, la función devuelve el número total de electrones simulados y la cantidad de electrones generados en la última generación de cada potencia.

Al implementar esta versión, en un principio no se tenía muy claro como iba a ser el funcionamiento del programa, debido a que realmente se están creando más procesos que el número de cores con el que se ejecuta el programa. Si en el parámetro parallel del programa ponemos, por ejemplo, un 8, a la hora de repartir la simulación en distintos procesos se están repartiendo tanto las potencias como los electrones de una generación dentro de cada potencia en 8 procesos cada uno.

Sin embargo, después de realizar algunas pruebas y analizar el funcionamiento en un ordenador personal con el anteriormente mencionado comando *htop* (ya que en el supercomputador ATLAS no es tan sencillo realizar este seguimiento), se comprobó que realmente lo que sucede al ejecutar esta versión del programa es que se crean muchos

más procesos que el número de cores con los que se van a ejecutar y es el propio Scheduling (sistema de reparto de tareas entre los procesadores) del sistema operativo el que se encarga de asignar los procesos creados a los cores que los van a ejecutar.

De hecho, al ejecutar el programa en ATLAS, el número máximo de cores que se pueden utilizar para ejecutarlo se determina en el script de lanzamiento del programa, por lo que nunca se puede usar una cantidad mayor que la asignada. Sin embargo, al ejecutar esta versión en un PC personal, se observó que si por ejemplo se utiliza un 2 en el parámetro parallel, al dividir cada "pool" de ejecución en 2 procesos distintos, a la hora de la verdad había 4 procesos ejecutándose al mismo tiempo, por lo que realmente los tiempos de ejecución no serían los correspondientes a una ejecución con 2 cores, ya que se estaban utilizando más que los determinados.

Obviamente, esto no debería ser aceptable y fue una de las principales razones por las que se decidió hacer más adelante otra versión, pero debido a que en el supercomputador el número de cores utilizados sí coincide con el asignado, también es interesante analizar los resultados obtenidos de esta versión en la que la cantidad de procesos generados es mayor que el número de cores con los que se van a ejecutar para comprobar si es mejor o peor que las otras.

4.6. Versión OnePool

En esta última versión, el objetivo principal es unir las dos "pools" de ejecución de la versión anterior en una sola. Por un lado, se pretende evitar la descompensación que surge en la versión 2Pools debido a que el reparto de ejecuciones en los distintos cores es bastante caótico, ya que dependiendo del número de potencias y electrones simulados, el reparto puede ser muy diferente. Por otro lado, también es necesario que el número de procesos creados coincida con el número de cores que se utilizan para ejecutar el programa, independientemente del sistema donde se ejecute el mismo.

Para tratar de lograr todo esto, es necesario cambiar el orden en el que se ejecutaban los distintos bucles del programa original (potencias, generaciones dentro de cada potencia y electrones dentro de cada generación).

Como ya se ha mencionado anteriormente, las únicas simulaciones que tienen dependencias entre sí son las de distintas generaciones de electrones, debido a que hasta que no termina una generación no se puede saber cuántos electrones hay que simular en la siguiente, o incluso si ya no hay que simular ninguna más. Sin embargo, tanto los electrones de distintas potencias y los electrones de una misma generación se pueden ejecutar en paralelo sin ningún problema.

Teniendo esto en cuenta, en Listing 4.5 tenemos el pseudocódigo de esta versión:

Listing 4.5: Pseudocódigo de la versión OnePool.

```
1
2   input: Generations, Power_range, Threads
3   output: Electrons_per_power
4   begin
5     for p in Power_range,
6       Electron_list_per_power[p]= generate_electrons(p)
7     end
8     electrons = Initial_electrons_per_power
9     for g in [0, Generations)
10      new_electrons = ProcessPoolExecutor(track_1_e(electrons),workers=Threads)
11      electrons = new_electrons
12    end
13  end
```

- **Run:**
  - Primer bucle, por cada potencia a simular:
    - Se calculan y almacenan los electrones iniciales de cada potencia
  - Una vez calculados todos los electrones iniciales, Bucle principal, por cada generación a simular:
    - Se crea una pool de ejecución con todos los electrones a simular de las distintas potencias en dicha generación. El número de ejecutores será igual al número de cores. La función ejecutada es track_1_e.
    - Se almacenan los resultados de la función y, si quedan generaciones a simular, continua con el bucle.
  - Una vez simuladas todas las generaciones, la función muestra el número total de electrones simulados en cada potencia.

De esta manera, ahora se ejecutan todos los electrones posibles en paralelo independientemente de que potencia sean, siendo el único limitante el número de generación al que pertenecen los electrones. Para realizar esta versión fue necesario cambiar una de las funciones originales del programa: Ahora la función track_1_e tiene un resultado más como salida, que es la potencia del electrón. Gracias a esto, se pueden crear las listas de los electrones a simular en las nuevas generaciones, diferenciando los electrones de distintas potencias, además de almacenar por separado las cantidades de electrones obtenidos en cada potencia a pesar de simularlos todos al mismo tiempo.

Además de esto, antes del bucle principal de generaciones, se calculan primero todos los electrones iniciales de las distintas potencias, en lugar de tener que calcular con cada una de ellas en tiempo de simulación la cantidad de electrones iniciales que surgen.

Como también puede ser interesante saber cuántos electrones se han generado con cada potencia por separado, se ha añadido la funcionalidad de mostrar dicha información por pantalla también, poniendo como “True” la variable debug del archivo .mpc, que anteriormente no tenía ninguna funcionalidad. También se ha comprobado que activar o desactivar el debug no tiene ningún impacto en el tiempo de ejecución del programa, debido a que únicamente se imprimen por pantalla algunas líneas más de información.

En la Figura 4.1 se puede ver la información mostrada por salida en una ejecución de la versión Onepool con debug como False y en la Figura 4.2 con debug como True.

```
Building point search tree to accelerate distance queries.
Computed bounding box tree with 38779 nodes for 19390 points.
Data file:  data/coaxial_103mm_704MHz_EB_field.txt
Mesh file;  data/coaxial_103mm_704MHz.mphtxt.h5
Data read. Num surface elems: 2458, Num volume elems; 19390

seed:1
cpus:4
Initial #electrons: 236
Completed secondary run 0, electrons alive: 5, time: 1.883 min
Completed secondary run 1, electrons alive: 8, time: 0.015 min
Completed secondary run 2, electrons alive: 9, time: 0.018 min
Completed secondary run 3, electrons alive: 11, time: 0.056 min
Completed secondary run 4, electrons alive: 8, time: 0.024 min
Completed secondary run 5, electrons alive: 9, time: 0.048 min
Completed secondary run 6, electrons alive: 6, time: 0.017 min
Completed secondary run 7, electrons alive: 7, time: 0.054 min
Completed secondary run 8, electrons alive: 7, time: 0.019 min
Total electrons: [70], Total time: 2.135 min
```

Figura 4.1: Datos de una ejecución sin debug.

```
Building point search tree to accelerate distance queries.
Computed bounding box tree with 38779 nodes for 19390 points.
Data file:  data/coaxial_103mm_704MHz_EB_field.txt
Mesh file;  data/coaxial_103mm_704MHz.mphtxt.h5
Data read. Num surface elems: 2458, Num volume elems; 19390

seed:1
cpus:4
Initial #electrons: 236
N of electrons generated with power 200000.0 = 59
N of electrons generated with power 400000.0 = 59
N of electrons generated with power 600000.0 = 59
N of electrons generated with power 800000.0 = 59
Completed secondary run 0, electrons alive: 5, time: 1.692 min
N of electrons generated with power 400000.0 = 1
N of electrons generated with power 600000.0 = 1
N of electrons generated with power 800000.0 = 3
Completed secondary run 1, electrons alive: 8, time: 0.009 min
N of electrons generated with power 400000.0 = 1
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 5
Completed secondary run 2, electrons alive: 9, time: 0.008 min
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 7
Completed secondary run 3, electrons alive: 11, time: 0.037 min
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 9
Completed secondary run 4, electrons alive: 8, time: 0.016 min
N of electrons generated with power 600000.0 = 1
N of electrons generated with power 800000.0 = 7
Completed secondary run 5, electrons alive: 9, time: 0.035 min
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 7
Completed secondary run 6, electrons alive: 6, time: 0.007 min
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 4
Completed secondary run 7, electrons alive: 7, time: 0.038 min
N of electrons generated with power 600000.0 = 2
N of electrons generated with power 800000.0 = 5
Completed secondary run 8, electrons alive: 7, time: 0.009 min
N of electrons generated with power 600000.0 = 1
N of electrons generated with power 800000.0 = 6
Total electrons: [70], Total time: 1.851 min
```

Figura 4.2: Datos de una ejecución con debug.

5. CAPÍTULO

Análisis de los resultados

Una vez explicadas las distintas versiones que se han implementado a lo largo del proyecto, es momento de ver las gráficas de las ejecuciones que se han realizado y los speed-up que se logran en cada versión respecto al programa original en serie.

Para ello, en este capítulo se van a analizar los resultados obtenidos con las distintas versiones desarrolladas: MultiProcessing, 2Pools y OnePool.

5.1. Entorno de experimentación

Para poner en contexto el entorno en el que se han ejecutado todas las simulaciones que quedan, vamos a ver primero las características del supercomputador ATLAS, los mallados de las piezas que se han utilizado para las simulaciones y los distintos parámetros utilizados para ejecutarlas.

- **Supercomputador Atlas**

  Atlas es el nombre del supercomputador que se encuentra en el recinto del DIPC, el cual se ha utilizado para ejecutar el programa de este proyecto. Este supercomputador está compuesto por dos sistemas distintos, ATLAS-FDR y ATLAS-EDR. En nuestro caso, para el TFG se ha utilizado el sistema ATLAS-EDR, el cual está compuesto por un total de 73 nodos. Los procesadores que se utilizan en estos nodos son Intel Xeon Platinum 8168, Intel Xeon Platinum 8280, Intel Xeon Platinum 8268 e Intel Xeon Gold 6248R. Cada nodo tiene 48 o 56 nodos y entre 64 y 384 GB de memoria RAM. También dispone de 2 NVIDIA RTX 3090 y 3 NVIDIA Tesla P40 como tarjetas gráficas y un sistema de memoria con un tamaño total de 1830 TB. El Sistema Operativo que se utiliza es un CentOS Linux 7 y para repartir el trabajo entre los nodos se utiliza el sistema de gestión de tareas SLURM. Más información sobre el supercomputador en la documentación oficial[1].

Durante las simulaciones, se han ejecutado cada una de las versiones de los distintos modos de ejecución con un total de 3 mallados de 2 piezas distintas, para comprobar los resultados obtenidos en contextos diferentes:

1. **Coaxial:** Es la primera pieza que se proporcionó desde ESS Bilbao junto al resto del programa a modo de ejemplo, y con la que se fueron realizando todas las pruebas de simulaciones a la hora de implementar las distintas versiones. El nombre completo del mallado de la pieza es “coaxial_103mm_704MHz_EB_field” y consta de 3885 nodos.

2. **Pikachu 500k y 900k:** Esta pieza consta de dos modelos de piezas reales de gran tamaño de mallado, que son unos acopladores de potencia de un dispositivo llamado RFQ. Este dispositivo de pruebas se llama "testbox". El acoplador es un modelo del prototipo diseñado por ESS Bilbao y que tiene el nombre en código de "Pikachu".

   Estos mallados fueron proporcionados por ESS Bilbao durante la parte final del proyecto, para poder probar las simulaciones en modelos de un tamaño mucho mayor y que además son de interés actual. El modelo pequeño tiene un mallado de 500k tetraedros y un total de 100.553 nodos, mientras que el modelo grande tiene 900k tetraedros y 177.897 nodos, por lo que la diferencia entre el mallado de coaxial y estos dos es enorme.

[1] Disponible en http://dipc.ehu.es/cc/computing_resources/

En la Figura 5.1 podemos ver un modelo sólido de la figura, mientras que en la Figura 5.2 se puede ver un análisis RF de la misma.

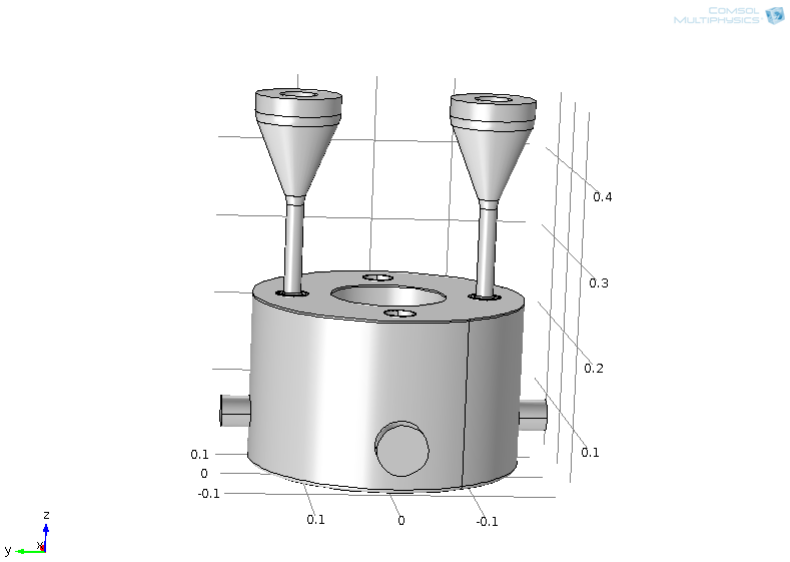

Figura 5.1: Modelo de la pieza "Pikachu testbox"1/2

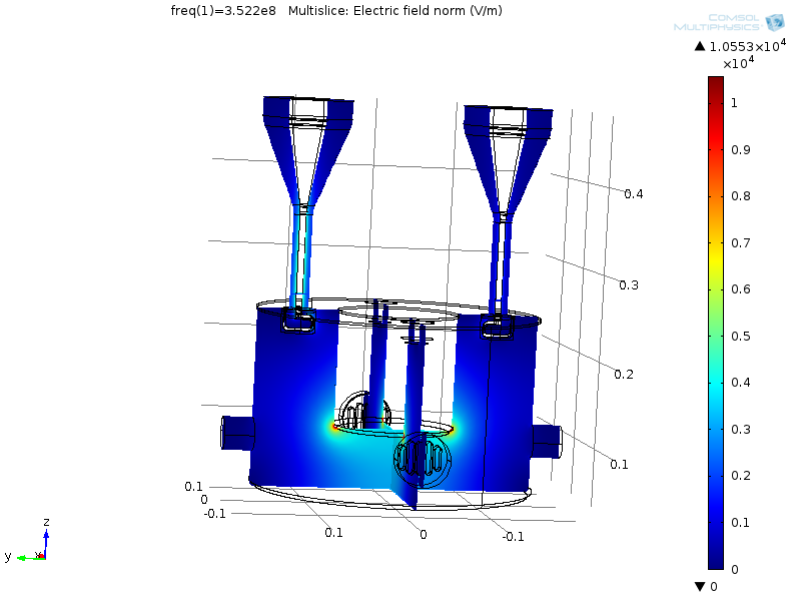

Figura 5.2: Modelo de la pieza "Pikachu testbox"2/2

A continuación, vamos a ver todas las simulaciones que se han analizado y los parámetros con los que se han ejecutado cada una de ellas. En este caso, para el modo de ejecución Single electron multipacting, se ha analizado únicamente la versión MultiProcessing y para el modo Power Range se han analizado las tres versiones implementadas. MultiProcessing, 2Pools y OnePool.

Además, en todas las versiones de un mismo modo de ejecución se han utilizado los mismos parámetros y semillas para poder comparar los tiempos de ejecución y speed-up de las mismas simulaciones.

Resumen de las simulaciones y sus parámetros:

- **Single electron multipacting:** Versión MultiProcessing (Sección 5.2)

Parámetro	**Valor**
Secondary_runs	10
Power	4e6
RF_Frequency	704.4e6
N_cycles	500
log	True
plot	False

Tabla 5.1: Parámetros simulaciones Single electron multipacting

- **Power range:** Versiones MultiProcessing (Sección 5.3.1), 2Pools (Sección 5.3.2) y OnePool (Sección 5.3.3)

Parámetro	**Valor**
Secondary_runs	8
Power	range(200e3,1000e3,200e3)
RF_Frequency	704.4e6
N_cycles	500
log	True
plot	False

Tabla 5.2: Parámetros simulaciones Power range

Debido a la gran cantidad de simulaciones que se realizaron para analizar las distintas versiones del programa, se decidió crear un breve script de bash para facilitar la recopilación de datos.

Este script lee la semilla y la cantidad de cores utilizados para cada ejecución y crea un archivo con terminación *.csv* con los datos de los tiempos totales de ejecución. De esa manera, se pueden importar los datos directamente en cualquier hoja de cálculo para crear una tabla de resultados.

5.2. Resultados Single electron multipacting

Para este modo de ejecución, se ha utilizado la media de los resultados de 8 semillas distintas y se han buscado las semillas adecuadas para que en cada uno de los mallados se genere multipacting y, por lo tanto, se simule un número grande de electrones. Esto es así para evitar que las simulaciones donde no ocurre multipacting, cuya ejecución es instantánea independientemente del número de procesos, no afecten el speed-up de los casos en que nos interesan, que son aquellos en los que sí ocurre multipacting. Para permitir la reproducibilidad de los experimentos, las semillas utilizadas han sido las siguientes:

- **Coaxial:** 128, 150, 152, 167, 24, 28, 44, 45.
- **Pikachu 500k:** 13, 152, 15, 161, 197, 37, 6, 70.
- **Pikachu 900k:** 119, 137, 15, 37, 4, 53, 77, 8.

Como se puede ver en la Figura 5.3 el funcionamiento en paralelo ahora es mucho más rápido y se logran unos speed-up máximos de 34 en la pieza Coaxial, 32 en Pikachu 500k y 30 en Pikachu 900k. La desviación típica aproximada es de 5 y en el mallado Coaxial el mejor tiempo de ejecución se obtiene utilizando 56 cores, mientras que en los mallados grandes pikachu500k y pikachu900k se obtienen mejores resultados utilizando menos cores.

Aun así, podemos decir que la escalabilidad es bastante buena, ya que en los mallados grandes Pikachu 500k y Pikachu 900k se siguen obteniendo speed-up similares a los que se obtienen con el mallado más liviano, a pesar de la gran diferencia de tamaño entre los mismos.

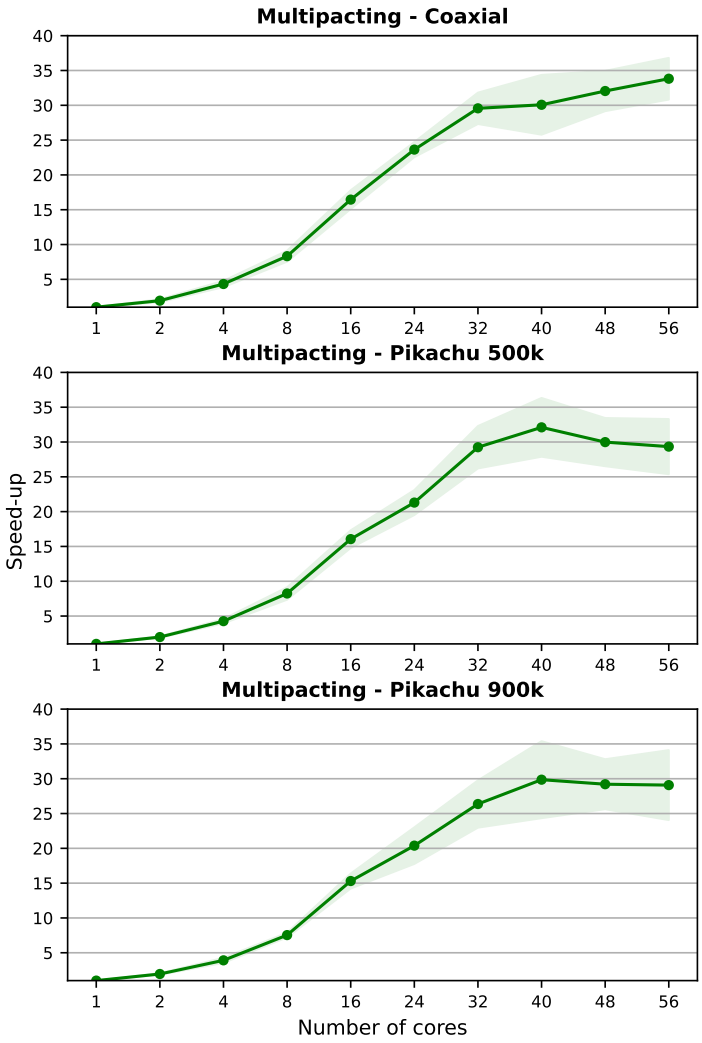

Figura 5.3: Speed-up con la versión MultiProcessing en el modo Single electron multipacting. La línea sólida representa la media y el área sombreada la desviación típica.

Es importante mencionar también que en el caso del supercomputador ATLAS, no todos los nodos tienen 56 cores, por lo que a la hora de lanzar el programa con este número tan alto de cores puede ocurrir que esté bastante tiempo en la cola de ejecución esperando a que se liberen los recursos necesarios para ejecutar el programa, lo que ha limitado el número de repeticiones que se han usado en estos experimentos.

También vemos como cuando el número de cores es muy alto, la relación número de cores/speed-up empeora un poco, lo cual es bastante lógico también, debido a que en las primeras generaciones (Run) hay pocos electrones y, por lo tanto, no se puede aprovechar bien la cantidad de cores del programa.

Sin embargo, cuando las cantidades de electrones son más grandes, que es cuando los

tiempos de ejecución son también mayores, sí se aprovechan bien todos los cores con los que se lanza el programa.

5.3. Resultados del modo Power range

Para asegurar que las comparaciones entre resultados sean realistas e imparciales, en todas las versiones del modo de ejecución Power range se han utilizado los mismos parámetros de entrada y las mismas semillas, por lo que la cantidad y el recorrido de todos los electrones es exactamente el mismo en todas las simulaciones.

En el modo de ejecución Power range, dependiendo de la cantidad de electrones generados por cada potencia, el comportamiento de distintas versiones puede ser muy diferente: En el modo de MultiProcessing, los electrones dentro de cada potencia se ejecutan en paralelo, pero las potencias se ejecutan en serie, por lo que cuando el número de electrones generado por cada potencia es bajo, no se puede aprovechar mucho el paralelismo.

Por esta razón, y porque al contrario que en el modo de ejecución Single electron multipacting, hay varios electrones iníciales por cada potencia, esta vez no se han buscado semillas con las que el número de electrones generado fuese grande, ya que se buscaba un comportamiento variado para obtener valores medios de simulaciones distintas.

En este caso, se han calculado las medias de los tiempos de ejecución de 7 semillas distintas. Únicamente se han descartado algunas semillas con las que la cantidad de electrones generados eran excesivamente grandes y no daban tiempo a que terminase debido a que tardaban varios días, por lo que las semillas utilizadas han sido las siguientes:

- **Coaxial, Pikachu 500k, Pikachu 900k:** 1, 2, 3, 5, 6, 7, 9.

5.3.1. Versión MultiProcessing

Como podemos ver en la Figura 5.4, en este caso el speed-up es mucho más bajo que en el modo de Single electron multipacting a pesar de utilizar la misma versión del programa. En las piezas Coaxial y Pikachu 500k se obtiene un speed-up máximo de 15 utilizando 48 y 32 cores respectivamente, mientras que en Pikachu 900k se logra llegar a casi 20 de speed-up con 32 cores.

Además, podemos ver también que la desviación típica es mucho mayor, llegando incluso a valores cercanos a 10 con números altos de cores, por lo que los resultados muestran

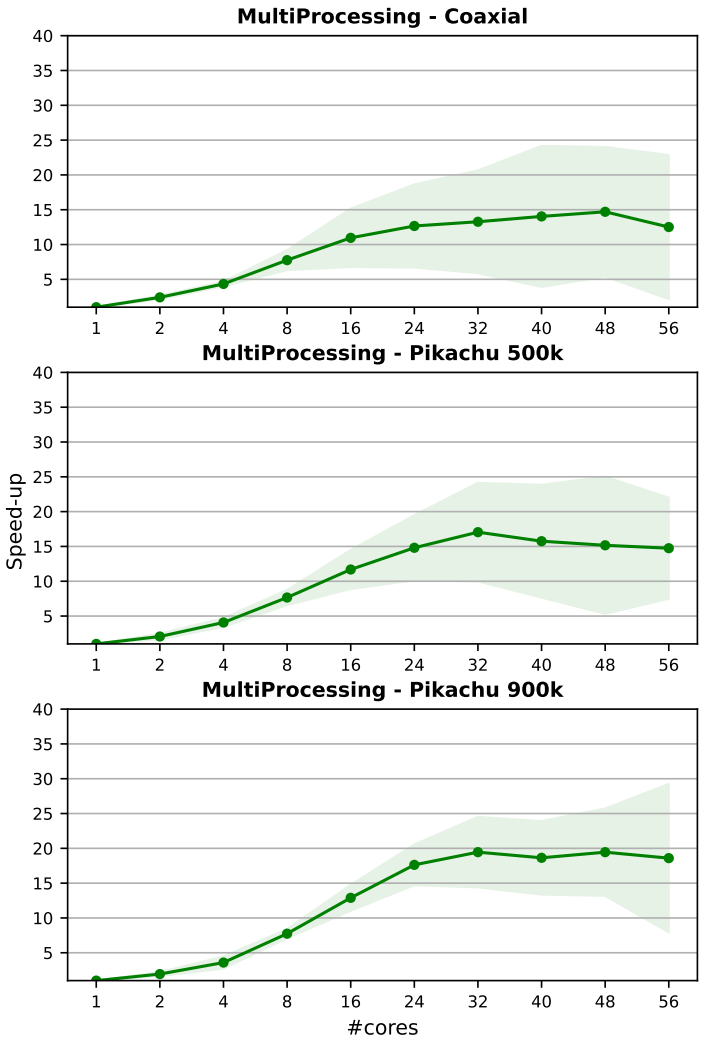

Figura 5.4: Speed-up con la versión MultiProcessing en el modo Power range. La línea sólida representa la media y el área sombreada la desviación típica.

gran variabilidad y justifican la necesidad de implementar una versión distinta para mejorar estos aspectos.

La razón principal de la desviación tan alta que se da en estas simulaciones es porque los tiempos de ejecución dependen mucho de los parámetros de entrada. Para entender bien esto vamos a comparar los resultados de dos tipos de simulaciones distintas que se realizaron, donde los parámetros de entrada eran distintos, de manera que, en el primero de los casos, se simulan pocas potencias distintas, pero se generan muchos electrones por cada potencia y en el segundo caso, se simulan muchas potencias distintas, pero en cada una de ellas la cantidad de electrones generados es pequeña.

Primera simulación: (power=range(500e3,3000e3,500e3), seed=2, s_runs=10)

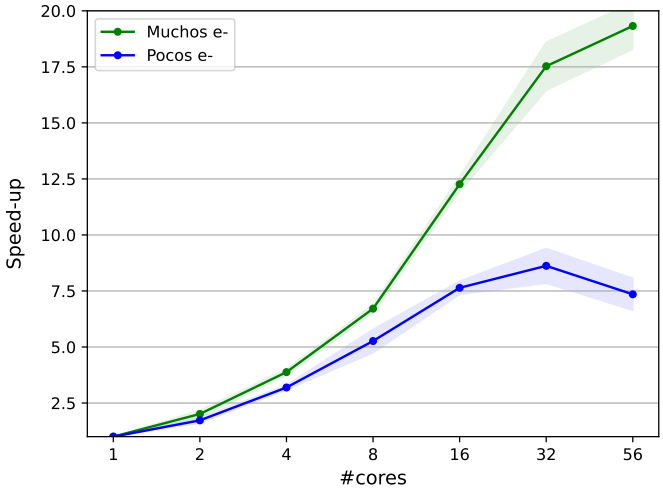

Figura 5.5: Speed-up cuando se generan muchos y pocos electrones. La línea sólida representa la media y el área sombreada la desviación típica.

En esta primera simulación, como se generan muchos electrones con todas las potencias, el speed-up es bastante bueno, llegando a un máximo de 19,3 con 56 cores (Figura 5.5 línea verde) aunque es peor que en el modo de ejecución de Single electron multipacting.

Segunda simulación: (power=range(50e3,1000e3,50e3), seed=1, s_runs=10)

En esta segunda simulación (Figura 5.5 línea azul), como se simulan muchas potencias distintas, pero en algunas no se generan grandes cantidades de electrones, el speed-up baja bastante y ni siquiera se logra obtener 10 de speed-up.

Esto es lógico, ya que en esta versión las potencias se ejecutan en serie y únicamente los electrones dentro de cada potencia se ejecutan en paralelo, por lo que vemos claramente que la eficiencia del programa al ejecutarse en paralelo y los speed-ups obtenidos varían mucho dependiendo del comportamiento del programa, lo cual no debería ser aceptable.

5.3.2. Versión 2Pools

En este caso, como podemos ver en la Figura 5.6, el speed-up llega a 25 en todas las piezas, por lo que la mejora frente a la versión MultiProcessing es muy considerable. Además de esto, la desviación típica vuelve a ser aproximadamente 5 como en el modo Single electron multipacting, que era uno de los principales problemas de la versión MultiProcessing en el modo Power range.

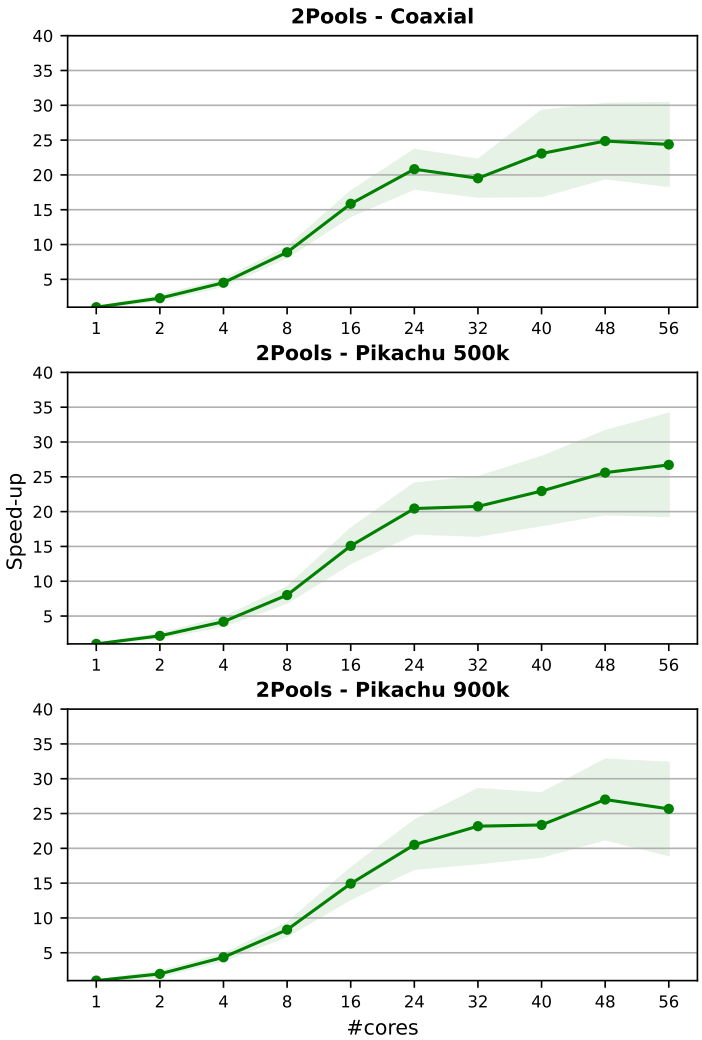

Figura 5.6: Speed-up con la versión 2Pools en el modo Power range. La línea sólida representa la media y el área sombreada la desviación típica.

En esta versión, la cantidad de procesos que se crean es mucho mayor al número de cores con los que se ejecuta el programa, y esto parecía que podría afectar negativamente al tiempo de ejecución del programa, sin embargo, a pesar de crear una cantidad tan grande de procesos, es el propio Scheduling del sistema operativo el que se encarga de repartir los procesos entre los cores disponibles y de esta manera la utilización de los mismos es muy alta.

Aun así, el speed-up máximo no se ha obtenido con el número más alto de cores, lo cual seguramente se debe a que en este caso la cantidad de procesos creados si está afectando negativamente al tiempo de ejecución, bien por un reparto desbalanceado, o por la sobrecarga de la gestión de procesos.

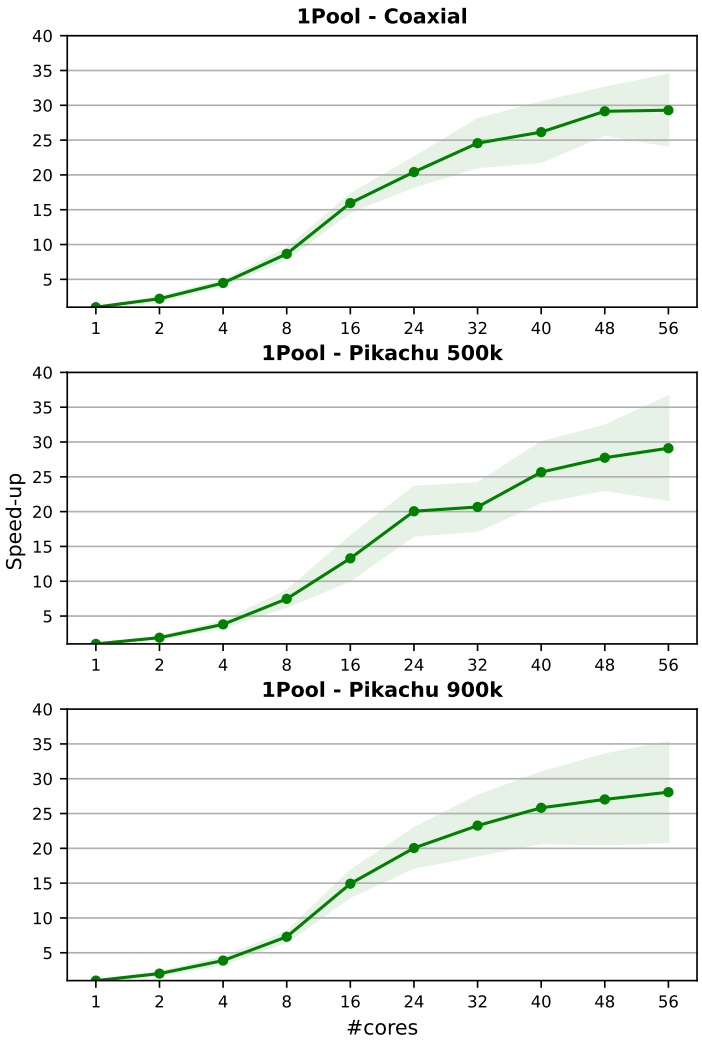

Figura 5.7: Speed-up con la versión OnePool en el modo Power range. La línea sólida representa la media y el área sombreada la desviación típica.

De todas formas, como se ha explicado en la Sección 4.5, si se ejecuta en otro lugar distinto al supercomputador se utilizarán más cores de los indicados, por lo que se llegó a la conclusión de que había una solución mejor aún, y se implementó la última versión, OnePool.

5.3.3. Versión OnePool

Como vemos en la Figura 5.7, en este caso sí logramos los mejores resultados de speed-up con el número máximo de cores, llegando prácticamente a 30 de speed-up en todas las piezas.

Esto se debe a que se genera una cantidad menor de procesos y se gestionan todos en una única "pool" de ejecuciones, por lo que a pesar de que en la versión 2Pools se obtenían unos resultados mejores de lo esperado, con esta versión hemos logrado mejorarlos un poco más aún. Además, la desviación es también ligeramente menor en todos los casos (antes la desviación era algo mayor que 5 y ahora algo mayor que 4), por lo que es también una versión con menor variabilidad.

De todas formas, al margen de los resultados, la manera de gestionar los electrones en esta versión es más ordenada y lógica, además de que el programa puede funcionar correctamente tanto en un supercomputador como en un PC personal, utilizando únicamente el número de cores que se indique por parámetro y con la opción añadida de activar o desactivar la opción debug para saber la cantidad de electrones generados por cada potencia.

Por lo tanto, coincidiendo con lo esperado, además de ser la más completa, los resultados nos indican que la versión OnePool es la mejor de las tres implementadas en cuanto a tiempos de ejecución.

6. CAPÍTULO

Conclusiones y trabajo futuro

A lo largo de este proyecto, se ha tratado de mejorar el rendimiento de un programa científico de ESS Bilbao de simulación de partículas, tratando de aprovechar lo máximo posible la capacidad de cómputo del supercomputador ATLAS del DIPC.

Teniendo en cuenta los objetivos marcados al inicio del proyecto y los resultados obtenidos, podemos considerar que se han cumplido de manera satisfactoria. Gracias a las modificaciones realizadas, ahora es posible ejecutar en unos minutos simulaciones que con la versión original tardarían varias horas, logrando speed-ups mayores que 30 en todas las piezas en el modo Single electron multipacting y speed-ups muy cercanos a 30 en el modo Power range.

Además, como ahora es posible ejecutar el programa en el supercomputador ATLAS, se pueden mandar a ejecutar en cuestión de segundos cientos de ejecuciones con distintas semillas para poder obtener una cantidad más amplia de datos y más rápidamente que con el programa original, recopilando también los resultados de una manera mucho más cómoda.

Cabe destacar que se ha escrito un artículo detallando las implementaciones y resultados obtenidos en este proyecto. El artículo ha sido aceptado para su presentación en las XXXII Jornadas de Paralelismo (JP2022) de las Jornadas Sarteco [1] (La Sociedad de Arquitectura y Tecnología de Computadores), que tendrán lugar del 21 al 23 de septiembre de 2022 en Alicante. Además, actualmente está en desarrollo otro artículo más detallado que se

[1] http://www.jornadassarteco.org/

enviará a la conferencia internacional SBAC-PAD [2], (The IEEE International Symposium on Computer Architecture and High Performance Computing), que se celebrará del 2 al 5 de noviembre de 2022 en Bordeaux, Francia.

A pesar de considerar que los resultados obtenidos son suficientemente buenos y que se han logrado cumplir todos los objetivos marcados a lo largo del desarrollo del proyecto, siempre puede haber margen de mejora, y más en el ámbito de la computación científica y el paralelismo.

La principal limitación que tiene el programa a la hora de paralelizar ejecuciones, son las generaciones de electrones, debido a que es necesario terminar la simulación de un electrón para saber si se generan o no más electrones y las características de los mismos.

Tal vez, si se cambiase un poco la lógica del programa y en lugar de asignar un número máximo de generaciones a simular, se estableciese por ejemplo un tiempo máximo de simulación o número máximo de electrones totales a simular, se podría tratar de hacer un programa similar. En ese caso, se podrían comenzar a simular los electrones de la siguiente generación mientras se simulan los últimos electrones de la generación actual, tratando de que no baje en ningún momento el porcentaje de utilización de los cores.

Sin embargo, esto es simplemente una idea y seguramente no se podría implementar utilizando la misma librería para paralelizar las ejecuciones, pero quizá utilizando alguna herramienta distinta para la paralelización del código podría llegar a implementarse.

Otro análisis que también podría ser interesante es intentar averiguar si el propio lenguaje de programación Python, que es interpretado, pero no compilado, tiene algún impacto en cuanto al tiempo de ejecución del programa. Normalmente, son otros lenguajes de programación como C/C++ los más utilizados para paralelizar código, por lo que podría ser interesante tratar de replicar el mismo programa, pero en alguno de estos dos lenguajes, y comprobar si realmente el lenguaje utilizado supone una mejora suficiente como para que merezca la pena realizar el cambio.

Si todavía intentásemos sacarle el máximo partido posible al programa después de pasarlo a C/C++, se podrían intentar implementar otras herramientas de paralelismo diferentes a la librería utilizada, como OpenMP o MPI, mediante lo cual es posible que se obtengan aún mejores tiempos de ejecución, pero es algo que no se puede saber hasta que no se pruebe.

[2] https://project.inria.fr/sbac2022/

Bibliografía

[con, 2022] (2022). Python standard library concurrent.futures documentation. https://docs.python.org/3/library/concurrent.futures.html.

[Alnæs et al., 2015] Alnæs, M., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., Ring, J., Rognes, M. E., and Wells, G. N. (2015). The fenics project version 1.5. *Archive of Numerical Software*, 3(100).

[Bacon, 2021] Bacon, J. W. (2021). Slurm workload manager. https://slurm.schedmd.com/documentation.html.

[Garoby et al., 2017] Garoby, R. et al. (2017). The european spallation source design. *Physica Scripta*, 93(1):014001.

[Palach, 2014] Palach, J. (2014). *Parallel programming with Python*. Packt Publishing Ltd.

[Pérez et al., 2020] Pérez, M., Sordo, F., Bustinduy, I., Muñoz, J. L., and Villacorta, F. J. (2020). Argitu compact accelerator neutron source: A unique infrastructure fostering r&d ecosystem in euskadi. *Neutron News*, 31(2-4):19–25.

[Vaughan, 1988] Vaughan, J. (1988). Multipactor. *IEEE Transactions on Electron Devices*, 35(7):1172–1180.