Universidad Nacional de La Plata

Facultad de Informática

Trabajo Final presentado para obtener el grado de Especialista en Cómputo de Altas Prestaciones y Tecnología GRID

Comparación de Rendimiento y Esfuerzo de Programación entre Rust y C en Arquitecturas Multicore. Caso de estudio: Simulación de N Cuerpos Computacionales

---

Autor:
Costanzo, Manuel

Director:
Naiouf, Marcelo
Co-Director:
Rucci, Enzo

Marzo 2021

Resumen

Históricamente, Fortran y C han sido los lenguajes de programación por defecto en la computación de alto rendimiento (*High-Performance Computing*, HPC). Ambos ofrecen al programador primitivas y funciones que permiten manipular la memoria del sistema e interactuar directamente con el hardware subyacente, resultando en un código eficiente tanto en tiempos de respuesta como en uso de recursos. Como contrapartida, resulta un verdadero desafío generar código que sea mantenible y escalable a lo largo del tiempo en estos tipos de lenguajes.

En el 2010 surge Rust, un nuevo lenguaje de programación diseñado para aplicaciones concurrentes y seguras, que adopta características de lenguajes procedurales, orientados a objetos y funcionales. Entre sus principios de diseño, Rust busca igualar a C en términos de eficiencia pero aumentado la seguridad y la productividad del código. Este trabajo presenta un estudio comparativo entre C y Rust en términos de rendimiento y esfuerzo de programación, seleccionando como caso de estudio la simulación de N cuerpos computacionales (N-Body), un problema popular en la comunidad HPC. A partir del trabajo experimental, fue posible determinar que Rust es un lenguaje que reduce el esfuerzo de programación, manteniendo rendimientos aceptables, posicionándolo como una posible alternativa a C para HPC.

Índice general

Capítulo 1

Introducción

En primer lugar, se presenta la motivación de este trabajo (Sección 1.1); luego, se mencionan los objetivos y la metodología a emplear (Sección 1.2) y por último, se explican las contribuciones del trabajo (Sección 1.3).

1.1. Motivación

La computación de alto rendimiento (*High-Performance Computing*, HPC) consiste en el uso de sistemas de extraordinario poder de cómputo y técnicas de procesamiento paralelo para la resolución de problemas complejos con alta demanda computacional [1]. Para lograr este objetivo, es necesario disponer no sólo de arquitecturas que otorguen la capacidad de procesamiento necesaria, sino también de software que permita computar de forma eficiente el problema. Es por esto que el lenguaje de programación a utilizar no es una elección trivial y su selección tendrá impacto tanto en la rendimiento de la aplicación como también el esfuerzo de programación requerido.

En general, los sistemas HPC deben computar el problema de manera eficiente en pos de mejorar los tiempos de respuestas del programa. Para lograr este objetivo, el lenguaje de base debe proveer primitivas y funciones que permitan aprovechar el hardware subyacente. Esto implica la posibilidad de generar múltiples tareas que se puedan ejecutar en paralelo y sean capaces de sincronizar y comunicarse entre sí, que exploten la localidad de los datos, que saquen ventaja de las unidades vectoriales de los procesadores, entre otras características.

En la actualidad, los lenguajes más usados en el área del HPC son Fortran y C, los cuales se caracterizan por ser lenguajes de bajo nivel de abstracción [1]. Esta clase de lenguajes permite al programador tener un control exhaustivo del

[1] Usualmente se los denomina simplemente *de bajo nivel*

programa y de los datos que procesa, lo que puede mejorar significativamente los tiempos de respuesta como también el uso de recursos de la aplicación [2].

A su vez, existen librerías que permiten extender la funcionalidad de los lenguajes base y así proveer capacidades de procesamiento concurrente y paralelo para arquitecturas multiprocesador tanto de memoria compartida como distribuida [3, 4, 5].

Con el paso del tiempo, tanto Fortran como C se han establecido como los lenguajes por defecto en HPC, siendo ampliamente aceptados por la comunidad. Aun así, resulta un desafío muy grande generar un código que sea mantenible y escalable a lo largo del tiempo en estos lenguajes; situación que no sucede con lenguajes de alto nivel de abstracción [2], como puede ser Python o Java. Por ejemplo, en los lenguajes a bajo nivel, es responsabilidad del programador evitar comportamientos indefinidos (del inglés, *undefined behaviour*) debido a que en muchas ocasiones el compilador no los podrá detectar [6]. En ese sentido, al aumentar el tamaño del sistema, se incrementa la posibilidad de que ocurran errores en memoria en tiempos de ejecución, lo que lleva al software a ser inestable y propenso a errores.

En los últimos años, muchos lenguajes de programación han tratado de incorporar soporte para concurrencia y paralelismo, de forma tal de extender su funcionalidad y poder utilizarse en áreas diversas. Entre los lenguajes más populares que han tratado de imponerse como un competidor para C y Fortran se encuentran los lenguajes Java y Python.

Con respecto a Java, en [7] los autores lo comparan con C y Fortran en ambientes de memoria compartida y distribuida, concluyendo que tiene rendimientos comparables con los lenguajes nativos, pero que éste aporta un overhead propio de los lenguajes de alto nivel que no existen en los otros. En [8] los autores comparan Fortran, C++ y Java en una aplicación orientada a objetos y evalúan no sólo la eficiencia de la aplicación, sino también la facilidad de programación en estos lenguajes. En ese sentido, Fortran obtuvo resultados comparables a los otros dos lenguajes orientados a objetos, pero el esfuerzo de programación requerido fue mayor.

En cuanto a los estudios sobre Python, en [9] se traduce un sistema optimizado para C al lenguaje Python. Si bien los autores afirman que el código Python generado es mucho más legible en comparación al de C, el primero agrega overheads que afectan de manera negativa el rendimiento de la aplicación. En la misma línea se encuentra el estudio [10], donde los autores realizan una comparación entre el lenguaje Python y C para la implementación de metaheurísticas paralelas. El mayor obstáculo de Python consiste en la inmadurez en las librerías multi-hilo, en comparación con la de OpenMP en C, lo que lleva

[2] También conocidos simplemente como *de alto nivel*.

a tener rendimientos inferiores y menos escalables.

A pesar de los esfuerzos de las comunidades de Python y de Java como también de otros lenguajes menos populares [11], lamentablemente ninguno ha podido establecerse satisfactoriamente como alternativa en la comunidad de HPC por el momento.

En el año 2010, los investigadores de Mozilla hacen público el lenguaje Rust [12], el cual fue diseñado para el desarrollo de sistemas altamente concurrentes y seguros. Rust es un lenguaje compilado que realiza verificaciones de seguridad en los accesos a memoria en tiempo de compilación. También posee un modelo de propiedad de datos que evita los problemas de condiciones de carrera en forma previa a la ejecución. De acuerdo a sus autores, Rust permite tener un código legible y escalable sin afectar en el rendimiento.

Por su filosofía, Rust se posiciona como un competidor directo de C y C++, pero con un fuerte enfoque en la seguridad de los accesos a la memoria [13]. Desde el punto de vista de la programabilidad, combina elementos de lenguajes de programación procedurales con otros de programación orientada a objetos y funcional [14]. El objetivo principal de Rust es proveer al programador de un control del sistema equivalente a C, pero sin perder de vista la seguridad con el fin evitar los problemas de comportamientos indefinidos mencionados anteriormente [15]. Además, soporta acceso directo al hardware y control de grano fino sobre las representaciones de la memoria, lo que permite optimizar los algoritmos de manera específica [16].

En los últimos años, Rust ha incrementado su popularidad y uso en diferentes ámbitos a tal punto que en el año 2019 se consagró como el lenguaje de programación más "amado" por cuarto año consecutivo en Stack Overflow [17]. Asimismo, varias empresas continúan migrando sus sistemas a Rust, entre las que se destacan: Facebook, que ha migrado en forma parcial su sistema de control a Rust [18]; Mozilla, que está desarrollando su motor de navegador Servo en Rust [19]; Discord, que está migrando su sistema por completo del lenguaje Go a Rust [20]; Dropbox, que usa Rust para el manejo de hilos y concurrencia en sus sistemas [21]; NPM, que ha migrado a Rust para eliminar cuellos de botellas en el rendimiento [22]; entre otras [23].

Como se mencionó al comienzo, resulta indispensable conocer las ventajas y desventajas de cada lenguaje de programación para implementar sistemas HPC, así como las sentencias y directivas en cada lenguaje para obtener aplicaciones de alto rendimiento. Por sus características, Rust se presenta como una opción prometedora de alternativa a los lenguajes tradicionales. De manera de contribuir a la evaluación de Rust como alternativa de lenguaje base para aplicaciones de procesamiento paralelo, este trabajo se enfoca en su comparación con C en términos de rendimiento y esfuerzo de programación. Para ello,

se selecciona como caso de estudio la simulación de N cuerpos computacionales (N-Body), un problema con alta demanda computacional y que resulta popular en la comunidad HPC. Mediante este estudio comparativo se espera contribuir a la identificación de las fortalezas y debilidades de Rust para su uso en el ámbito de HPC.

1.2. Objetivo y metodología

El objetivo general de esta investigación es comparar las prestaciones (rendimiento y esfuerzo de programación) de los lenguajes C y Rust para soluciones al problema de N-Body en arquitecturas multicore.

Los objetivos específicos son los siguientes:

- Explorar soluciones a N-Body en arquitecturas multicore que empleen los lenguajes C y Rust.
- Analizar las prestaciones de implementaciones de N-Body en arquitecturas multicore que hayan sido desarrolladas usando los lenguajes C y Rust.
- Comparar las prestaciones de los lenguajes C y Rust para soluciones a N-Body en arquitecturas multicore.

Para poder cumplir con los diferentes objetivos se han realizado las siguientes actividades:

- Se estudió el problema N-Body y sus requisitos computacionales.
- Se relevó la bibliografía existente en la temática a partir de la búsqueda en bases de datos especializadas.
- Se diseñaron y desarrollaron diferentes soluciones paralelas en Rust al problema estudiado, considerando distintas optimizaciones aplicables.
- Se midió el rendimiento y el esfuerzo de programación entre ambas soluciones y se realizó un análisis comparativo.
- Se analizaron los resultados para determinar fortalezas y debilidades del lenguaje Rust para aplicaciones que requieran cómputo intensivo.

1.3. Contribuciones

Entre las contribuciones de este trabajo, se pueden mencionar:

- Una implementación optimizada en el lenguaje Rust [3] que compute N-Body sobre arquitecturas multicore, la cual se dispone en un repositorio web público para beneficio de la comunidad científica [4].
- Una comparación rigurosa de las soluciones en Rust y en C para N-Body en arquitecturas multicore considerando el rendimiento y esfuerzo de programación. Este estudio comparativo contribuirá en la evaluación del potencial de Rust como un lenguaje de base alternativo en el ámbito de HPC.

[3] Para beneficio de la comunidad científica, los algoritmos se encuentran disponibles en un repositorio web público: `https://github.com/ManuelCostanzo/Gravitational_N_Bodies_Rust`.

[4] Para el estudio comparativo, se dispone de una solución optimizada para el mismo problema en el lenguaje C [24]

Capítulo 2

Antecedentes

En primer lugar, se realiza una breve introducción sobre los lenguajes de programación (2.1); luego, se mencionan las características de los lenguajes de programación tradicionales para HPC (2.2); posteriormente se introduce sobre la historia, las características y particularidades tanto del lenguaje C (2.2.1), como de Rust (2.3); y finalmente, se explica el funcionamiento del algoritmo de N-Body (2.4).

2.1. Lenguajes de programación

Un lenguaje de programación es un lenguaje artificial diseñado para expresar cálculos, que pueden ser realizados por una máquina, particularmente por una computadora [25].

A partir de los años 50, los lenguajes de programación se han desarrollado muy rápidamente, lo que ha llevado a la creación de cientos de diferentes lenguajes. Con el constante avance tecnológico, el desafío se centra tanto en la creación de nuevos lenguajes que aporten herramientas novedosas, para mejorar los tiempos de respuesta y la facilidad de programación, pero también en la adaptación de lenguajes que ya fueron creados, mejorándolos conforme las tecnologías evolucionan [26].

Debido a que existen una gran variedad de lenguajes, la elección del mismo no resulta trivial y dependerá del problema a resolver. El lenguaje de programación suele impactar tanto positiva o negativamente sobre el programador; incrementando la complejidad y el esfuerzo al momento de codificar, como también la dificultad de entender el código escrito por otros desarrolladores. Además, impacta en el mantenimiento y la escalabilidad del programa. De la misma forma, el lenguaje también tendrá su impacto internamente en el sistema, afectando en los tiempos de respuesta y en el uso adecuado de los

recursos.

2.2. Lenguajes de programación tradicionales en HPC

Al momento de elegir un lenguaje de programación en el área de HPC, usualmente el foco se centra en el impacto que tendrá en el sistema. Es por esto que históricamente, Fortran y C han sido los lenguajes de programación base para HPC. Estos tipos de lenguajes deben aportar herramientas que posibiliten el desarrollo eficiente de la solución. En ese sentido, el lenguaje debe proveer al menos las siguientes características:

- Ofrecer punteros que permitan manipular directamente los contenidos de la memoria.
- Poseer un conjunto de operadores de manipulación de bits, que usados adecuadamente pueden mejorar los tiempos del programa en forma considerable.
- Admitir diferentes flags u opciones de compilación que optimizan un código de acuerdo a la arquitectura de soporte.
- Permitir utilizar/embeber instrucciones nativas de la arquitectura subyacente para aprovechar sus características hardware.
- Entre otras.

Además, con el objetivo de mejorar los tiempos de respuesta, es necesario que el lenguaje ofrezca herramientas o librerías que permitan extender la funcionalidad de los lenguajes base y así proveer capacidades de procesamiento concurrente y paralelo para arquitecturas multiprocesador, tanto de memoria compartida como distribuida.

Por un lado, OpenMP [3] se ha convertido en el estándar más popular para aplicaciones multi-hiladas, la posibilidad de alcanzar un rendimiento alto a un bajo costo de programación, entre otros beneficios. Por otro lado, OpenMPI [4] y MPICH [5] son dos de las librerías que implementa el estándar MPI (*Message Passing Interface*), cuyo principal objetivo es proveer una interfaz estándar para el desarrollo de aplicaciones basadas en pasaje de mensajes.

2.2.1. Lenguaje de programación C

C es un lenguaje de programación creado en 1972 por Dennis M. Ritchie en los Laboratorios Bell para escribir el código del sistema operativo UNIX [27]. Entre las características más importantes se puede mencionar:

- Es un lenguaje de propósito general.
- Es un lenguaje estructurado.
- Es un lenguaje de bajo nivel.
- Es un lenguaje tipado.
- No depende del hardware, por lo que es posible migrar código C de una arquitectura a otra.
- Ofrece al programador el control absoluto del sistema.

C es elegido por la comunidad HPC principalmente debido a que es un lenguaje potente y eficiente, que proporciona un completo control de los recursos del sistema. Además, es compatible con las librerías de programación paralela más usadas en la actualidad, tanto para arquitecturas de memoria compartida como de memoria distribuida.

Debido a estas características es que grandes paquetes de software, utilizan C/C++ como lenguaje base, como pueden ser: LAMMPS [28], un simulador paralelo masivo atómico/molecular a gran escala; GROMACS [29], un programa para la simulación de la dinámica molecular de sistemas con cientos de millones de partículas; o SETI@Home [30], un proyecto que analiza señales de radio buscando signos de inteligencia extraterrestre.

2.3. Rust

En esta sección primero se realiza una introducción en cuanto al origen y la actualidad de Rust (Sección 2.3.1), luego se mencionan las características principales del lenguaje (Sección 2.3.2) y por último se explica las capacidades de Rust para el cómputo concurrente y paralelo (Sección 2.3.3).

2.3.1. Orígenes y actualidad

Rust es un lenguaje de programación que surgió en 2010 en la empresa Mozilla como un proyecto personal de Graydon Hoare, en busca de un lenguaje

que permitiera escribir un código extremadamente veloz, al mismo nivel que C, pero sin los problemas de gestión de memoria que llevaron históricamente a errores en los accesos y en las carreras de datos [1].

Es por esto que Rust surge como un competidor directo de C y C++, pero con un fuerte enfoque en la seguridad de los accesos a la memoria [13]. Desde el punto de vista de la programabilidad, combina elementos de lenguajes de programación procedurales con otros de programación orientada a objetos y funcional [14]. El objetivo principal de Rust es proveer al programador de un control del sistema equivalente a C, pero sin perder de vista la seguridad con el fin de evitar los problemas de comportamientos indefinidos mencionados en la secciones anteriores [15]. Además, soporta acceso directo al hardware y control de grano fino sobre las representaciones de la memoria, lo que permite optimizar los algoritmos de manera específica [16].

2.3.2. Características principales

Entre las características principales de Rust se destacan: la no necesidad de un *Garbage Collector* (recolector de basura), el concepto de *ownership* y el concepto de *borrowing*.

Al igual que C, Rust no posee *Garbage Collector*, pero rara vez es necesario liberar la memoria manualmente. Con este objetivo, Rust emplea el modelo de *ownership* (propiedad) integrado en su sistema de tipos para la gestión de la memoria. En particular, este concepto se basa en 3 reglas principales:

1. Cada valor en Rust tiene una variable que se llama propietario.
2. Solo puede haber un propietario a la vez.
3. Cuando el propietario sale del ámbito de la aplicación, el valor se elimina.

Este concepto ha ido ganando popularidad entre los académicos así como entre los principales desarrolladores de lenguajes y es una característica única de Rust [16].

Otro concepto importante es el de referencia y *borrowing* (préstamo). La referencia permite obtener un valor sin tomar el *ownership* del mismo, mientras que el *borrowing* es una consecuencia de la referencia: cuando las funciones tienen referencias como parámetros, no se debe devolver el *ownership* porque nunca se lo tuvo. Las reglas de referencias son las siguientes:

[1] La carrera de datos (del inglés *data race*) es una condición que ocurre cuando 2 o más hilos acceden a una variable compartida/global y al menos uno de los hilos la escribe.

- En cualquier momento, se puede tener una referencia mutable o cualquier número de referencias inmutables.
- Las referencias deben ser siempre válidas.

2.3.3. Concurrencia y paralelismo

Rust fue diseñado con el foco en la concurrencia. El concepto de *ownership* no solo elimina los errores comunes en memoria sino que también evita muchos errores de condiciones de carrera. En C, estos errores normalmente son difíciles de detectar, debido a que ocurren bajo circunstancias específicas que son difíciles de reproducir. Gracias a los conceptos de *ownership* y *borrowing*, Rust detecta muchos de estos errores (pero no todos) en el momento de la compilación, por lo que resulta un lenguaje muy adecuado para problemas que requieran concurrencia y paralelismo [12].

Rust asegura que las actualizaciones en la memoria se realicen atómicamente y que todos los hilos verán el valor actualizado mediante lo que se denomina referencias mutables [31]. Además, provee muchas funcionalidades que ayudan al programador a desarrollar código concurrente y paralelo de forma segura y rápida. Su filosofía es detectar el mayor número de errores durante la compilación, lo que garantiza que cuando un programa en Rust compila, el programador tiene la certeza de que es seguro en memoria y libre de carrera de datos [12].

Rust posee una librería específica para el paralelismo de datos llamada *Rayon* [32]. Rayon busca convertir código secuencial de Rust en un código paralelo de forma simple, garantizando la ausencia de carrera de datos. Rayon proporciona su propia implementación para la creación de hilos, lo que permite que el programador no tenga que encargarse de esto y mejora la eficiencia al momento de la creación y asignación de tareas.

El mecanismo central por el cual la librería define las tareas que pueden correr en paralelo es el *join*. Rayon intentará crear los hilos en paralelo, pero en tiempo de ejecución verifica si los núcleos están disponibles; en caso de que no haya núcleos disponibles, lo ejecuta en forma secuencial. La idea es similar a la de las regiones paralelas de OpenMP que residen en el código fuente. Sin embargo, en vez de ejecutar siempre el código seleccionado en paralelo, Rayon dinámicamente decide si tiene sentido ejecutar en paralelo dependiendo de los recursos del sistema en ese momento [33].

Adicionalmente, Rayon provee iteradores que recorren las estructuras de datos y realizan acciones en paralelo. Un punto importante a aclarar es que Rayon no sólo asigna a cada hilo una cantidad igual de datos y espera hasta que todos los hilos estén terminados, sino que también utiliza una técnica conocida

como *work stealing* (robo de trabajo). Básicamente, cada hilo tiene su propia cola de trabajos pendientes, procesándolas hasta que la cola se vacíe. Si un hilo termina con sus tareas, le "roba" tareas de la cola de otro hilo. Esto evita que haya hilos ociosos y ayuda a finalizar antes con el trabajo.

Si bien Rayon busca ejecutar la mayor cantidad de hilos eficientemente, también permite definir la cantidad de hilos manualmente, como el orden de ejecución de los mismos.

En cuanto al modelo de pasajes de mensajes, Rust también provee soporte a través de una variedad de librerías, donde la principal es *mpsc* (de las siglas *Multiple Producer Single Consumer*). Para enviar mensajes entre hilos, una de las principales herramientas que tiene Rust es el canal (del inglés, *channel*). Un canal en programación puede verse como un canal de agua, en donde si se inserta algo al inicio del canal, navegará por el mismo hasta la salida [34]. Sin embargo, su descripción queda fuera del alcance del presente trabajo.

2.4. Simulación de N cuerpos Computacionales con Atracción Gravitacional

El problema consiste en simular la evolución de un sistema compuesto por N cuerpos durante una cantidad de tiempo determinada. Todo cuerpo cuenta con un estado inicial dado por su velocidad y posición, mientras que el movimiento del sistema se simula a través de instantes discretos de tiempo. En cada uno de estos instantes, el estado de un cuerpo se ve afectado debido a la aceleración que sufre por efecto de la atracción gravitacional que se da entre todos los cuerpos.

La física que subyace a la simulación se basa en la mecánica Newtoniana. Esta simulación se realiza en 3 dimensiones espaciales y la atracción gravitacional entre dos cuerpos C_i y C_j se calcula mediante la ley de gravitación universal de Newton:

$$F = \frac{G \times m_i \times m_j}{r^2}$$

donde F se corresponde con la magnitud de la fuerza gravitacional entre ambos cuerpos, G es la constante de gravitación universal, m_i y m_j corresponden a las masas de los cuerpos C_i y C_j respectivamente y r corresponde a la distancia Euclídea entre los cuerpos C_i y C_j.

Cuando $N > 2$, la fuerza de gravitación sobre un cuerpo se obtiene sumando todas las fuerzas de gravitación ejercidas por los *N-1* cuerpos restantes.

Esta fuerza de atracción provoca una aceleración del cuerpo mediante la aplicación de la segunda ley de Newton:

$$F = m \times a$$

donde F es el vector fuerza, que se calcula mediante la ecuación de gravitación y el sentido y dirección del vector desde el cuerpo afectado hacia el cuerpo que ejerce la atracción.

Despejando la ecuación previa, se puede calcular la aceleración de un cuerpo determinado mediante la división de la fuerza total por su masa. Durante un intervalo de tiempo pequeño dt, la aceleración a_i del cuerpo C_i es prácticamente una constante, por lo que el cambio en velocidad se aproxima a:

$$dv_i = a_i dt$$

La modificación de la posición de un cuerpo es igual a la integral de su velocidad y aceleración sobre el intervalo de tiempo dt, que se aproxima a:

$$dp_i = v_i dt + \frac{a_i}{2} dt^2 = (v_i + \frac{dv_i}{2}) dt$$

Esta fórmula utiliza el esquema de integración Leapfrog [35], en donde una mitad del cambio de posición emplea la velocidad vieja mientras que la otra considera la velocidad nueva.

En la Figura 2.1 se puede observar un pseudocódigo de la implementación directa de N-Body.

```
1  Para cada cuerpo de i = 1 a N:
2      Para cada cuerpo de j = 1 a N:
3          Calcular la fuerza que ejerce j sobre i.
4          Sumar las fuerzas que afectan a i.
5      Calcular el desplazamiento del cuerpo i.
6      Mover el cuerpo i.
```

Figura 2.1: Pseudocódigo de la implementación del problema de N-Body

En este pseudocódigo se pueden detectar dos diferentes dependencias de datos. La primera dependencia radica en que antes de desplazar los cuerpos, se deben computar las fuerzas del paso anterior. La segunda dependencia consiste en que el cálculo de las fuerzas de una iteración, depende de la iteración anterior.

Estas dependencias deberán ser tenidas en cuenta al momento de aplicar optimizaciones al algoritmo de N-Body base.

2.4.1. Implementación en C

Para realizar una comparación justa entre C y Rust se utilizó un algoritmo optimizado para arquitecturas multicore presentado en [24]. El algoritmo posee las siguientes características:

- Multihilado: a través de directivas OpenMP, realiza en paralelo los cálculos de fuerza por separado del que mueve los cuerpos. Para esto, utiliza la directiva *for* con las cláusulas *static*, para distribuir equitativamente la cantidad de cuerpos entre los hilos.
- Optimizaciones escalares: Se realizan optimizaciones para reducir el costo computacional de operaciones simples que se ejecutan varias veces:
  - Para efectuar una potencia, utiliza la función *powf* cuando el tipo de dato es float y *pow* cuando el tipo de dato es double.
  - Especifica las constantes como flotante en el caso que corresponda.
  - Reemplaza las divisiones que tienen una constante como denominador por multiplicaciones con el inverso multiplicativo de dicha constante.
- Vectorización: la vectorización permite realizar una misma operación sobre múltiples datos en paralelo, reduciendo así el tiempo de ejecución requerido por las operaciones que pueden ser vectorizadas. Utiliza la directiva simd de OpenMP 4.0 que garantiza la vectorización del código.

  in order to exploit data locality, bodies are processed using a blocking technique. To achieve this, bodies are divided in fixed-size portions called blocks. The bodies loop is replaced by two others: a loop that iterates over all blocks and an inner loop that iterates over the bodies of each block
- Procesamiento por bloques: con el fin de explotar la localidad de datos, se implementa un procesamiento por bloques de los cuerpos. Para lograr esto, el total de cuerpos se dividen en porciones de tamaño fijo denominados *bloques*. El bucle de los cuerpos es reemplazado por dos diferentes: un primer bucle que itera sobre todos los bloques de cuerpos y un bucle interno que itera sobre los cuerpos de ese bloque.
- Desenrrollado de bucles: es una técnica de optimización que puede mejorar el rendimiento de un programa, desenrollando las iteraciones de bucles en operaciones individuales. Esto puede hacerse manualmente por el programador, o automáticamente por el compilador mediante la directiva *unroll*.

En la Figura 2.2 se puede observar el código en C utilizado.

```
  for (t = 1; t <= D; t += DT)
  {

#pragma omp parallel for schedule(static) private(i, j)
    for (ii = 0; ii < N; ii += BLOCKSIZE)
    {

      __declspec(align(ALIGNMENT)) DATATYPE forcesx[BLOCKSIZE] = {0.0};
      __declspec(align(ALIGNMENT)) DATATYPE forcesy[BLOCKSIZE] = {0.0};
      __declspec(align(ALIGNMENT)) DATATYPE forcesz[BLOCKSIZE] = {0.0};

#pragma unroll(BLOCKSIZE)
      for (j = 0; j < N; j++)
      {
        PRINT2
#pragma omp simd aligned(xpos, ypos, zpos, forcesx, forcesy, forcesz)
        for (i = ii; i < ii + BLOCKSIZE; i++)
        {
          const DATATYPE dx = xpos[j] - xpos[i];
          const DATATYPE dy = ypos[j] - ypos[i];
          const DATATYPE dz = zpos[j] - zpos[i];
          const DATATYPE dsquared = (dx * dx) + (dy * dy) + (dz * dz) + SOFT;

          const DATATYPE F = G * masses[i] * masses[j];

          const DATATYPE d32 = 1 / POW(dsquared, 3.0f / 2.0f);

          forcesx[i - ii] += F * dx * d32;
          forcesy[i - ii] += F * dy * d32;
          forcesz[i - ii] += F * dz * d32;
        }
      }

#pragma omp simd aligned(dpx, dpy, dpz, xvi, yvi, zvi, forcesx, forcesy, forcesz)
      for (i = ii; i < ii + BLOCKSIZE; i++)
      {
        const DATATYPE ax = forcesx[i - ii] / masses[i];
        const DATATYPE ay = forcesy[i - ii] / masses[i];
        const DATATYPE az = forcesz[i - ii] / masses[i];

        const DATATYPE dvx = (xvi[i] + (ax * DT * 0.5));
        const DATATYPE dvy = (yvi[i] + (ay * DT * 0.5));
        const DATATYPE dvz = (zvi[i] + (az * DT * 0.5));

        dpx[i] = dvx * DT; dpy[i] = dvy * DT; dpz[i] = dvz * DT;

        xvi[i] = dvx; yvi[i] = dvy; zvi[i] = dvz;
      }
    }

#pragma omp parallel
    {
#pragma unroll(BLOCKSIZE)
#pragma omp for simd aligned(xpos, ypos, zpos, dpx, dpy, dpz)
      for (int i = 0; i < N; i++)
      {
        xpos[i] += dpx[i]; ypos[i] += dpy[i]; zpos[i] += dpz[i];
      }
    }
  }
```

Figura 2.2: Implementación optimizada en C

2.5. Resumen

Uno de los lenguajes de programación más usados en el área de HPC es el lenguaje C. Este lenguaje se caracteriza por ser eficiente y de bajo nivel, lo que permite al programador controlar aspectos del sistema tales como accesos a la memoria y uso de recursos. Además, tiene la capacidad de extenderse con librerías para procesamiento concurrente y paralelo tales como OpenMP, OpenMPI, MPICH, entre otras. El problema de C, y generalmente de los lenguajes de bajo nivel, consiste en que es responsabilidad del programador evitar compartimientos indefinidos, debido a que el compilador no realiza ningún chequeo en los accesos a memoria. Además, generalmente desarrollar código en C requiere de un esfuerzo mayor a comparación de lenguajes de alto nivel.

Es por esto que en 2010 surge el lenguaje de programación Rust, cuyo objetivo es desarrollar algoritmos con un rendimiento equivalente a C, pero aumentando lo mayor posible la seguridad en los accesos a memoria y sin perder las bondades de un lenguaje de alto nivel.

Con el fin de comparar tanto el rendimiento como el esfuerzo de programación de Rust frente a C, se seleccionó como caso de estudio el problema de N-Body. Este es un problema conocido en el ámbito de HPC, ya que se beneficia del procesamiento paralelo por ser de alta demanda computacional.

Capítulo 3

Aceleración de la Simulación de N Cuerpos Computaciones con Atracción Gravitacional usando Rust

En este capítulo se explican las versiones implementadas del algoritmo de N-Body en Rust. Se desarrolla una primera versión secuencial (Sección 3.1) y se aplican diferentes optimizaciones incrementales (Sección 3.2) hasta obtener una versión paralela optimizada.

3.1. Implementación secuencial base

Inicialmente se desarrolló una versión secuencial, en la que se obtuvieron los mismos resultados que la equivalente en *C*. La implementación secuencial optimizada se muestra en la Figura 3.1.

```
1  (0..D).for_each(|_| {
2      let forces = poses.iter().zip(&masses[..]).map(|(pj, massj)| {
3          let force = Force::new();
4          zip(&poses[..], &masses[..]).for_each(|(pi, massi)| {
5              let dx = pi.x - pj.x;
6              let dy = pi.y - pj.y;
7              let dz = pi.z - pj.z;
8              let dsquared = (dx * dx) + (dy * dy) + (dz * dz) + SOFT;
9              let d32 = 1. / dsquared.sqrt().powi(3);
10             let f = G * *massj * *massi;
11
12             force.x += f * dx * d32;
13             force.y += f * dy * d32;
14             force.z += f * dz * d32;
15         });
16
17         force.x = (force.x  / *massj) * 0.5 * DT;
18         force.y = (force.y  / *massj) * 0.5 * DT;
19         force.z = (force.z  / *massj) * 0.5 * DT;
20     });
21
22     bodies.iter_mut().zip(forces).for_each(|(body, force)| {
23         let (dvx, dvy, dvz) = body.add_force(&force);
24
25         body.xvi = dvx;
26         body.yvi = dvy;
27         body.zvi = dvz;
28         body.dpx = dvx * DT;
29         body.dpy = dvy * DT;
30         body.dpz = dvz * DT;
31     });
32
33     poses
34         .iter_mut()
35         .zip(&bodies[..])
36         .for_each(|(pos, body)| pos.add_body(body));
37 });
38
```

Figura 3.1: Implementación secuencial

El primer *for_each* (línea 1) itera sobre el total de los instantes de tiempo de la simulación. Luego, en la línea 2, se crea un iterador que realizará las N^2 iteraciones posibles para calcular las fuerzas para cada uno de los N cuerpos. En la línea 4 se realiza un *zip* que concatena para cada posición, su masa y calcula la nueva fuerza de cada cuerpo. Finalmente, en el iterador de la línea 22, se desplazan los cuerpos.

A partir de esta versión, se consideraron diferentes optimizaciones que se han aplicado de manera incremental. Estas optimizaciones se explican en la siguiente sección.

3.2. Optimizaciones

3.2.1. Multi-hilado

En esta versión se paraleliza el procesamiento de los datos mediante el uso de la librería mencionada anteriormente: *Rayon*.

Para lograr este objetivo, se deben respetar las siguientes dependencias del problema:

1. Es necesario que se terminen de calcular las fuerzas para comenzar a desplazar los cuerpos.

2. Se debe esperar a que todos los hilos muevan sus cuerpos antes de usar esas posiciones para el cálculo de las fuerzas de la siguiente iteración.

En ese sentido, en la Figura 3.1 queda en evidencia las secciones del código donde se necesita sincronizar: como se explicó, en la línea 2 se crea el iterador que calcula las fuerzas. En la línea 22 se ejecuta el iterador de los cuerpos y para cada cuerpo se concatena una iteración del iterador de la línea 2. Al hacer esta concatenación, se ejecutará una iteración de las fuerzas, que consiste en iterar N veces sobre las posiciones y las masas, para finalmente tener la nueva fuerza. Con este nuevo dato de la fuerza junto con el cuerpo, se ejecuta el código de la iteración entre las líneas 23 y 30; de esta forma, se cumple la dependencia 1. En el iterador de la línea 33 se reposicionan los cuerpos para posteriormente pasar a la siguiente iteración $(D + 1)$, respetando la dependencia 2.

En la Figura 3.2 puede observarse que únicamente agregando a los iteradores el prefijo *par_* se paralelizan los mismos. Cada sección paralela en *Rayon* agrega una barrera implícita, asegurando la sincronización de los hilos.

```
1  (0..D).for_each(|_| {
2      let forces = poses.par_iter().zip(&masses[..]).map(|(pj, massj)| {
3          let force = Force::new();
4          zip(&poses[..], &masses[..]).for_each(|(pi, massi)| {
5              let dx = pi.x - pj.x;
6              let dy = pi.y - pj.y;
7              let dz = pi.z - pj.z;
8              let dsquared = (dx * dx) + (dy * dy) + (dz * dz) + SOFT;
9              let d32 = 1. / dsquared.get().sqrt().powi(3);
10             let f = G * *massj * *massi;
11             force.x += f * dx * d32,
12             force.y += f * dy * d32,
13             force.z += f * dz * d32,
14
15         });
16
17         force.x =  (force.x / *massj) * 0.5 * DT;
18         force.y =  (force.y / *massj) * 0.5 * DT;
19         force.z =  (force.z / *massj) * 0.5 * DT;
20     });
21
22     bodies.par_iter_mut().zip(forces).for_each(|(body, force)| {
23         let (dvx, dvy, dvz) = body.add_force(&force);
24
25         body.xvi = dvx;
26         body.yvi = dvy;
27         body.zvi = dvz;
28         body.dpx = dvx * DT;
29         body.dpy = dvy * DT;
30         body.dpz = dvz * DT;
31     });
32
33     poses
34         .par_iter_mut()
35         .zip(&bodies[..])
36         .for_each(|(pos, body)| pos.add_body(body));
37 });
```

Figura 3.2: Implementación paralela

3.2.2. Iterador *fold*

En la iteración de la línea 4 de la Figura 3.2, es necesario iterar por cada posición para calcular la nueva fuerza. En definitiva, el resultado de esta iteración será un sólo dato. Esto puede optimizarse empleando el método *fold* (*reduce*, en programación funcional), que irá actualizando la fuerza en cada iteración. El código modificado puede observarse en la Figura 3.3, a partir de la línea 3.

```rust
(0..D).for_each(|_| {
    let forces = poses.par_iter().zip(&masses[..]).map(|(pj, massj)| {
        let force =
            zip(&poses[..], &masses[..]).fold(Force::new(), |force_acc, (pi, massi)| {
                let dx = pi.x - pj.x;
                let dy = pi.y - pj.y;
                let dz = pi.z - pj.z;
                let dsquared = (dx * dx) + (dy * dy) + (dz * dz) + SOFT;
                let d32 = 1. / dsquared.sqrt().powi(3);
                let f = G * *massj * *massi;
                Force::new_with(
                    force_acc.x + f * dx * d32,
                    force_acc.y + f * dy * d32,
                    force_acc.z + f * dz * d32,
                )
            });

        Force::new_with(
            (force.x / *massj) * 0.5 * DT,
            (force.y / *massj) * 0.5 * DT,
            (force.z / *massj) * 0.5 * DT,
        )
    });

    bodies.par_iter_mut().zip(forces).for_each(|(body, force)| {
        let (dvx, dvy, dvz) = body.add_force(&force);

        body.xvi = dvx;
        body.yvi = dvy;
        body.zvi = dvz;
        body.dpx = dvx * DT;
        body.dpy = dvy * DT;
        body.dpz = dvz * DT;
    });

    poses
        .par_iter_mut()
        .zip(&bodies[..])
        .for_each(|(pos, body)| pos.add_body(body));
});
```

Figura 3.3: Implementación paralela con fold

3.2.3. Optimizaciones matemáticas

Rust provee intrínsecas [36] que permiten aplicar optimizaciones en las operaciones matemáticas basado en reglas algebraicas, lo que posibilita computar con mayor velocidad, pero cediendo precisión. Si bien aún no están en la versión estable de Rust, estas intrínsecas son muy utilizadas para mejorar el rendimiento de las operaciones matemáticas. Entre ellas se destacan:

- *fadd_fast*: suma optimizada.
- *fsub_fast*: resta optimizada.
- *fmul_fast*: multiplicación optimizada.
- *fdiv_fast*: división optimizada.
- *frem_fast*: MOD optimizado.

Para hacer uso de estas instrucciones, se utilizó la librería fast-floats [1] que implementa estas optimizaciones de forma transparente sin la necesidad de llamar a las rutinas **_fast* en cada operación.

Se han verificado que los resultados obtenidos mediante la aplicación de estas optimizaciones son equivalentes a los propios en el algoritmo en *C*.

3.2.4. Vectorización

La vectorización permite realizar una misma operación sobre múltiples datos en paralelo, siendo una de las técnicas más comunes para mejorar los tiempos de respuesta de una aplicación. Mediante directivas que se envían al compilador LLVM, es posible generar código vectorizado para las instrucciones propias del hardware de base, en vez de las instrucciones vectoriales básicas.

Por defecto Rust intenta vectorizar en forma automática, pero si la complejidad del código impide que el compilador lo logre por su cuenta, el programador puede ser capaz de generar el código vectorizado de forma manual. En este caso, la vectorización manual no fue requerida para el algoritmo desarrollado (se verificó el código de máquina resultante) y, debido a que el hardware utilizado en este trabajo emplea instrucciones AVX 512, la directiva `target-feature=+avx512f` fue enviada al compilador para generar el código vectorizado con este tipo de instrucciones.

3.2.5. Jemalloc

Por defecto, todos los programas en Rust utilizan el alocador del sistema, el cual es de propósito general e intenta adecuarse lo mejor posible a diferentes ámbitos. Afortunadamente, existen alocadores que se adaptan mejor dependiendo del tipo de aplicación. Uno de los alocadores más utilizados para aplicaciones concurrentes y paralelas es *Jemallocator* [37], el cual se usa en este trabajo.

Jemalloc es un alocador cuyo beneficio principal es la escalabilidad en sistemas multiprocesador y multiproceso. Esto lo logra mediante el uso de múltiples arenas, que son fragmentos de memoria sin procesar a partir de los cuales se realizan las asignaciones.

Jemalloc funciona de la siguiente manera: en ambientes con múltiples subprocesos, Jemalloc crea muchas arenas (cuatro veces más arenas que procesadores) y los hilos se asignan a estas arenas de forma circular; esto reduce el bloqueo entre hilos, debido a que estos competirán únicamente si pertenecen

[1] Esta librería se encuentra en https://github.com/bluss/fast-floats y se utilizó la extensión https://github.com/bluss/fast-floats/pull/2 para evitar resultados indefinidos.

a la misma arena, en caso contrario, los hilos no se afectarán entre sí. A su vez, Jemalloc intenta optimizar la localidad de la caché para tener accesos contiguos a los datos en la memoria [38].

Para utilizar Jemalloc en una aplicación hecha en Rust, se debe importar la libreria Jemallocator y configurar el alocador del sistema como muestra la Figura 3.4.

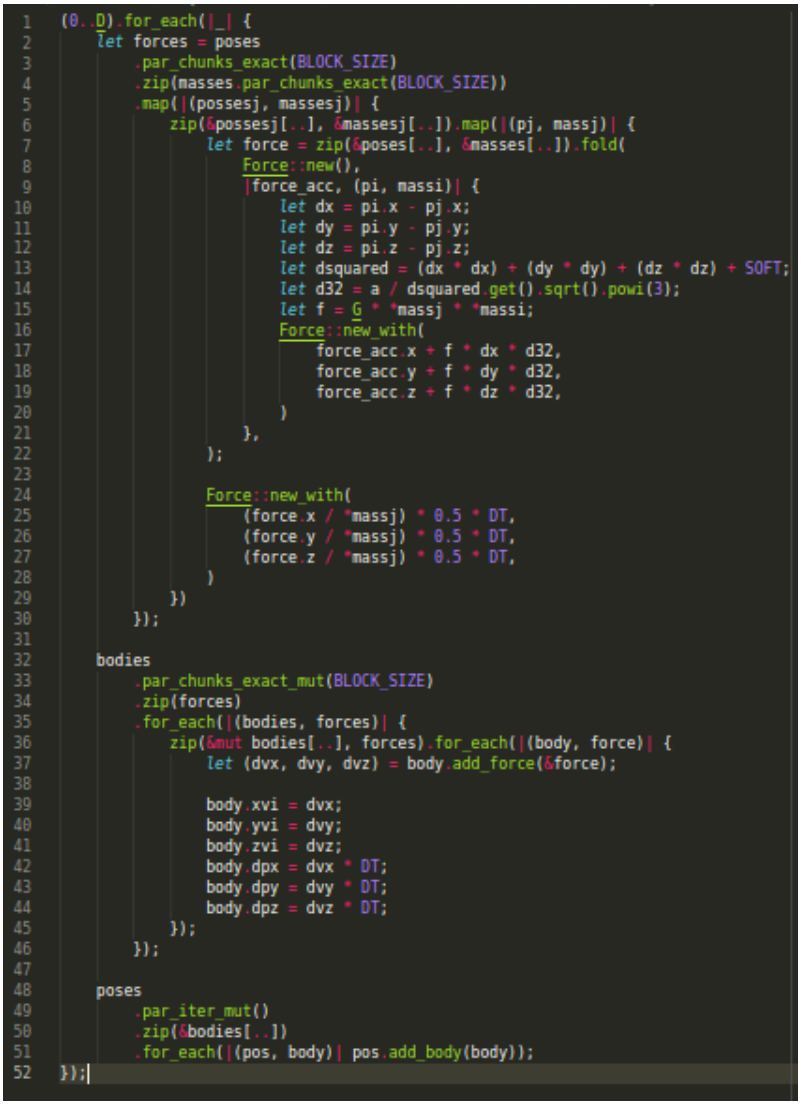
```
extern crate jemallocator;

#[global_allocator]
static GLOBAL: jemallocator::Jemalloc = jemallocator::Jemalloc;
```

Figura 3.4: Configuración de Jemalloc en código de Rust

3.2.6. Procesamiento por bloques

Con el fin de aprovechar la localidad de datos, es posible re-estructurar el cómputo de este problema para poder realizar un procesamiento por bloques. Esto requiere modificar el código fuente ya que es necesario alterar los bucles, lo que conlleva a cambiar la lógica de acceso a los datos, el lugar donde se realizan las operaciones, entre otras modificaciones. Además, para este tipo de optimizaciones se debe configurar tanto el tamaño de los bloques como la cantidad de hilos a ejecutar, y el rendimiento dependerá de la arquitectura subyacente; es decir que un bloque de tamaño X, puede ser óptimo para una arquitectura, pero no para otra.

En este trabajo se intentó realizar un procesamiento en bloques, variando el tamaño de los bloques entre 8, 16, y 32. El código de la Figura 3.5 muestra el algoritmo procesando por bloques. La idea central es que cada proceso trabaje con bloques de tamaño *BLOCK_SIZE*, aprovechando la localidad de los datos.



```
(0..D).for_each(|_| {
    let forces = poses
        .par_chunks_exact(BLOCK_SIZE)
        .zip(masses.par_chunks_exact(BLOCK_SIZE))
        .map(|(possesj, massesj)| {
            zip(&possesj[..], &massesj[..]).map(|(pj, massj)| {
                let force = zip(&poses[..], &masses[..]).fold(
                    Force::new(),
                    |force_acc, (pi, massi)| {
                        let dx = pi.x - pj.x;
                        let dy = pi.y - pj.y;
                        let dz = pi.z - pj.z;
                        let dsquared = (dx * dx) + (dy * dy) + (dz * dz) + SOFT;
                        let d32 = a / dsquared.get().sqrt().powi(3);
                        let f = G * *massj * *massi;
                        Force::new_with(
                            force_acc.x + f * dx * d32,
                            force_acc.y + f * dy * d32,
                            force_acc.z + f * dz * d32,
                        )
                    },
                );

                Force::new_with(
                    (force.x / *massj) * 0.5 * DT,
                    (force.y / *massj) * 0.5 * DT,
                    (force.z / *massj) * 0.5 * DT,
                )
            })
        });

    bodies
        .par_chunks_exact_mut(BLOCK_SIZE)
        .zip(forces)
        .for_each(|(bodies, forces)| {
            zip(&mut bodies[..], forces).for_each(|(body, force)| {
                let (dvx, dvy, dvz) = body.add_force(&force);

                body.xvi = dvx;
                body.yvi = dvy;
                body.zvi = dvz;
                body.dpx = dvx * DT;
                body.dpy = dvy * DT;
                body.dpz = dvz * DT;
            });
        });

    poses
        .par_iter_mut()
        .zip(&bodies[..])
        .for_each(|(pos, body)| pos.add_body(body));
});
```

Figura 3.5: Implementación con procesamiento por bloques

3.3. Resumen

En este capítulo se presentaron los algoritmos desarrollados en Rust, desde su versión secuencial hasta la versión paralela optimizada. Los algoritmos fueron optimizados de manera incremental: en primer lugar, se realizó una versión secuencial que fue paralelizada mediante la librería Rayon, donde únicamente fue necesario agregar el prefijo *par_* en cada iterador que se quiera ejecutar en paralelo. Luego, esta solución fue optimizada mediante la modificación de del iterador *for_each* común, a un iterador de tipo reducción (*fold*). Poste-

riormente se aplicaron optimizaciones matemáticas, que consisten en rutinas específicas que sacrifica precisión en el cálculo en pos de mejorar el rendimiento en la operación. Seguido a esto, se investigó en relación a la vectorización automática del compilador de Rust (LLVM), resultando en la necesidad de indicarle al compilador el conjunto de instrucciones SIMD del hardware objetivo. Seguidamente, se presentó Jemalloc, el alocador de memoria utilizado, que obtiene buenos rendimientos en aplicaciones paralelas y concurrentes. Como última optimización, se intentó modificar la lógica del código para procesar el algoritmo por bloques, que consistió en separar el procesamiento en bloques de tamaño *BLOCK_SIZE*.

Capítulo 4

Resultados Experimentales

En este capítulo se describe el entorno utilizado para las pruebas experimentales, se explican las pruebas que se realizaron y se analizan los resultados obtenidos.

4.1. Diseño experimental

Todas las pruebas fueron realizadas en un servidor 2×Intel Xeon Platinum 8276 con 28-core 2.20Ghz (con instrucciones AVX-512) y 256 GB de RAM; cuenta con 112 hilos hardware. Para la compilación del código en C, se utilizó el compilador ICC (versión 19.1.0.166), enviando las siguientes directivas:

- `-fp-model fast=2`: activa las optimizaciones matemáticas.
- `-xCORE-AVX512`: habilita las instrucciones AVX-512.
- `-qopt-zmm-usage`: alienta al compilador a usar las unidades SIMD.
- `-qopenmp`: librería de OpenMP.

En cuanto a Rust, se utilizó la versión 1.48.0 del lenguaje y la versión 11.0 del compilador LLVM. Se envió el flag de compilación `-Ctarget-feature=+avx512f` para habilitar las instrucciones vectoriales AVX-512.

Para las pruebas realizadas, en primer lugar se partió de una versión base del algoritmo de Rust y se analizó el impacto de aplicar cada optimización incrementalmente sobre el mismo.

Luego, se seleccionó la mejor versión de Rust para comparar con su equivalente de C. Se dividieron las pruebas en precisión simple y precisión doble, variando tanto la carga de trabajo (N = {65536, 131072, 262144, 524288 y

1048576}) como el número de hilos (T = {56, 112 y auto}) y el tamaño de bloque (BLOCK_SIZE = {8, 16, 32}) donde corresponda.

Los programas trabajaron sobre los mismos datos y se ha verificado la corrección de los resultados de los algoritmos.

4.2. Rendimiento

Para evaluar el rendimiento se emplea la métrica GFLOPS (mil millones de FLOPS)

$$GFLOPS = \frac{20 \times N^2 \times I}{T \times 10^9}$$

donde N es el número de cuerpos, I es el número de pasos, T es el tiempo de ejecución (en segundos) y el factor 20 representa la cantidad de operaciones en punto flotante requerida por cada iteración.

En la Figura 4.1 se pueden ver los rendimientos al variar la cantidad de cuerpos y el número de hilos en la versión paralela más simple, sin ningún tipo de optimización junto con los rendimientos de la versión secuencial. Esta última apenas alcanza los 4 GFLOPS. Comparando la versión secuencial con la paralela, la última es hasta un 50× más rápida. Por otro lado, se puede apreciar que el uso de *hyper-threading* provee una pequeña ganancia de rendimiento (hasta 5 %). Finalmente, es posible notar también que la opción `auto` de Rust selecciona apropiadamente el número de hilos a usar, alcanzando tasas de FLOPS similares a las obtenidas configurando el número de hilos de forma manual.

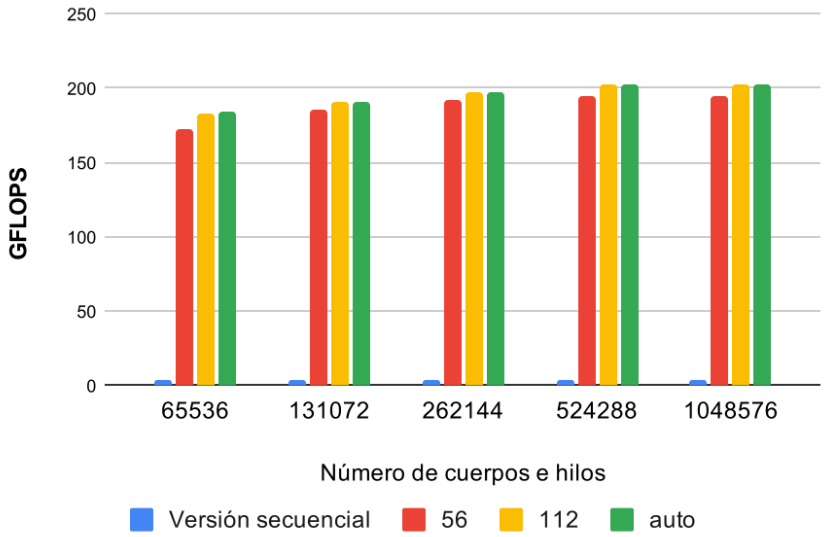


Figura 4.1: Rendimientos obtenidos para diferentes cantidades de cuerpos y número de hilos en la primera versión paralela

En la Figura 4.2 se pueden observar los rendimientos al modificar el algoritmo paralelo con el iterador *fold*. Es evidente que Rust no adhiere costos al utilizar aspectos de lenguajes de alto nivel como este tipo de iteradores. Esto permite mejorar el código para hacerlo más entendible y escalable, sin sacrificar rendimiento.

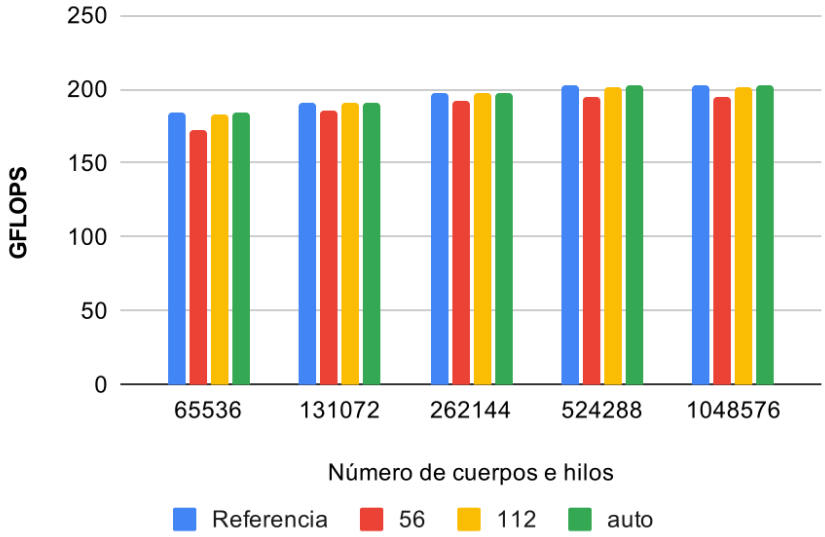


Figura 4.2: Rendimientos obtenidos al incorporar el iterador *fold*

Al aplicar las optimizaciones matemáticas (Figura 4.3), es notable la diferencia de rendimiento frente a la versión anterior. la tasa de FLOPS incrementa en un 6.1-6.4× con 112 hilos (el mejor caso). Sin embargo, es importante remarcar que esta mejora se obtiene a costa de disminuir la precisión en los resultados.

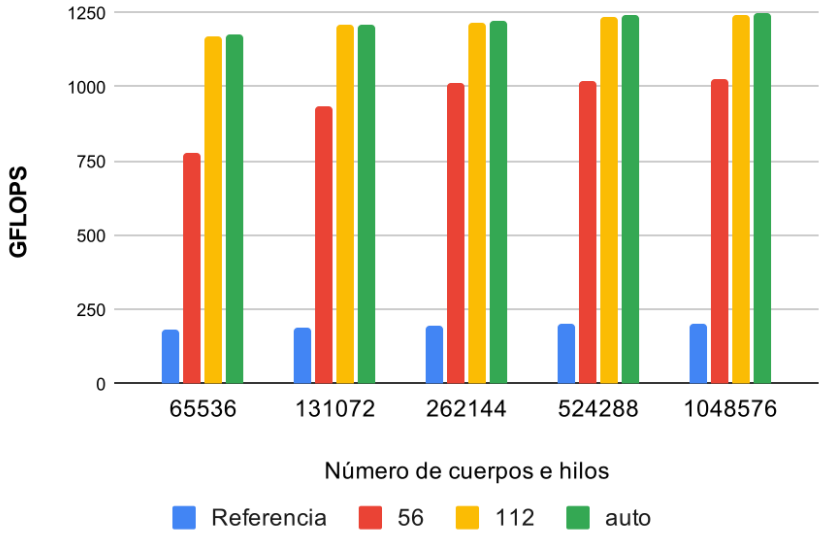


Figura 4.3: Rendimientos obtenidos al aplicar optimizaciones matemáticas

En la Figura 4.4 se observan los rendimientos al indicar al compilador que debe vectorizar el código utilizando instrucciones AVX-512. Al aplicar esta optimización, el rendimiento aumentó aproximadamente un 68 %. Rust detecta el tipo de hardware e intenta vectorizar con las instrucciones correspondientes por defecto, pero al indicar explícitamente el tipo a utilizar, puede mejorar su traducción incorporando instrucciones específicas como la `vrsqrt14ps` (o su variante en 64-bit `vrsqrt14pd`), usada para la aproximación de la raíz cuadrada recíproca.

En la Figura 4.5 se pueden observar los rendimientos al cambiar el alocador de memoria por Jemalloc. Con esta modificación se produce una mejora de 1.1× aproximadamente al usar T={112, auto}. Si bien no es una gran mejora, su costo es prácticamente nulo, por lo que su inclusión resulta productiva.

Como última optimización, se aplica un procesamiento por bloques. La Figura 4.6 presenta para cada tamaño de bloque y cantidad de cuerpos, el rendimiento obtenido. Estas mediciones se hicieron configurando el número de hilos de forma automática debido a que fue la de mejor rendimiento. Como se puede notar, no hay diferencias significativas en el rendimiento obtenido para cada tamaño de bloque configurado.

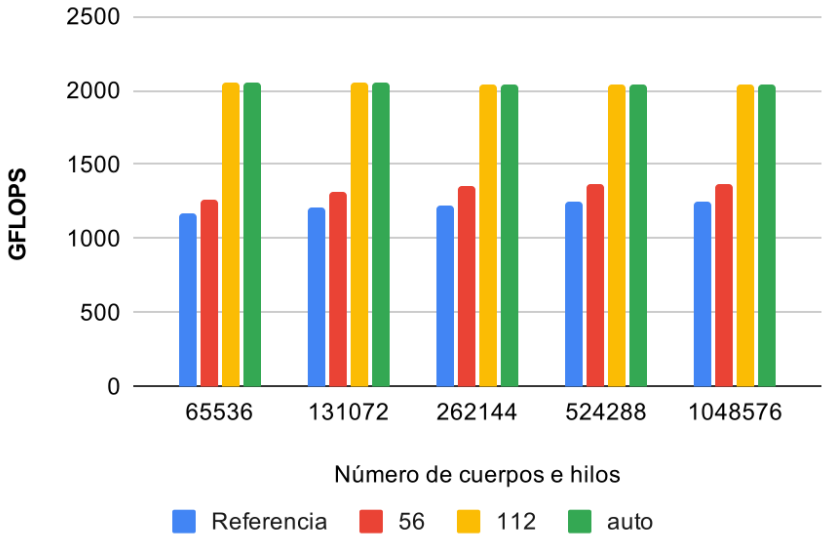


Figura 4.4: Rendimientos obtenidos al vectorizar con instrucciones AVX-512

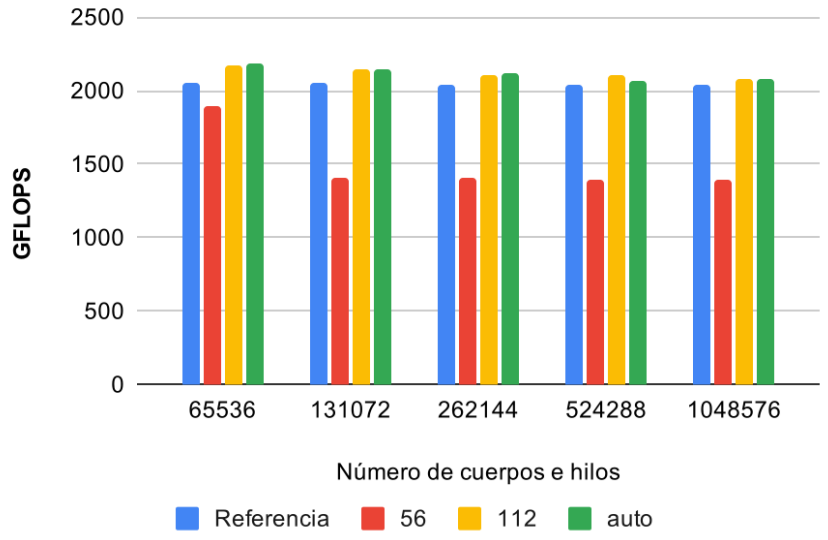


Figura 4.5: Rendimientos obtenidos al utilizar Jemalloc como alocador por defecto

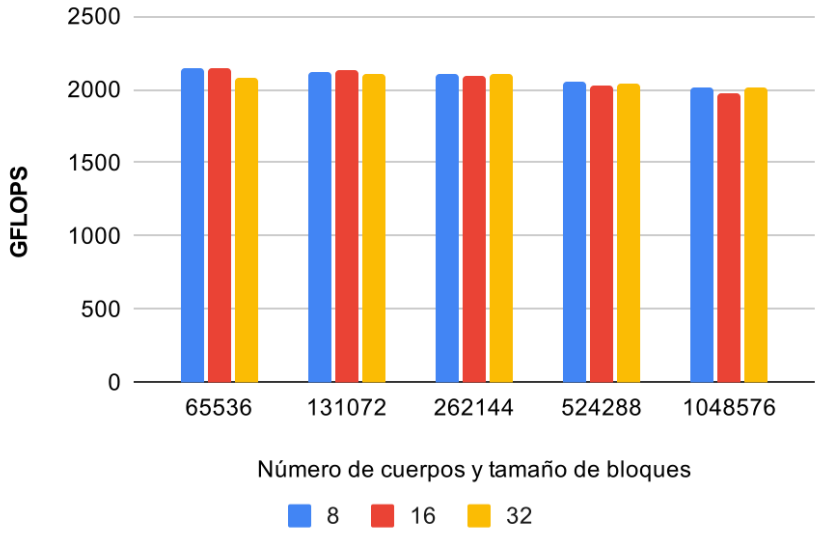


Figura 4.6: Rendimientos obtenidos al aplicar procesamiento por bloques

En la Figura 4.7 se compara la última versión optimizada junto con la versión que procesa por bloques. Queda en evidencia que el procesamiento por bloques no mejora el rendimiento de la aplicación. Esto es debido a que los iteradores de Rust administran la memoria de forma eficiente, aprovechando la localidad de datos implícitamente.

Por último, en la Tabla 4.8 se presenta la comparación final entre C y Rust, variando el número de cuerpos y el tipo de datos entre precisión simple (float) y precisión doble (double). En precisión simple, la versión de C supera a la de Rust para todos los tamaños de problemas, logrando mejoras de hasta 1.18×, mientras que en precisión doble las dos implementaciones tienen prácticamente el mismo rendimiento. Al analizar el código assembler generado por ambas implementaciones, se puede notar que C realiza una traducción más eficiente del código principal cuando las optimizaciones matemáticas (relajación de precisión) son usadas. Este comportamiento no se replica en precisión doble, donde ambos códigos son muy similares. Como se explicó, estas optimizaciones aún no están incluidas en la versión estable de Rust, por lo que se espera que mejore en el futuro.

4.3. Esfuerzo de programación

En este trabajo no sólo interesa el rendimiento de la aplicación sino también el esfuerzo de programación requerido para desarrollarlo junto a su manteni-

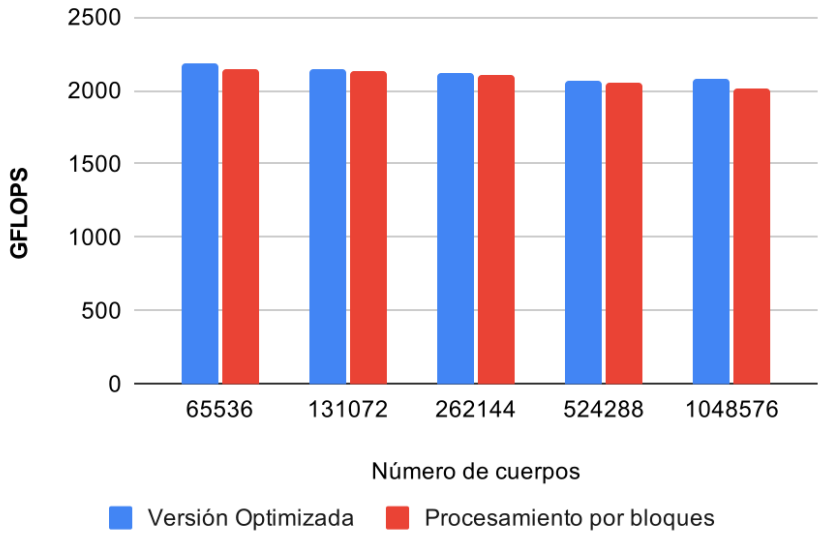


Figura 4.7: Comparación de rendimiento entre la versión optimizada y la versión con procesamiento por bloques

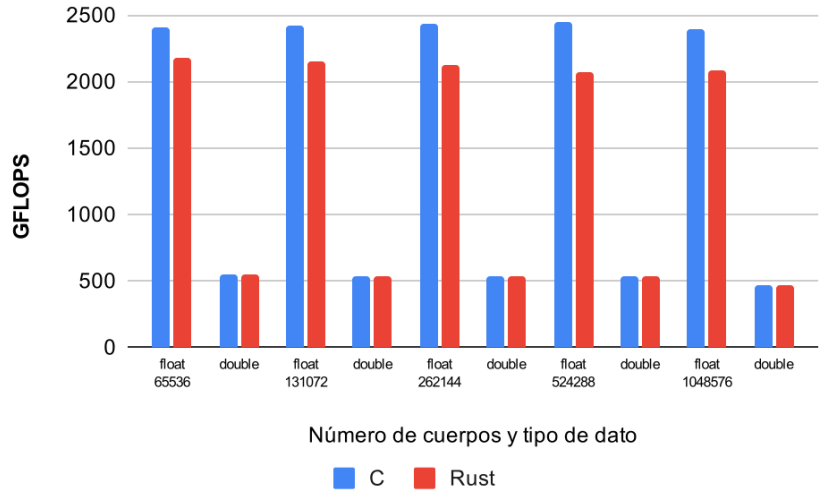


Figura 4.8: Comparación de rendimientos entre las versiones más optimizadas de C y Rust

bilidad.

Existen numerosas propuestas para medir el costo de programación. Algunas incluyen contar la cantidad de líneas de código (*Source Lines Of Code*, *SLOC*) o el número de caracteres (incluyendo líneas en blanco y comentarios). A pesar de su sencillez, estos parámetros no reflejan la complejidad del algoritmo [39]. Otras alternativas miden el tiempo de desarrollo aunque se vuelve dependiente de la experiencia del programador y no siempre es posible hacerlo, como en este trabajo [1].

La subjetividad de estas métricas dificulta la evaluación del costo de programación. Es por eso que en este trabajo, además de medir el costo de programación a través del indicador SLOC, se realiza una comparación cualitativa del esfuerzo requerido en las soluciones. Como ambas partes son complementarias, permiten al lector un entendimiento más amplio del esfuerzo de programación requerido.

	C	**Rust**
Principal	66	40
Total	219	195

Tabla 4.1: SLOC en los algoritmos finales

En particular, como puede observarse en la Tabla 4.1, el bloque de código principal de este algoritmo en C ocupa **66** líneas de código (Figura 2.2), mientras que el propio en Rust tiene **40** líneas de código. En total, modularizando la solución, junto con todas las líneas necesarias, en C se necesitaron **219** líneas de código mientras que en Rust **195** líneas de código. Considerando el bloque de código principal, C requirió 65 % más de SLOC que Rust. Si bien este porcentaje se reduce 12.5 % al considerar todo el código, resulta importante aclarar que en Rust se crearon clases para representar una fuerza, un cuerpo y una posición, cada una con sus respectivos métodos; en líneas generales, esto impacta favorablemente en la mantenibilidad del programa. Además, Rust tiene como ventaja que al ser un lenguaje con bondades de alto nivel, funcional y orientado a objetos, permite desarrollar código más compacto que C y más autoexplicativo.

Respecto a la comparación cualitativa, se pueden considerar los siguientes aspectos: en C fue necesario cambiar la lógica de la solución para adaptarla a un procesamiento por bloques con el fin de aprovechar la localidad de los datos. Esto requiere tanto encontrar el tamaño de bloque óptimo, como

[1] Como se ha utilizado una versión de C ya desarrollada, no fue posible medirlo.

también realizar modificaciones en el código para garantizar el procesamiento adecuado. Para este algoritmo, Rust administró eficientemente la memoria, evitando tener que modificar la solución. Además, en Rust fue posible generar código paralelo de manera sencilla (agregando el prefijo *_par* a los iteradores), mientras que en C fue necesario indicar diferentes opciones de OpenMP para paralelizar correctamente el código. Otro aspecto importante a destacar es que en Rust es muy simple agregar librerías externas, como por ejemplo la librería de optimizaciones matemáticas, o la de Rayon. Únicamente se debe especificar el nombre de la librería y la versión en un archivo de configuración (*Cargo.toml*), y en la próxima compilación, ya estará disponible para su uso.

En favor de C frente a Rust, las optimizaciones matemáticas pueden habilitarse enviando el flag *-fp-model fast=2* al compilador. En cambio en Rust es necesario utilizar las intrínsecas del lenguaje, que si bien es posible usar librerías para evitar realizar grandes modificaciones en el código, no deja de ser un punto desfavorable del lenguaje hasta el momento.

4.4. Trabajos relacionados

Al día de hoy, sólo se pueden mencionar unos pocos estudios preliminares que exploran las capacidades del lenguaje Rust para aplicaciones de procesamiento paralelo.

En el 2015, se publicó una tesina [11] que evalúa el rendimiento y la productividad de los lenguajes Rust, Go y C para un caso de estudio en particular (cálculo del camino más corto en grafos). Como conclusión, los autores expresaron que la implementación Rust fue la que mayor rendimiento y productividad obtuvo. En este TFI, aunque la versión Rust no logró superar en rendimiento a su contraparte en C, sí fue la de menor costo de programación.

En [40] los autores exploran las ventajas y desventajas de Rust en el ámbito de la astrofísica al re-implementar partes fundamentales del software Mercury-T (originalmente escrito Fortran). Los autores concluyen que Rust ayuda a evitar errores comunes tanto en el acceso a la memoria como también en las condiciones de carrera. Además, expresan que si bien Rust puede tener una curva de aprendizaje inicial costosa, una vez que se aprende, los beneficios son inmediatos. En el transcurso de la realización de los algoritmos en Rust para este TFI, fue posible apreciar cómo el compilador guía al programador para evitar errores en los accesos a la memoria, informando en detalle el problema en cuestión. Además, como explican los autores, puede ser desafiante al principio desarrollar código en Rust, pero cuando se entiende la filosofía del lenguaje, se reducen los tiempos de desarrollo.

En [41] se comparan implementaciones de N-Body considerando los lengua-

jes Rust y C. Los autores concluyen en que pudieron desarrollar una solución paralela en Rust de forma simple, pero que aún está en proceso de optimización. Es por esto que el algoritmo Rust obtuvo un menor rendimiento que C; lo que da muestras de que es un lenguaje que requiere que se estudien sus particularidades para obtener sus beneficios. Sucedió lo mismo con los algoritmos de Rust implementados en este TFI, debido a que en un principio se partió de una solución no optimizada y, mediante la aplicación de diferentes optimizaciones, fue posible obtener resultados cercanos a los tiempos en C.

4.5. Resumen

En este capítulo, se realizó una comparación entre Rust y C para el problema N-Body considerando el rendimiento y esfuerzo de programación requerido.

Como primera instancia, se evaluaron los rendimientos de los algoritmos desarrollados en Rust, donde se analizaron las optimizaciones implementadas con el fin de conocer el impacto que genera cada una en el rendimiento. Para esto se plantearon pruebas experimentales que variaron en la cantidad de hilos y el tamaño del problema y se tomaron mediciones con diferentes valores para estas variables. Estas mediciones fueron calculadas en base a la métrica de GFLOPS.

Las optimizaciones que se compararon conformaron la paralelización del código secuencial, la aplicación del iterador *fold* en lugar del bucle *for_each*, las optimizaciones matemáticas, el envío de la directiva `-Ctarget-feature=+avx512f` para activar la vectorización con instrucciones AVX-512, la inclusión de Jemalloc como el alocador de memoria por defecto y la modificación de la lógica del algoritmo para procesar por bloques.

Cabe destacar, por un lado, que la optimización que mayor impacto produjo fueron en primer lugar la paralelización del código secuencial, y en segundo lugar las optimizaciones matemáticas, que aumentaron los GFLOPS de la solución considerablemente. Por otro lado, la optimización de procesamiento por bloques no dio mejores resultados, por lo cual no fue tenido en cuenta para el algoritmo final.

Una vez optimizado el algoritmo Rust, se comparó con el propio en C. En precisión simple, C fue quien superó a Rust; sin embargo, en precisión doble, ambos lenguajes prácticamente tuvieron el mismo rendimiento. La causa de la diferencia de rendimientos radica en que por el momento, Rust no optimiza tan bien como C las operaciones matemáticas en precisión simple.

Por último, se evaluaron los lenguajes en cuanto al esfuerzo de programación requerido y en las capacidades de ambos lenguajes para ser escalables y mantenibles. Para medir el costo de programación se utilizó el indicador SLOC

que reflejó que el código principal en Rust necesitó 65 % de líneas menos que en C. Además, Rust a diferencia de C, no requirió de un procesamiento por bloques lo que evita modificar el código para que se adapte a esta optimización, eliminando la necesidad del programador de cambiar la lógica de procesamiento.

Capítulo 5

Conclusiones y Líneas de Trabajo a Futuro

La elección del lenguaje de programación es una decisión fundamental que impactará tanto en el desarrollo como en el rendimiento del sistema. En la comunidad HPC el foco está centrado en el rendimiento, lo que evita usar lenguajes de alto nivel que si bien mejoran la experiencia y reducen los tiempos de desarrollo, aportan una gran cantidad de *overhead* y no poseen las bondades de los lenguajes de bajo nivel.

En la última década, Rust se ha posicionado como una seria alternativa a C para software general como juegos, multimedia y cualquier otro tipo de aplicación que requiera procesamiento concurrente y distribuido.

Para evaluar la factibilidad del uso de Rust en el ámbito de HPC, en este trabajo se seleccionó el algoritmo de N-Body, que es un problema que requiere cómputo intensivo y paralelo. Con este fin, se realizaron algoritmos en Rust que partieron desde una versión base y, aplicando optimizaciones incrementales, se obtuvo a la versión final. Estas optimizaciones fueron las siguientes:

1. Multi-hilado: dado el código de base, el código paralelo se obtuvo únicamente agregando el prefijo *_par*. Adicionalmente, Rust seleccionó adecuadamente la cantidad de hilos en forma automática.

2. Modificación de iterador procedural (*for_each*) por el iterador de reducción (*fold*): esta modificación prácticamente no reflejó impacto alguno en el rendimiento pero aumenta la autocomprensión del código.

3. Optimizaciones matemáticas: si bien estas optimizaciones provocan una pérdida de precisión en los resultados, sí mejoraron considerablemente el rendimiento del algoritmo. Se utilizó una librería que permite (casi)

de manera implícita incorporar estas optimizaciones sin la necesidad de alterar el código.

4. Vectorización: Rust permitió auto-vectorizar el código (no fue necesaria una vectorización manual). Sin embargo, la ausencia de primitivas de vectorización en este lenguaje (como `simd` en OpenMP) puede ser una limitación cuando la auto-vectorización no es posible. Adicionalmente, es recomendable indicarle al compilador el conjunto de instrucciones SIMD a utilizar, con el fin obtener una mejor selección de las mismas.

5. Alocador de memoria: el rendimiento incrementó al utilizar Jemalloc como el alocador de memoria, en lugar del alocador predeterminado. Si bien la mejora no fue significativa, su inclusión prácticamente no tiene costo

6. Procesamiento por bloques: esta optimización no produjo mejoras, lo que evita tener que modificar la lógica del algoritmo para procesar el problema en bloques.

Después de aplicar las optimizaciones, se comparó el algoritmo Rust resultante con el de C. Los resultados del rendimiento fueron cercanos en precisión doble, pero no lo fueron en precisión simple, donde C fue superior. Esto ocurre debido a que Rust no optimiza tan bien como C las operaciones matemáticas en este tipo de dato.

Respecto al esfuerzo de programación, Rust a diferencia de C, posee aspectos de un lenguaje de alto nivel lo que facilita la generación de un código mantenible de manera sencilla. Además, como tiene características de lenguaje funcional y orientado a objetos, permite generar código más compacto, lo que repercutió en una menor cantidad de líneas de código del bloque principal del programa. Adicionalmente, Rust intenta administrar la memoria eficientemente y, en algunos casos - como en este trabajo - no fue necesario realizar modificaciones a la lógica del algoritmo para sacar provecho de la localidad de los datos.

A partir de los resultados obtenidos y el análisis realizado, se considera que Rust se puede posicionar como una alternativa a C para HPC en contextos similares a los del presente estudio. Como el lenguaje aun se encuentra en constante evolución, el apoyo y soporte que le otorgue su comunidad se tornará un factor determinante en la viabilidad final.

Como trabajo a futuro, interesa extender el estudio realizado considerando dos posibles aspectos. Por un lado, seleccionar otros casos de estudio que sean intensivos en cómputo pero que posean características diferentes al seleccionado en este TFI. Por otro lado, considerar otras arquitecturas HPC, como

pueden ser los procesadores de AMD. Ambas extensiones contribuirían a darle mayor representatividad al estudio.

Bibliografía

[1] European Commission. *High Performance Computing*. https://ec.europa.eu/digital-single-market/en/high-performance-computing. 2020.

[2] Iris Christadler, Giovanni Erbacci y Alan D. Simpson. «Performance and Productivity of New Programming Languages». En: *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*. Ed. por Rainer Keller, David Kramer y Jan-Philipp Weiss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, págs. 24-35. ISBN: 978-3-642-30397-5. DOI: 10.1007/978-3-642-30397-5_3. URL: https://doi.org/10.1007/978-3-642-30397-5_3.

[3] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.0*. Mayo de 2008. URL: http://www.openmp.org/mp-documents/spec30.pdf.

[4] R. L. Graham y col. «Open MPI: A High-Performance, Heterogeneous MPI». En: *2006 IEEE International Conference on Cluster Computing*. 2006, págs. 1-9. DOI: 10.1109/CLUSTR.2006.311904.

[5] William Gropp. «MPICH2: A New Start for MPI Implementations». En: *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2002, pág. 7. ISBN: 3540442960.

[6] Xi Wang y col. «Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior». En: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, págs. 260-275. ISBN: 9781450323888. DOI: 10.1145/2517349.2522728. URL: https://doi.org/10.1145/2517349.2522728.

[7] Guillermo L. Taboada y col. «Java in the High Performance Computing arena: Research, practice and experience». En: *Science of Computer Programming* 78.5 (2013). Special section: Principles and Practice

of Programming in Java 2009/2010 & Special section: Self-Organizing Coordination, págs. 425-444. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2011.06.002. URL: http://www.sciencedirect.com/science/article/pii/S0167642311001420.

[8] Shahid Alam. «Is Fortran Still Relevant? Comparing Fortran with Java and C++». En: *ArXiv* abs/1407.2190 (2014).

[9] Stefano Masini y Paolo Bientinesi. «High-Performance Parallel Computations Using Python as High-Level Language». En: *Euro-Par 2010 Parallel Processing Workshops*. Ed. por Mario R. Guarracino y col. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, págs. 541-548. ISBN: 978-3-642-21878-1.

[10] Jan Gmys y col. «A comparative study of high-productivity high-performance programming languages for parallel metaheuristics». En: *Swarm and Evolutionary Computation* 57 (2020), pág. 100720. ISSN: 2210-6502. DOI: https://doi.org/10.1016/j.swevo.2020.100720. URL: http://www.sciencedirect.com/science/article/pii/S2210650220303734.

[11] F. Wilkens. «Evaluation of performance and productivity metrics of potential programming languages in the HPC environment». En: 2015.

[12] Steve Klabnik y Carol Nichols. *The Rust Programming Language*. USA: No Starch Press, 2018. ISBN: 1593278284.

[13] Frederic Pascal Sautter. «A FRAMEWORK FOR OBSERVING THE PARALLEL EXECUTION OF RUST PROGRAMS». En: (2019).

[14] Samuel Johansson y Ludvig Rappe. *Transitioning from C to Rust in Media Streaming Development - An Industrial Case Study*. eng. Student Paper. 2020.

[15] Abhiram Balasubramanian y col. «System Programming in Rust: Beyond Safety». En: 51.1 (sep. de 2017), págs. 94-99. ISSN: 0163-5980. DOI: 10.1145/3139645.3139660. URL: https://doi.org/10.1145/3139645.3139660.

[16] Ralf Jung y col. «RustBelt: Securing the Foundations of the Rust Programming Language». En: 2.POPL (dic. de 2017). URL: https://doi.org/10.1145/3158154.

[17] *Stack Overflow: Developer Survey Results 2019*. https://insights.stackoverflow.com/survey/2019.

[18] *EdenSCM*. https://github.com/facebookexperimental/eden.

[19] *Servo*. https://servo.org/.

[20] *Why Discord is switching from Go to Rust*. https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f.

[21] *Rewriting the heart of our sync engine*. https://dropbox.tech/infrastructure/rewriting-the-heart-of-our-sync-engine.

[22] *NPM Adopted Rust to Remove Performance Bottlenecks*. https://www.infoq.com/news/2019/03/rust-npm-performance/.

[23] *Rust companies*. https://www.rust-lang.org/production/users.

[24] Enzo Rucci y col. «Optimization of the N-Body Simulation on Intel's Architectures Based on AVX-512 Instruction Set». En: *Computer Science – CACIC 2019*. Ed. por Patricia Pesado y Marcelo Arroyo. Springer International Publishing, 2020, págs. 37-52. ISBN: 978-3-030-48325-8.

[25] Edgar Ruiz Lizama. «Lenguajes de programación: conceptos y paradigmas». En: *Industrial Data* 4.1 (2001), págs. 071-074. DOI: 10.15381/idata.v4i1.6605. URL: https://revistasinvestigacion.unmsm.edu.pe/index.php/idata/article/view/6605.

[26] Hao Chen. «Comparative Study of C, C++, C# and Java Programming Languages». En: 2010.

[27] Dennis M. Ritchie. «The Development of the C Language». En: HOPL-II. Cambridge, Massachusetts, USA: Association for Computing Machinery, 1993, págs. 201-208. ISBN: 0897915704. DOI: 10.1145/154766.155580. URL: https://doi.org/10.1145/154766.155580.

[28] *LAMMPS*. https://lammps.sandia.gov/.

[29] *GROMACS*. www.gromacs.org.

[30] *SETI@Home*. https://setiathome.berkeley.edu/.

[31] *Fearless Concurrency with Rust*. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html.

[32] *Rayon*. https://docs.rs/rayon/1.0.3/rayon/.

[33] *Rayon: data parallelism in Rust*. http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[34] *Rust - Message Passing*. https://doc.rust-lang.org/book/ch16-02-message-passing.html.

[35] P. Young. «Leapfrog method and other " symplectic " algorithms for integrating Newton ' s laws of motion». En: 2013.

[36] *Rust intrinsics*. https://doc.rust-lang.org/std/intrinsics/index.html.

[37] *Rust Jemallocator*. https://crates.io/crates/jemallocator.

[38] *Jemalloc*. http://jemalloc.net/.

[39] Kaushal Bhatt y col. *Analysis Of Source Lines Of Code(SLOC) Metric*.

[40] Sergi Blanco-Cuaresma y Emeline Bolmont. «What can the programming language Rust do for astrophysics?» En: *Proceedings of the International Astronomical Union* 12.S325 (2016), págs. 341-344. DOI: 10.1017/S1743921316013168.

[41] Alexander Hansen y Mark Lewis. «The Case for N-Body Simulations in Rust». En: jul. de 2016.