Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Análisis del rendimiento en Computación de Altas Prestaciones

Performance Analysis in High Performance Computing

Ariane Zanardi Francisco

La Laguna, 11 de septiembre de 2020

D. **Vicente Blanco Pérez**, con N.I.F. 42.171.808-C profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"*Análisis del rendimiento en Computación de Altas Prestaciones*"

ha sido realizada bajo su dirección por D. **Ariane Zanardi Francisco**, con N.I.F. 78.729.613-S.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 11 de septiembre de 2020

Agradecimientos

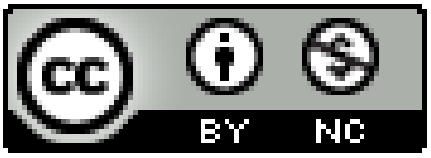
En primer lugar, agradecer a todos los profesores y, en especial a mi tutor Vicente Blanco Pérez, por la formación que me han procurado durante estos años, además de la ayuda que me han otorgado para facilitar la elaboración de este trabajo de fin de grado. En segundo lugar, quiero agradecer a familiares y amigos que me ha apoyado y dado ánimos para seguir adelante. Por último, me gustaría agradecer a mis padres, su apoyo incondicional y confianza en mí es lo que me ha hecho llegar hasta aquí.

De nuevo, muchas gracias.

La Laguna, 11 de septiembre de 2020

Licencia

* Si quiere permitir que se compartan las adaptaciones de tu obra y NO quieres permitir usos comerciales de tu obra indica:



Resumen

La Computación de Alto Rendimiento está ganando fuerza en una gran variedad de campos. Podemos encontrar el uso de esta rama de la informática en nuestro día a día, desde aplicaciones móviles y redes sociales, hasta la previsión meteorológica y el desarrollo de medicamentos. Es por eso que esta herramienta debe usarse correctamente, buscando los mejores resultados en un tiempo razonable. Para lograrlo, los métodos y herramientas de Análisis del Rendimiento que se encargan de estudiar el comportamiento de los programas que se ejecutan en este tipo de sistemas son de una importancia capital, con el fin de optimizar el proceso de desarrollo y los algoritmos implementados.

Los objetivos principales de este proyecto buscan, por un lado, mejorar la salida de la herramienta CALL con la que se obtienen distintas métricas de rendimiento de un algoritmo y, por otro, realizar a modo de ejemplo un análisis de rendimiento con dos algoritmos distintos pero que buscan resolver el mismo problema. Esta comparativa pretende demostrar la importancia de la visualización de los datos y del estudio del rendimiento de los algoritmos y aplicaciones en Computación de Altas Prestaciones.

Palabras clave: Computación de Alto Rendimiento, Análisis del Rendimiento, algoritmos paralelos, computación en paralelo, visualización, XML, JSON, Matplotlib, Python, métricas.

Abstract

High Performance Computing (HPC) is gaining strength in a great variety of fields. We can find the use of this type of computing technology in many usual software applications: from mobile applications and social networks, to weather forecasting and pharmaceuticals development for instance. For this reason, HPC technology must be used correctly, looking for the best results and performance. To achieve this goal, Performance Analysis methods and tools used to study the behavior of the programs executed in this kind of systems have a great importance. These methodologies help to optimize the development process and to implemented better algorithms.

Two main goals are stated in this proyect: first, to improve the output of the instrumentation tool CALL, a tool with the ability to obtain different performance metrics of an algorithm. JSON, a modern data format is used for this purpose. Second, to develop a set of python scripts to analyze the performance data obtained for two different algorithms but seeking to solve the same problem. This comparison aims to demonstrate the importance of visualizing data and studying the performance of algorithms and applications in HPC.

Keywords: High Performace Computing, Performance Analysis, parallel algorithms,parallel computing, visualization, XML, JSON, Matplotlib, Python, metrics

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Computación de Alto Rendimiento

El campo de la computación de alto rendimiento o High Performance Computing(HPC) nace de la necesidad de resolver problemas computacionales muy complejos, comúnmente problemas del campo de la ciencia y la ingeniería, como pueden ser simulaciones o el análisis de grandes cantidades de datos, con mayor velocidad y mejores prestaciones. Se suele usar este tipo de tecnologías en una gran variedad de campos como pueden ser el diseño asistido por ordenador para aviones o estructuras mecánicas, contenidos digitales, análisis financiero, modelos climáticos o transporte y logística entre otros.

Un ejemplo del uso de la computación de alto rendimiento y de su relevancia en sectores tan importantes como el de la Salud, es que hoy en día se está utilizando esta rama de la ingeniería informática para entender cómo funciona y se transmite el virus del covid-19 [22] mediante simulaciones, con el propósito de elaborar una vacuna lo más rápido posible.

Pero la supercomputación también está presente en nuestro día a día, ya que detrás de las aplicaciones y redes sociales más conocidas y las cuales usamos a diario, se encuentra este tipo de tecnología. Un claro ejemplo puede ser la red social Facebook que con más de 2.2 billones de usuarios activos al mes[1], se puede considerar una de las redes sociales más grande del mundo y una de las que más información posee sobre sus usuarios. Se estima que esta famosa aplicación genera alrededor de 4 Petabytes de datos por día, por lo que en un solo año habrá almacenado más de 1400 Petabytes (1400000 Terabytes) de información. Para tratar esta enorme cantidad de datos Facebook cuenta con distintos grupos de Big Data y varios superordenadores, como el que se encuentra en su sede principal en Estados Unidos y que cuenta con 60.512 cores[19].

Como veremos más adelante, la computación de alto rendimiento no se basa únicamente en la cantidad de cores o en la potencia del hardware, también se debe tener en cuenta el software que va a trabajar sobre dicha máquina, el cual

debe estar diseñado con el propósito de obtener el máximo rendimiento posible. Para ello, la computación de alto rendimiento hace uso de la denominada computación paralela, sobre la cual hablaremos con más detalle a continuación.

1.2. Computación en paralelo

A lo largo de los últimos años, una de las principales metas en el mundo de la tecnología ha sido conseguir los mejores resultados en el menor tiempo posible. En el área de la computación, y más específicamente, en lo que se refiere a hardware, esto se traduce en computadores cada vez más veloces, con un mayor número de procesadores y capaces de trabajar en paralelo.

Sin embargo, para poder aprovechar de la mejor manera toda esta potencia computacional que aporta el hardware, es necesario diseñar las herramientas software adecuadas para sacar el máximo beneficio.

Por ello, en sectores como el de la computación de alto rendimiento, donde son tan importantes tanto los resultados como el tiempo de ejecución, se utiliza la técnica de la computación paralela para mejorar su desempeño.

Dicha técnica consiste en la utilización de múltiples elementos de proceso para dar solución a un problema computacional complejo. Para ello, se divide dicho problema en tareas de menor complejidad. Cada una de estas tareas se descompone a su vez, en diversas instrucciones que serán ejecutadas en diferentes procesadores de manera simultánea e independiente y cuyos resultados serán combinados al terminar.

En la computación en paralelo se hace uso tanto de computadoras paralelas (procesadores multi-núcleo, multi-procesadores, clusters, etc), como de técnicas de programación en paralelo.

Este paradigma de la programación proporciona métodos de programación específicos que tienen en cuenta el modelo computacional en paralelo. Se vale del uso de lenguajes paralelos (C, C++, Fortran, etc) y entornos de programación paralela, proporcionando al usuario del espacio y herramientas adecuadas para desarrollar y aplicar las técnicas propias de esta rama de la informática.

Estos últimos, permiten tratar con grandes tareas de computación mediante la paralelización, es decir, ejecutando dichas tareas por partes, de manera más rápida y eficaz.

1.3. Análisis del Rendimiendo

Debemos tener en cuenta que el desarrollo y uso de un superordenador conlleva un alto consumo energético y por tanto, un elevado coste monetario y medioambiental. Es por ello por lo que las aplicaciones que hagan uso de este tipo de hardware han de estar lo más optimizadas posibles, con el fin de aprovechar el máximo rendimiento de la máquina, obteniendo los mejores resultados y consumiendo lo menos posible. Es entonces donde se hace necesario el análisis del rendimiento en computación de altas prestaciones.

El análisis del rendimiento nos permite evaluar el comportamiento de un programa y predecir el coste computacional de la ejecución del mismo. A lo largo de los años se han creado herramientas cuyo propósito es obtener datos sobre el aprovechamiento que se está haciendo de los recursos disponibles. Estos datos pueden ser trazas de ejecución, recursos de tiempo y memoria consumidos por el algoritmo, datos estadísticos de los eventos observados, etc. Una vez obtenidos estos datos, podemos determinar si el algoritmo ofrece las prestaciones adecuadas o si el sistema paralelo sobre el cual estamos trabajando está siendo aprovechado.

Existe una gran variedad de herramientas que han sido desarrolladas con el propósito de realizar una investigación del comportamiento del programa en base a la información obtenida durante la ejecución del mismo. Estas herramientas nos permiten identificar aquellas partes del código ineficientes o que pueden ser ejecutadas por elementos de procesos de propósito específico (GPU's)

Algunos ejemplos de herramientas para el análisis del rendimiento son Vampir 9.9[8], Paraver[6], TAU[7] o EZTrace[5], todas ellas proporcionan una interfaz donde podemos visualizar fácilmente el comportamiento de cada uno de los procesadores. Sin embargo, no permiten realizar modelos analíticos. Los modelos analíticos nos permiten crear un modelo tanto de la arquitectura utilizada como del algoritmo mediante expresiones analíticas de forma que podamos analizar el programa de forma independiente al sistema sobre el cual se está ejecutando. Esto es algo que sí nos permite realizar el sistema CALL, herramienta con la cual trabajaremos para obtener los datos sobre la ejecución del experimento y de la cual hablaremos más adelante.

En definitiva, podemos concluir que el propósito del análisis del rendimiento es aumentar la productividad optimizando los recursos y herramientas disponibles, determinado el correcto funcionamiento de la aplicación frente al problema a resolver.

Capítulo 2

Recursos disponibles

En este capítulo se describen las herramientas que teníamos disponibles para la obtención de las medidas de rendimiento. Haremos uso del sistema CALL en conjunto con el drive PAPI para extraer información sobre el comportamiento del procesador durante la ejecución y de la biblioteca de paso de mensajes MPI para crear un algoritmo que se ejecute en paralelo.

2.1. Sistema CALL

El sistema CALL permite analizar el comportamiento de nuestro código durante su ejecución. Está formado por un traductor (call), una biblioteca de tiempo de ejecución (cll.h) y un analizador estadísticos (llac), además de librerías para la medición del rendimiento como PAPI de la cual hablaremos más adelante. CALL puede ser utilizado para analizar programas tanto secuenciales como en paralelo escritos en BSPlib, PUB, PVM, MPI y OpenMP [21].

Se trata de una herramienta de monitorización y modelado que nos permite especificar qué parte del código queremos analizar y qué eventos se observarán haciendo uso de los pragmas específicos de CALL. EL sistema será el encargado de instrumentar el código para posteriormente obtener los valores necesarios para realizar el análisis de validez del modelo y determinar el comportamiento del programa.

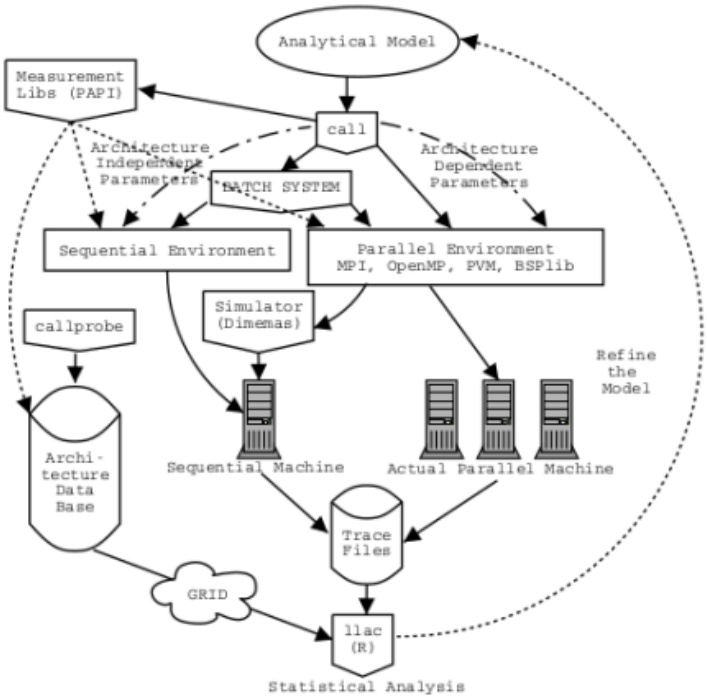


Figura 2.1: Esquema de funcionamiento de la herramienta CALL

2.1.1. Performance Application Programming Interface PAPI

Para acceder a los eventos del procesador, CALL utiliza la librería PAPI (Performance Application Programming Interface) [23] la cual consta de un estándar de eventos relevantes para medición y ajuste del rendimiento de las aplicaciones. PAPI proporciona una manera fácil de acceder a los contadores de los microprocesadores ayudando al análisis, modelado y ajuste del rendimiento de los programas.

PAPI ofrece dos interfaces para el hardware del contador: una interfaz de alto nivel para la obtención de medidas simples, así como para iniciar, detener y leer algunos conjuntos de datos. La interfaz de bajo nivel es completamente programable y proporciona opciones más sofisticadas para controlar los contadores, como el acceso a todos los eventos y el establecimiento de umbrales de interrupción.

PAPI puede realizar múltiples mediciones de diferentes eventos del hardware y relacionarlos entre sí para un mejor análisis del rendimiento.

2.1.2. Message Passing Interface MPI

Uno de los modelos de programación paralela más utilizado es el paradigma de paso de mensajes. Esta técnica permite la comunicación entre procesos mediante el envío y recepción de mensajes. Cada proceso utiliza su propia memoria local y se comunica con el resto de procesos mediante una red de interconexión.

MPI (Interfaz de Paso de Mensajes - Message Passing Interface) es un estándar que define cómo debe ser la sintaxis y la semántica de la programación por paso de mensaje. Proporciona una librería para aplicaciones en C, C++ y Fortran, la cual define funciones para comunicar datos entre procesos.

Esta librería fue desarrollada en consenso por el MPI Forum [16], conformado por 40 organizaciones y alrededor de 60 expertos, con la intención agrupar las mejores características de los sistemas de paso de mensajes, mejorarlas y estandarizarlas. De esta manera, en 1994 MPI se convierte en la primera librería de paso de mensajes estándar y portable que permite desarrollar programas que pueden ser fácilmente migrados a diferentes computadores paralelos .

Capítulo 3

Problema y solución

Para este proyecto se ha planteado el formato de salida **XML** por **JSON** modificando así el sistema **CALL** para obtener los datos de trazas de rendimiento y de esta manera utilizarlos con las librerías de análisis y visualización disponibles en Python.

En la primera parte del proyecto debíamos cambiar la salida de la herramienta de instrumentación CALL de formato **XML** a **JSON** para posteriormente, realizar gráficos que ilustren el rendimiento de los programas analizados.

JSON (JavaScript Object Notation - Notación de Objetos de JavaScript) [11] es un formato de texto ligero diseñado para el intercambio de datos. Permite almacenar los datos de manera sencilla y ordenada y se dispone de herramientas y librerías para su procesado. Está basado en un subconjunto del lenguaje de programación **JavaScript**, aunque se considera independiente del lenguaje ya que puede ser utilizado en diferentes lenguajes de programación.

Un objeto **JSON** está normalmente formado por dos estructuras:

- Una colección de objetos clave-valor.
- Una lista ordenada de valores.

Los principales motivos para realizar este cambio en el formato fueron:

- Formato más simple, por lo que es más fácil de analizar que un fichero en **XML**, sobre todo si es necesario el acceso por parte de un humano.
- Mayor velocidad de procesamiento.
- Generación de archivos de menor tamaño.

En un primer momento se planteó la idea de traducir la salida obtenida en XML a JSON, sin embargo, al final se optó por cambiar el código encargado de generar el fichero XML para que en su lugar se genere la salida en formato JSON directamente.

En la segunda parte del proyecto, se debía aprovechar estos datos para realizar un estudio del rendimiento y determinar el comportamiento de un algoritmo ejemplo. Desde un principio se tuvo claro el uso del lenguaje de programación **Python** [18] para conseguir este propósito debido a la gran cantidad de librerías para la visualización y el cálculo numérico que posee.

El objetivo era utilizar dichas librerías para crear gráficos donde poder apreciar más claramente el comportamiento del programa analizado en función del tiempo de ejecución, el número de procesadores utilizado u otras métricas de interés. Conseguido esto, podemos concluir bajo qué condiciones se obtiene un rendimiento óptimo del programa.

3.1. Herramientas utilizadas

3.1.1. Librería Json-c

Para realizar la salida en JSON se utilizó la librería **Json-C para C** [10] la cual proporciona diversos métodos para representar información en formato JSON. Json-C implementa un modelo de objeto de recuento de referencias que permite construir fácilmente objetos JSON en C, generarlos como cadenas con formato JSON y analizar cadenas estas cadenas como si fueran objetos en C.

Json-C permiten crear objetos JSON en función del tipo de dato a representar, como enteros o cadenas, que luego podremos añadir a un objeto padre que se convertirá en un árbol de referencias de objetos JSON. En los códigos 3.1 y 3.2 se muestra el uso de la librería y el resultado que se obtiene.

Para poder utilizar esta librería solo hay que instalarla siguiendo los pasos de la documentación[10] e incluir la cabecera “json-c/json.h”.

```
1  #include "json-c/json.h"
2
3  struct json_object *json;
4
5  json_object *string = json_object_new_string("Prueba");
6  json_object *entero = json_object_new_int(10);
7
8  json_object_object_add(json, "PRUEBA", string);
9  json_object_object_add(json, "ENTERO", entero);
10
11
12 printf(json_object_to_json_string_ext(json,
13                                        JSON_C_TO_STRING_PRETTY));
```

Listado 3.1: Ejemplo json-c

```
1 {"PRUEBA": "prueba",
2  "ENTERO": 10
3 }
```

Listado 3.2: Salida obtenida

3.1.2. Matplotlib

Como se ha dicho anteriormente, Python cuenta con una gran cantidad de librerías para la visualización y manipulación de datos. Una de las más conocidas y utilizadas para este propósito es **Matplotlib** [15]. Se trata de una biblioteca de visualización multiplataforma hecha para trabajar con Python y su extensión **Numpy** [13]. Cuenta con una gran variedad de gráficos en 2D (histogramas, espectros de potencia, gráficos de barras, gráficos de error, diagramas de dispersión, boxplots, etc.) pero además, existen extensiones con las cuales podemos crear gráficos más avanzados, como gráficos en 3D.

Python con Matplotlib nos permite crear varios gráficos a la vez, añadir líneas de tendencias y la posibilidad de personalizar la salida añadiendo leyendas, títulos, etc.

Para realizar el análisis de rendimiento que veremos en el **Capítulo 4** se utilizaron dos tipos de gráficos de los disponibles en Matplotlib:

- **Gráfica lineal** [3]: este tipo de gráficas es la más simple de elaborar con Matplotlib. Nos permite ver rápidamente el cambio de tendencia en los datos representados por puntos en la gráfica y unidos por segmentos lineales (Figura 3.1). Si bien este tipo de gráficos se suele utilizar para ver la progresión de variables cuantitativas a lo largo del tiempo, en nuestro caso, lo utilizaremos para representar el tiempo que se tarda en ejecutar el programa en función del número de procesadores.

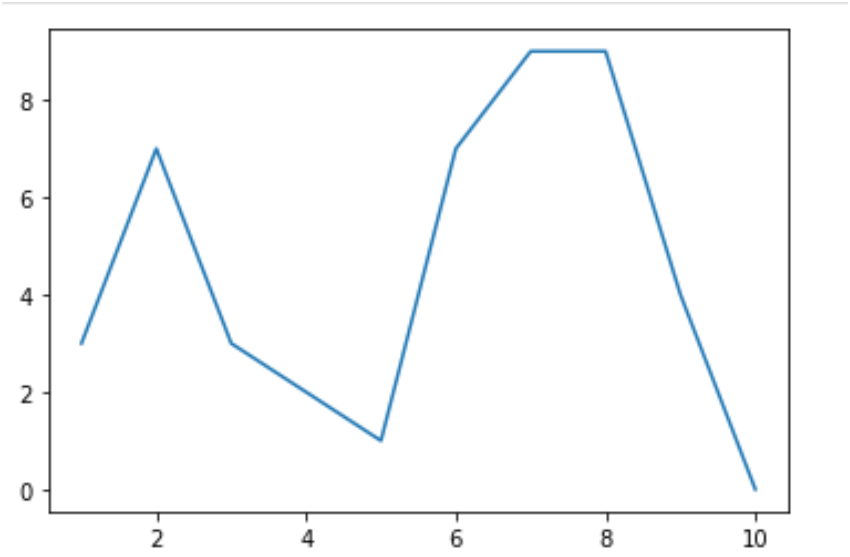


Figura 3.1: Ejemplo de gráfica lineal en Matplotlib

- **Boxplot o diagrama de caja** [2]: es una herramienta de visualización estadística que muestra el resumen de cinco números de un dato, es decir, el mínimo, el primer cuartil, la mediana, el tercer cuartil y el máximo, pudiendo representar de igual forma valores atípicos. Este tipo de diagrama nos permite observar cómo se distribuyen los valores en los datos, permitiéndonos identificar los valores medios, la dispersión y la asimetría en el conjunto de datos (Figura 3.2).

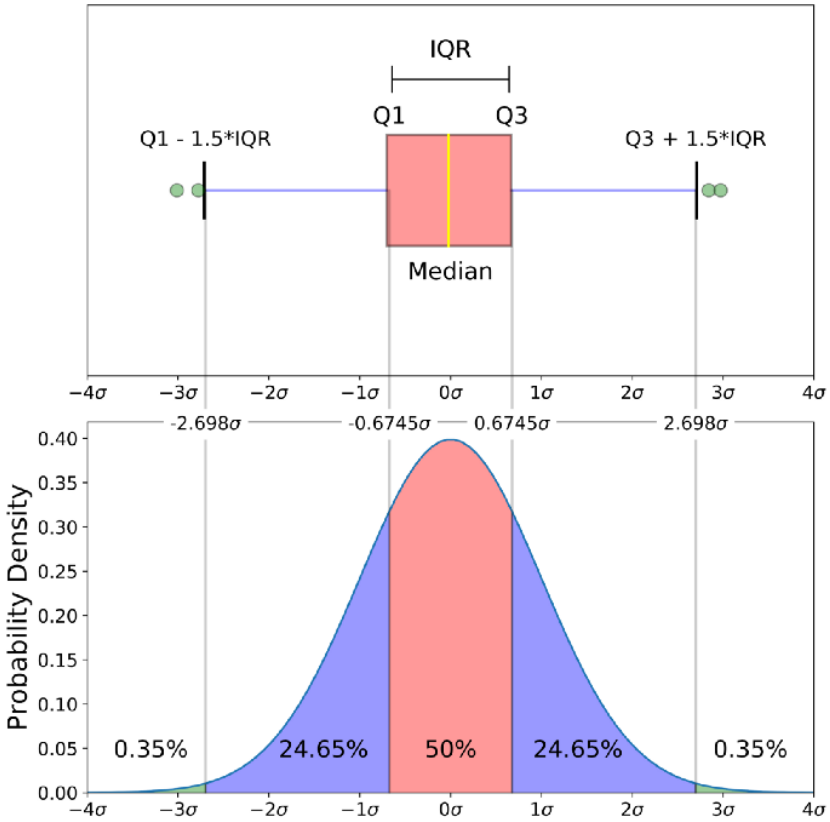


Figura 3.2: Comparación entre una gráfica de caja y una distribución normal

Para crear los gráficos anteriormente nombrados importamos el **módulo pyplot** [17] de matplotlib y hacemos uso de las funciones **"plot"** y **"boxplot"** para crear el gráfico lineal y el diagrama de cajas respectivamente.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  plt.figure(1)
5  plt.plot([1,2,3,4], [5,10,15,20], linestyle='-', marker='o')
6  plt.title('Ejemplo Gráfico Lineal')
7  plt.xlabel("Procesadores")
8  plt.ylabel("Tiempo de ejecución")
9
10 np.random.seed(10)
11 data = np.random.normal(100, 20, 200)
12 plt.figure(2)
13 plt.boxplot(data)
14 plt.title('Ejemplo Diagrama de cajas (Boxplot)')
15 plt.xlabel("Procesadores")
16 plt.ylabel("Tiempo de ejecución")
17
18 plt.show()
```

Listado 3.3: Ejemplo de visualización de Python

Como podemos observar en el ejemplo del listado 3.3, debemos pasar a ambas funciones los valores que queremos representar. En el caso del gráfico lineal debemos proporcionarle dos arrays con los valores de las coordenadas “x” e “y”. En la figura 3.3 se puede ver el gráfico resultante:

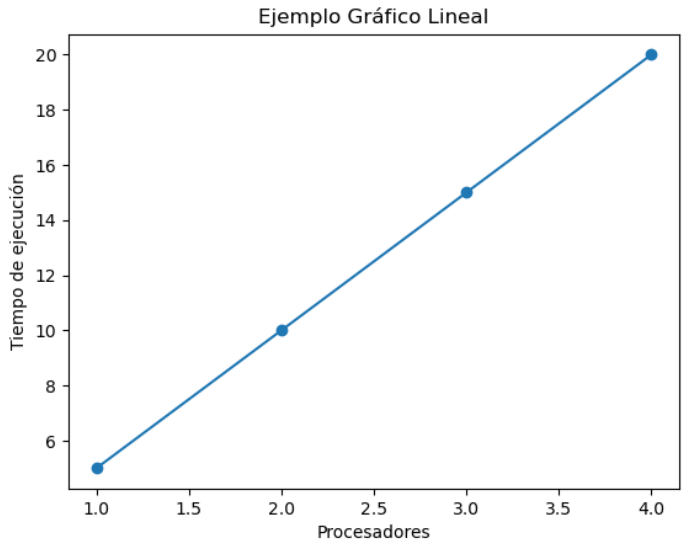


Figura 3.3: Ejemplo de Gráfico Lineal con Matplotlib

En cuanto al gráfico de cajas o Boxplot, debemos proporcionar una colección de valores por cada cuadro que queramos representar. Esta colección puede ser una lista, una tupla o una matriz. Una vez tengamos todas las colecciones para cada caja, debemos combinar todas está en una sola lista, es decir, una lista de colecciones. Hecho esto, obtendremos un diagrama como el siguiente (Figura 3.4):

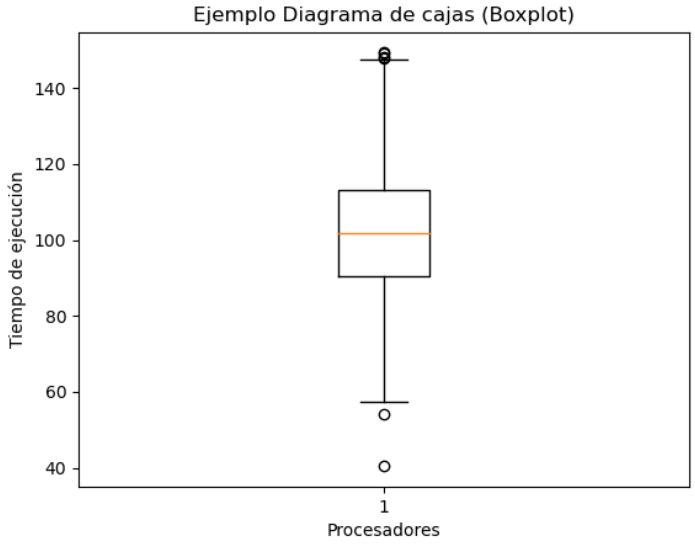


Figura 3.4: Ejemplo de Boxplot con Matplotlib

3.1.3. Otras herramientas

A parte de las herramientas anteriormente nombradas, fue necesario el uso de otras librerías de python con el propósito de hacer un correcto uso de los datos.

Librería Numpy

Numpy [13] es una librería open-source de Python que contiene funciones matemáticas de alto nivel como rutinas trigonométricas, estadísticas y algebraicas además de generadores de números aleatorios. Proporciona un objeto de matriz multidimensional de alto rendimiento y herramientas para trabajar con estas matrices.

Librería Json

La librería Json [12] permite analizar objetos de ficheros o strings y convertirlos en listas o diccionarios en Python y viceversa. Esta librería la usaremos varias veces para cargar el contenido de los ficheros json en los scripts creados para visualizar los datos extraídos.

Librería Os

Este módulo [14] permite acceder a las funcionalidades dependientes del Sistema operativo, sobre todo aquellas que nos permiten manipular ficheros y directorios (leer y escribir)

3.2. Metodología

3.2.1. De XML a Json

Para obtener la salida en formato JSON fué necesario modificar las funciones encargadas de generar el fichero XML, las cuales se encontraban principalmente en los ficheros **'cll_exp.h'** [apéndice B] y **'cll_MPI.h'** [apéndice C]. También fue necesario modificar el fichero **'cll.PAPI_EVENTS.h'** para obtener correctamente algunas de las métricas proporcionadas por PAPI, como el tiempo de ejecución del experimento **(PAPI_REAL_USEC)** o el número total de fallos de caché **(PAPI_L2_TCM)**.

Como podemos observar en el apéndice **A**, el documento XML está conformado por distintos elementos, los cuales a su vez, cuentan con múltiples atributos para almacenar los datos. Entre los elementos que conforman el árbol XML, podemos distinguir los siguientes:

- **<cll_data>**: se trata del elemento padre. De él surgen el resto de ramas o elementos hijos que componen el fichero XML. Dentro de esta etiqueta podemos distinguir una serie de atributos, como la versión de CALL que se está utilizando, el nombre del programa ejecutado o el sistema sobre el cual se está ejecutando.

- **<cll_experiment>**: esta etiqueta contiene todos los datos sobre el experimento. Las primeras líneas se tratan de atributos que nos describen el nombre del experimento, los observables que se están utilizando, el número de test realizados, etc. A continuación, se encuentran los elementos **<machines>** y **<headers>**, los cuales nos indican el nombre de las máquinas o nodos que se utilizan durante el experimento y los identificadores de las medidas obtenidas. Por último, tenemos la etiqueta **<sample>**, dicha etiqueta almacena todos los datos observados durante la ejecución del experimento por cada procesador. El orden en el que estas medidas se encuentran representadas en el fichero coinciden con el orden de los identificadores de la etiqueta <headers>[Listado 3.4].

```
    <headers>
      <h>CPU</h>
      <h>NCPUS</h>
      <h>PAPI_REAL_USEC  </h>
      <h>   PAPI_L2_TCM  </h>
    </headers>

    <sample>
      <row>
        <c>0</c>
        <c>2</c>
        <c>         710853  </c>
        <c>           1308  </c>
      </row>
```

Listado 3.4: Identificadores y medidas obtenidas

Teniendo en cuenta lo anterior, se fueron creando los objetos JSON para cumplir con la estructura ya establecida. De igual manera, se modificaron y crearon las funciones necesarias para almacenar los datos en dichos objetos.

En primer lugar se creó el objeto **'json'**, el cual será el objeto principal, en el incluiremos todos los demás objetos para crear el json final.

Para introducir los primeros datos sobre el experimento, se modificó la función **'cll_print_header'**. Como se puede ver en el **listado 3.5** esta función, al igual que las otras funciones ya existentes y que eran las encargadas de crear el fichero XML, hacían uso de **'fprintf()'** para escribir los datos en el fichero.

```
#define cll_print_header(experiment,driver) {  \
  if (!cll_first_report) { \
    struct utsname machine; \
    \
    uname(&machine);  \
    fprintf(cll_fdata, "<cll_data"); \
    fprintf(cll_fdata, "\n\tCALL_VERSION=\"%s\"", cll_version); \
    fprintf(cll_fdata, "\n\tPROGRAM=\"%s\"",      cll_programname); \
    ....
  } \
}
```

Listado 3.5: Función cll_print_header del fichero cll_exp.h con XML

```
40 #define cll_print_header(experiment,driver) { \
41   if(!cll_first_report) { \
42     struct utsname machine; \
43     uname(&machine); \
44 
45     json_object *call_version = json_object_new_string(cll_version);\
46     json_object *program = json_object_new_string(cll_programname);\
47 
48     ....
49 
50     json_object_object_add(json, "CALL_VERSION", call_version);\
51     json_object_object_add(json, "PROGRAM", program);\
52 
53     ....
54 
55     }
56 }
```

Listado 3.6: Función cll_print_header del fichero cll_exp.h con JSON

En el **listado 3.6** se puede ver como se crearon los objetos JSON para cada uno de los atributos. A diferencia del código anterior, que creaba el fichero XML escribiendo directamente en él, con JSON le podemos indicar el tipo de dato con el cual almacenaremos el atributo. Esta cualidad nos facilitará el trabajo posterior para extraer los datos del fichero JSON y hacer la visualización con Python. Esto lo podemos observar mejor en la función **'cll_print_experiment_samples'**, la cual es la encargada de escribir los datos obtenidos con CALL en el fichero.

Como se puede observar en el **listado 3.7**, para almacenar los datos de las mediciones de cada procesador se creó un vector de objetos JSON(**'sample'**).Dado que las medidas con las que vamos a trabajar se tratan de números enteros, crearemos un objeto JSON de tipo entero (**'instance'**) para incluir los distintos valores obtenidos de los observables en dicho vector. A continuación, incluiremos este vector en otro llamado **'samples'**, creando así una matriz con todas las medidas.

```
177 #define cll_print_experiment_samples(numidents, instances, \
178                                      numberoftests, processor, \
179                                      ncpus) { \
180   int cll_i, cll_k; \
181   for (cll_i = 0; cll_i < (numberoftests); cll_i++){ \
182     sample = json_object_new_array();\
183     cll_print_exp_fixed_fields(processor,ncpus);\
184     for(cll_k = 0; cll_k <numidents; cll_k++){\
185        json_object *instance = \
186        json_object_new_int(instances[cll_i].id[cll_k]);\
187        json_object_array_add(sample,instance);\
188     }\
189     ....
190   } \
191 }
```

Listado 3.7: Función cll_print_experiment_samples del fichero cll_exp.h

Por último, se creó la función **'end_of_json'** para incluir todos los datos de los experimentos en el vector **'cll_experiment'**, que hará la misma función que el elemento '<cll_experiment>' del fichero XML. De la misma forma insertamos este último vector en el objeto 'json' finalizando así el árbol de referencias que será el JSON final.

```
58 #define end_of_json() {\
59   json_object_object_add(cll_experiment, "sample", samples);\
60   json_object_object_add(json, "cll_experiment", cll_experiment);\
61   fprintf(cll_fdata,
62           json_object_to_json_string_ext(json,
63                                          JSON_C_TO_STRING_PRETTY));\
64 }
```

Listado 3.8: Función end_of_json del fichero cll_MPI.h

Todas estas funciones para crear el fichero JSON son invocadas desde el fichero cll_MPI.H en la función **'cll_report'**.

```
65 #define cll_report(experiment)                          \
66   {                                  \
67     ....
68     if (!myid) { \
69       cll_open_file(); \
70       start_json();\
71    cll_print_header((experiment),"MPI"); \
72
73    ...
74
75       cll_print_exp_fields_titles((experiment));\
76       cll_print_experiment_samples((experiment).numidents, \
77          (experiment).instance, (experiment).numtests, \
78          myid, numproc);\
79       for(proc = 1; proc < numproc; proc++) {  \
80         ....
81         cll_print_experiment_samples((experiment).numidents, \
82             (experiment).instance, cll_numtests[proc], \
83             proc, numproc); \
84         ....
85       } \
86       end_of_json();\
87      ....
88 }
```

Listado 3.9: Función cll_report del fichero cll_MPI.h

3.2.2. Visualización con Python

Haciendo uso de Python y de la librería Matplotlib se crearon gráficos con los que visualizar rápidamente los datos obtenidos con CALL. Se crearon múltiples gráficos con diferentes medidas sobre el rendimiento, sobre las cuales hablaremos en el **Capítulo 4**.

Para extraer todos los datos obtenidos del experimento fue necesario leer cada uno de los ficheros y seleccionar únicamente los valores del experimento **[Listado 3.10]**.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import pandas as pd
4  import json, os
5
6  path_to_json_1 = 'data/other_data/'
7
8  json_files_1 = [pos_json for pos_json in
9                  os.listdir(path_to_json_1) if
10                   pos_json.endswith('40.json')]
11
12 sample_1 = []
13 num_tests_1 = []
14
15 for js in (json_files_1):
16     with open(os.path.join(path_to_json_1, js)) as json_file:
17         json_data = json.load(json_file)
18         sample_1.append(json_data['cll_experiment']['sample'])
19         num_tests_1.append(json_data['cll_experiment']['NUMTESTS'])
20
21
22         number_cpus_1 = []
23     for p in sample_1:
24         for i in p:
25             number_cpus_1.append(i[1])
```

Listado 3.10: Lectura de los ficheros JSON

Hecho esto, separamos los datos por métricas y los guardamos en diferentes vectores. Con estos vectores procederemos a calcular los distintos puntos que representaremos en ambas gráficas **[Listado 3.11]**. Dichos cálculos se harán siguiendo las fórmulas descritas en el **Capítulo 4**.

```
50
51 cpus_1 = []
52 nums_1 = []
53 tcms_1 = []
54
55 for i in number_cpus_1:
56     cpu_1 = []
57     num_1 = []
58     tcm_1 = []
59
60     for p in sample_1:
61         for j in p:
62             if j[1] == i:
63                 cpu_1.append(j[0])
64                 num_1.append(j[2])
65                 tcm_1.append(j[3])
66
67     cpu_1 = list(dict.fromkeys(cpu_1))
68     cpus_1.append(cpu_1)
69     nums_1.append(num_1)
70     tcms_1.append(tcm_1)
71
72     ....
73     speedup_1 = []
74     efficiency_1 = []
75
76     for i in range(len(parallel_1)):
77         speedup_1.append(sequential_1 / parallel_1[i])
78         efficiency_1.append(sequential_1 /
79           (number_cpus_1[i]*parallel_1[i]))
```

Listado 3.11: Extracción de los datos del experimento

Por último, creamos las gráficas para ambas métricas **[Listado 3.12]**, especificando el número de procesadores y los datos obtenidos de cada uno. Además se incluyó en ambos gráficos una línea para representar el valor ideal de cada métrica.

```
103
104 plt.figure(1)
105 plt.plot(number_cpus,speedup, linestyle='-', marker='o')
106 plt.plot(number_cpus, number_cpus, "r--")
107
108
109 plt.title('Speedup')
110 plt.xlabel('Processors')
111 plt.show()
```

Listado 3.12: Creación de gráfica lineal para el Speedup

Como se comenta en el apartado **4.1.3**, los experimentos se llevaron a cabo con dos algoritmos distintos. Para poder comparar más fácilmente los resultados de ambos algoritmos, se incluyeron los valores obtenidos tras calcular las métricas en una sola gráfica. Esto se efectuó de la siguiente manera:

```
176
177 plt.figure(1)
178 al_1, = plt.plot(number_cpus_1, speedup_1,
179                  linestyle='-', marker='o',
180                  label='Algoritmo 1')
181 al_2, = plt.plot(number_cpus_2, speedup_2,
182                  linestyle='-', marker='o',
183                  label='Algoritmo 2')
184 op, = plt.plot(number_cpus_1, number_cpus_1,
185               "r--", label='Valor ideal')
186
187 plt.title('Speedup')
188 plt.xlabel('Processors')
```

Listado 3.13: Creación de gráfica lineal para el Speedup de ambos algoritmos

Capítulo 4

Análisis de los resultados

4.1. Análisis del rendimiento

Una vez realizado el cambio de la salida a JSON y comprobado que se genera correctamente el fichero con los datos obtenidos de las mediciones, podemos pasar a realizar algunos experimentos y llevar a cabo un análisis de rendimiento a modo de ejemplo.

Los experimentos se realizaron sobre el ejemplo de cálculo del número π siguiendo la **fórmula de Leibniz [4]**. Se realizaron diversos experimentos con distintos parámetros de configuración de MPI. Para configurar los parámetros de ejecución y realizar mediciones con distinto número de procesadores se utilizó la herramienta **qsub [9]**.

Cada experimento se repite 40 veces por procesador.

Tras la ejecución de los experimentos podemos ver el **fichero '.dat'** generado con los datos obtenidos y las métricas en formato JSON **(Apéndice A)**.

4.1.1. Métricas

Como se ha comentado anteriormente, uno de los objetivos principales del análisis del rendimiento es que consigamos los mejores resultados en el menor tiempo posible, sin embargo, no podemos tener solo en cuenta el tiempo que tarda nuestro algoritmo, si no también los recursos que utiliza. "Podemos tener dos algoritmos que resuelvan un mismo problema en el mismo tiempo, pero que uno de ellos use la mitad de los procesadores que el otro. El que use menos procesadores estará usando mejor los recursos que tiene a su disposición" (p.188) [20]. Por ello existen diversos parámetros para determinar la efectividad con la que un algoritmo usa los recursos del hardware, los más utilizados son:

Speedup

Se refiere a la relación entre el tiempo de ejecución de un algoritmo en paralelo con respecto al tiempo que tarda un algoritmo secuencial en resolver el mismo problema, es decir, podemos medir la ganancia de velocidad de un algoritmo frente a otro. Esta métrica la obtenemos al dividir el tiempo secuencial entre el tiempo paralelo **[20, pág 189]** tal y como se expresa en la siguiente ecuación:

$$S(n,p) = \frac{t(n)}{t(n,p)} \tag{4.1}$$

El speedup debe tener un valor entre 0 y *p* (número de procesadores), donde lo ideal sería que fuera igual a *p* [Figura 4.1], aunque podemos encontrarnos casos donde el speedup supere el valor de *p* causando así un speedup superlineal. Esto puede deberse a la gestión de la memoria en el caso del algoritmo paralelo, donde cada procesador trabaja con una porción menor de los datos que en el algoritmo secuencial.

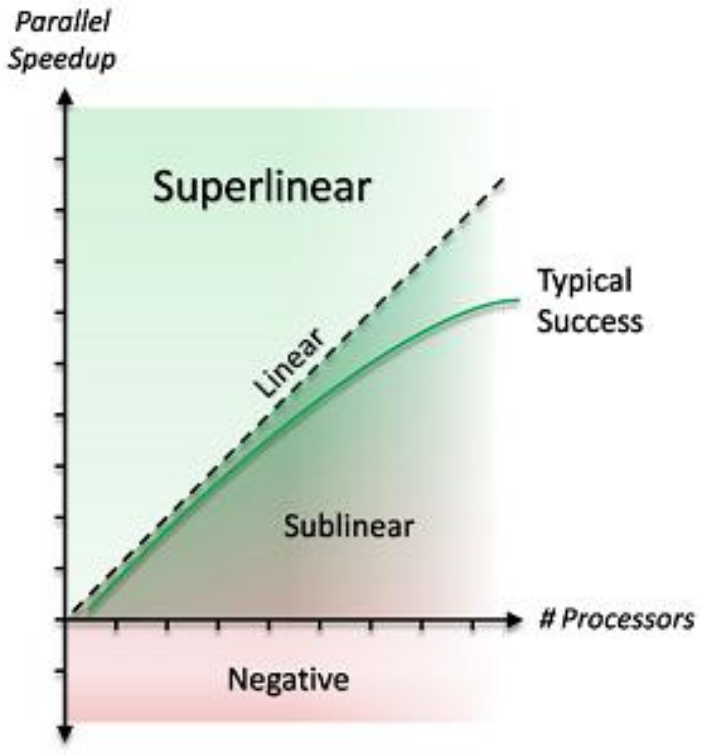


Figura 4.1: Gráfico típico de Speedup en algoritmos paralelos

Eficiencia

Mide la fracción de tiempo durante la cual se utiliza de manera útil un procesador. Podemos obtener esta medida dividiendo el speedup entre el número de procesadores **[20, pág 193]**.

$$E(n,p) = \frac{S(n,p)}{p} \tag{4.2}$$

Esta medida tendrá un valor entre 0 y 1, siendo 1 el valor ideal. Sin embargo, y al igual que en el caso del speedup, podemos obtener una eficiencia superior si se da un speedup superlineal.

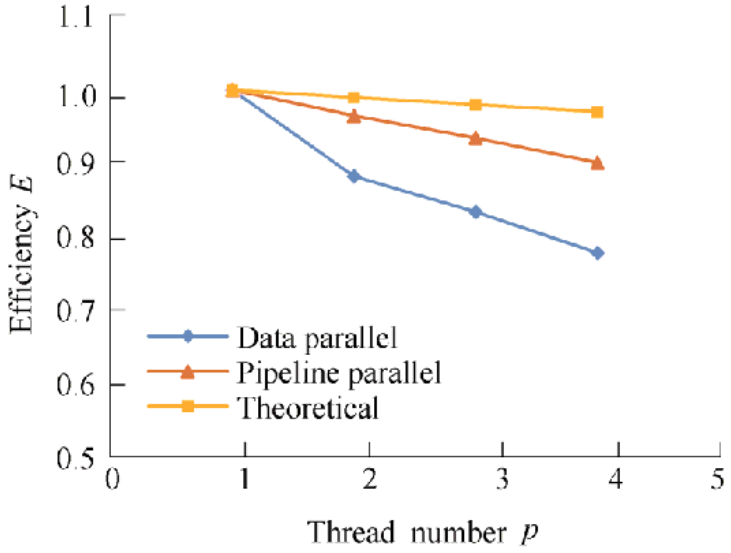


Figura 4.2: Gráfico típico de la Eficiencia en algoritmos paralelos

4.1.2. Cálculo de π

Para probar la instrumentacion con CALL y obtener medidas, se optó por hacer los experimentos con el cálculo de π ya que es un problema altamente paralelizable. A continuación, se explica en qué consiste este cálculo y cómo se llevó a cabo.

Existen distintas formas de realizar el cálculo de π en paralelo, en este caso utilizaremos la relación que existe entre π y la arcotangente de 1 (ecuaciones 4.3 y 4.4). Como la arcotangente se puede calcular mediante una intregral (el área bajo la curva y=$\frac{1}{1+x^2}$ entre los límites 0 y 1 tal y como observamos en la ecuación 4.5), podemos calcular aproximadamente el número π mediante un sumatorio, como el de la ecuación 4.6.

$$arctan(1) = \frac{\pi}{4} \tag{4.3}$$

$$arctan(x) = \int_0^1 \frac{1}{1+x^2} dx \tag{4.4}$$

$$\int_0^1 \frac{1}{1+x^2} dx = 4\ arctan(x)|_0^1 = 4\left(\frac{\pi}{4} - 0\right) = \pi \tag{4.5}$$

$$\pi = \sum_{i=0}^{N-1} \frac{4}{N(1+(\frac{i+0,5}{N})^2)} \tag{4.6}$$

Haremos uso de dos implementaciones paralelas en MPI para el cálculo del número π que pueden encontrarse en el repositorio de este trabajo de fin de grado.

4.1.3. Análisis y resultados

Se realizaron múltiples experimentos con dos algoritmos diferentes para el cálculo de π. El primero (algoritmo 1) utilizando comunicaciones colectivas **[20, pág 85]** y el segundo (algoritmo 2) con comunicaciones punto a punto **[20, pág 78]**. La diferencia entre estos dos métodos de paso de mensajes es que mientras que las comunicaciones punto a punto se limitan a la comunicación entre dos procesos, uno emisor y otro receptor, las comunicaciones colectivas permiten el intercambio de información entre más de dos procesos que coordinan sus operaciones.

Estos dos experimentos se ejecutaron varias veces cambiando la cantidad de nodos cada vez, llegando a utilizar hasta un máximo de 32 procesadores.

Número de procesadores	Resultado	Error
1	3.14159265359042 64	0.0000000000006333
2	3.14159265359002 23	0.0000000000002292
4	3.14159265359021 68	0.0000000000004237
8	3.14159265358961 37	0.0000000000001794
16	3.14159265358977 49	0.0000000000000182
32	3.14159265358977 36	0.0000000000000195

Tabla 4.1: Resultados con diferentes nodos. Algortimo 1

Número de procesadores	Resultado	Error
1	3.14159265359042 64	0.0000000000006333
2	3.14159265359002 23	0.0000000000002292
4	3.14159265359021 68	0.0000000000004237
8	3.14159265358961 37	0.0000000000001794
16	3.14159265358977 54	0.0000000000000178
32	3.14159265358977 36	0.0000000000000195

Tabla 4.2: Resultados con diferentes nodos. Algoritmo 2

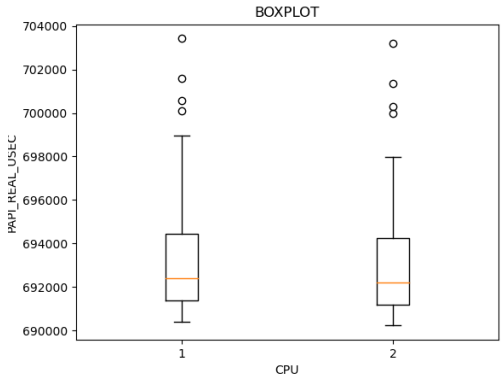
Tiempo de ejecución

En las tablas **4.1** y **4.2** podemos ver los resultados obtenidos. Podemos observar que cada vez que añadimos un procesador vamos obteniendo un error menor y por tanto un mejor resultado en cuanto al cálculo de π. Sin embargo, esto cambia al llegar a 16 procesadores, donde obtenemos el pico más bajo de error en ambos casos. A partir de aquí, a pesar de tener más procesadores dedicados a ejecutar el experimento, el error sigue siendo mayor que el obtenido al utilizar 16 procesadores.

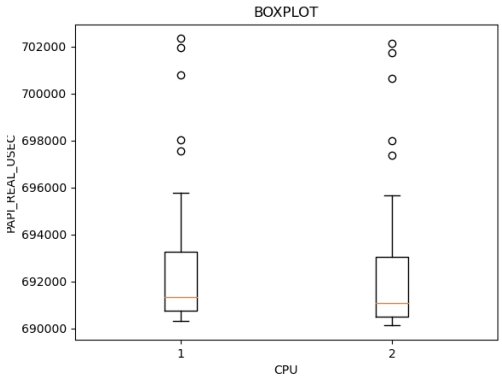
Podemos vislumbrar mejor la diferencia entre ambos algoritmos en las gráficas de la **figura 4.3**. En estas gráficas están representados los tiempos de ejecución de los experimentos con diferente número de procesadores. Los gráficos de la izquierda corresponden a los resultados del algoritmo con comunicaciones colectivas, mientras que los gráficos a la derecha representan los resultados obtenidos con el algoritmo de comunicaciones punto a punto.

Si comparamos el tiempo de ejecución de las figuras **4.3(a)** y **4.3(b)**, observamos que no hay una gran diferencia en los resultados de ambos algoritmos. Podemos decir que de media el segundo algoritmo tarda menos pero cuenta con más valores atípicos, es decir, en ciertos momentos de la ejecución los procesadores tardan más de lo normal.

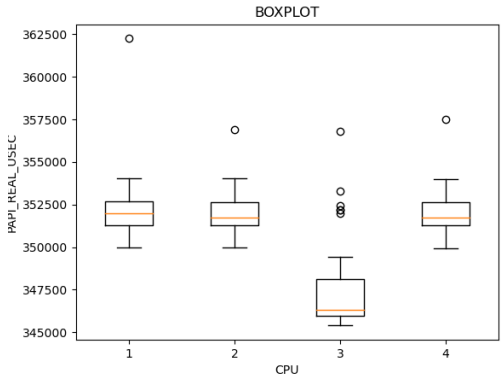
Sin embargo, con el resto de gráficas si podemos ver una clara diferencia entre ambos algoritmos en lo que a tiempo de ejecución se refiere. Por ejemplo,



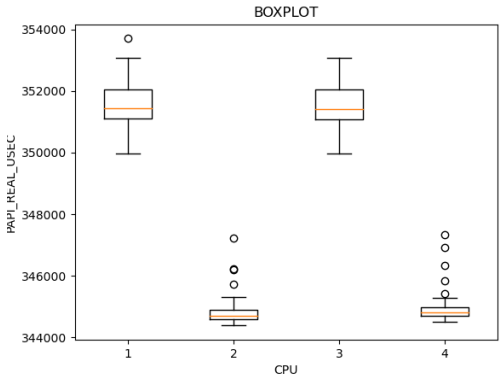
(a) algo1: 2 procs



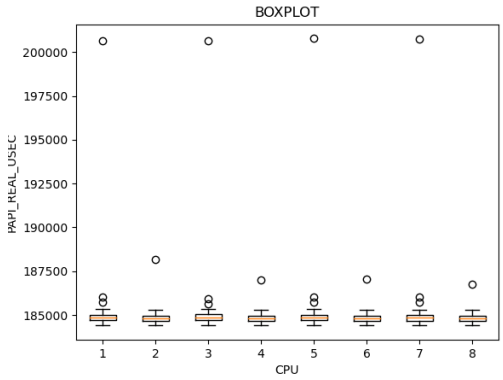
(b) algo2: 2 procs



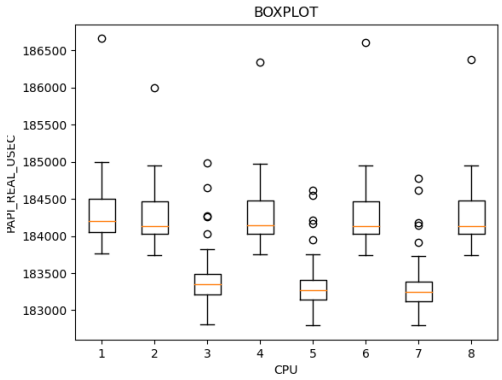
(c) algo1: 4 procs



(d) algo2: 4 procs

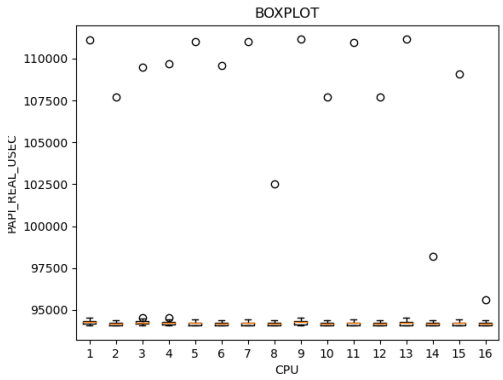


(e) algo1: 8 procs

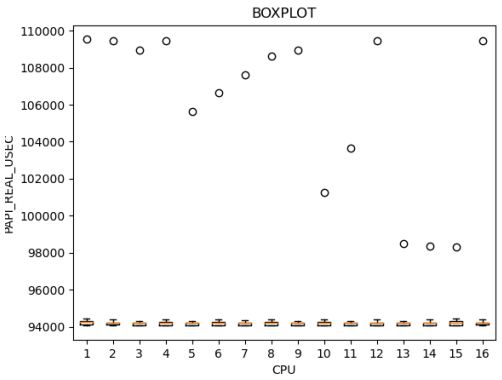


(f) algo2: 8 procs

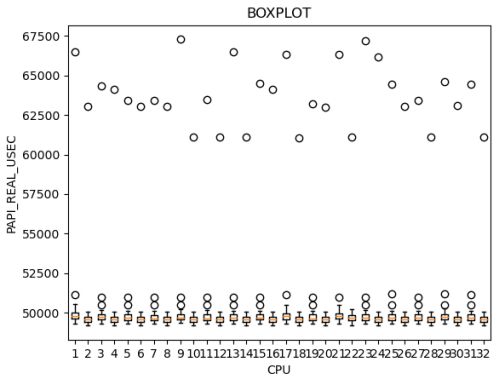
Figura 4.3: Tiempos de ejecución para los algoritmos 1 y 2 de calculo de π



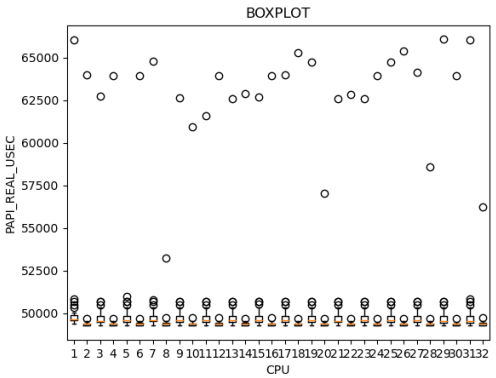
(a) algo1: 16 procs



(b) algo2: 16 procs



(c) algo1: 32 procs

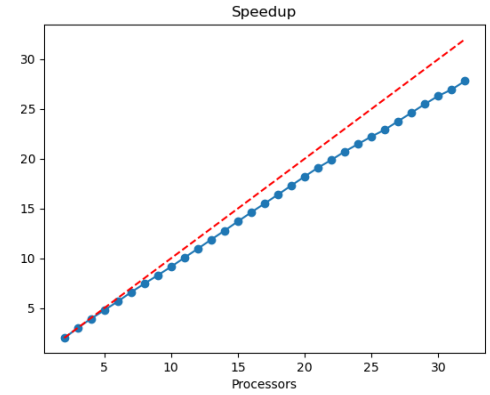


(d) algo2: 32 procs

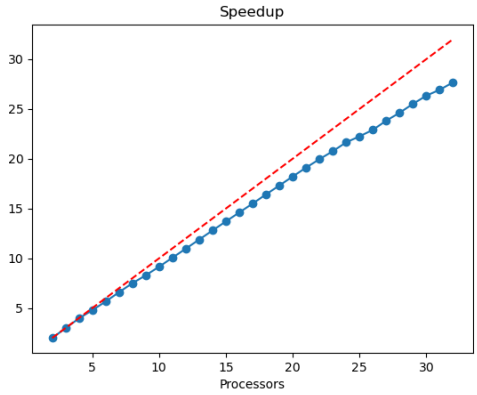
Figura 4.4: Tiempos de ejecucion para los algorimos 1 y 2 de calculo de π

en la gráfica **4.3(c)**, podemos ver como los tiempos de ejecución de los distintos procesadores son relativamente similares, con la excepción de los resultados del procesador 3 que son menores al resto. Por otro lado, en la gráfica **4.3(d)**, podemos apreciar una clara diferencia entre los tiempos de ejecución de los procesadores. Los valores de los procesadores 1 y 3 se asemejan a los valores obtenidos con el primer algoritmo, en cambio los tiempos de ejecución de los procesadores 2 y 4 son mucho menos que el resto y cuentan con más valores atípicos. Esto mismo ocurre cuando utilizamos 8 procesadores (**figuras 4.3(e)** y **4.3(f)**). Esta diferencia en tiempos de ejecución entre los procesadores que ejecutan el algoritmo de comunicación punto a punto probablemente se deba a la jerarquía de la memoria. A pesar de esto, la media en tiempo de ejecución de ambos algoritmos es similar.

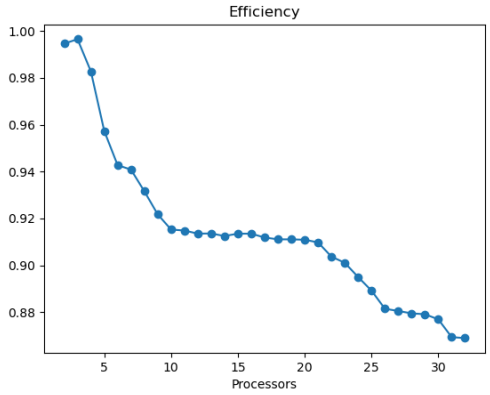
De igual manera, cuando observamos las gráficas de la **figura 4.4**, vemos que los tiempos de ejecución entre los procesadores son parecidos tanto para el algoritmo 1 como para el algoritmo 2. En el caso del algoritmo 2 esto se puede deber a que al ser mayor la cantidad de procesadores el número de comunicaciones también es mayor, por lo que el tiempo de ejecución en estos procesadores que antes tardaban menos ha aumentado.



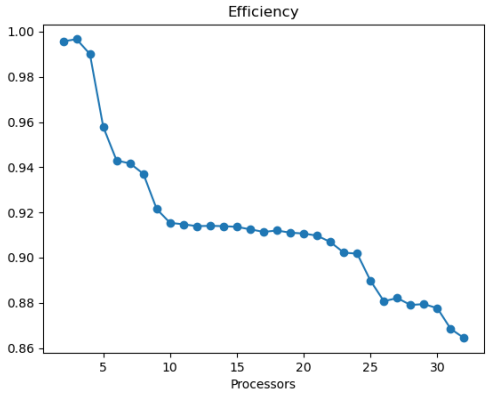
(a) algo1:speedup



(b) algo2:speedup



(c) algo1:efficiency



(d) algo2:effiency

Figura 4.5: Métricas de Speedup y Eficiencia para los algoritmos 1 y 2 de calculo de π

Efectividad del algoritmo

Para hacer un correcto análisis del rendimiento no podemos tener únicamente en cuenta el tiempo de ejecución, sino también la cantidad de recursos necesitados y el uso que se está haciendo de estos. Para realizar esta parte del análisis se crearon diferentes gráficas con las métricas comentadas en el **apartado 4.1.1**.

Las **gráficas 4.5(a) y 4.5(b)** corresponden al speedup o ganancia de velocidad del algoritmo a medida que vamos aumentando el número de procesadores. En ambas gráficas están representados tanto los valores de speedup por procesador como el valor ideal para esta métrica. Podemos comprobar que ambos algoritmos no difieren demasiado el uno del otro. La mayor diferencia entre ambas gráficas se aprecia en los valores de speedup para un número de procesadores entre 25 y 30, donde vemos que ambos algoritmos se alejan de los valores ideales, sin embargo, el algoritmo 1 muestra una tendencia a volver a acercarse a esta línea al llegar a 32 procesadores. Si comparamos los valores del speedup de ambos algoritmos en una sola gráfica (**Figura 4.6**) podemos observar mejor la diferencia.

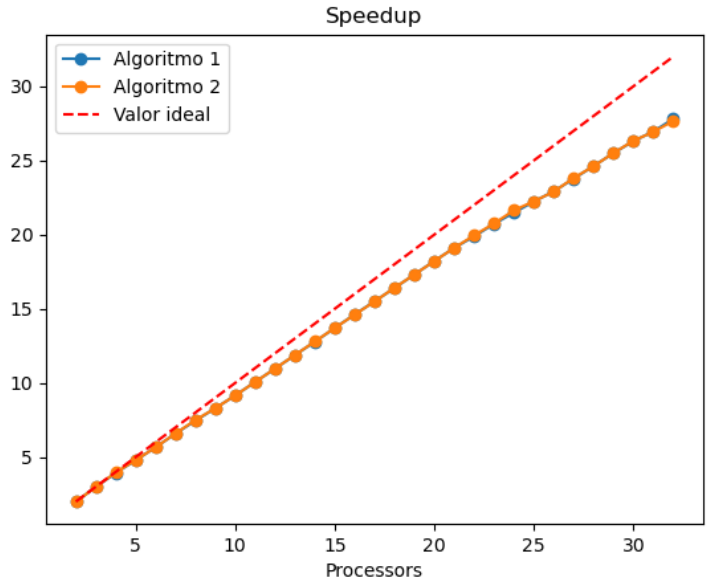


Figura 4.6: Comparación del Speedup de ambos algoritmos

En cuanto a la eficiencia, si bien con ambos algoritmos obtenemos valores muy parecidos, se puede apreciar una mayor diferencia entre las gráficas(**Figuras 4.5(c) y 4.5(d)**). Para empezar, se puede ver como el algoritmo 2 obtiene mejores valores para una cantidad de procesadores entre 2 y 10. A partir de este punto, ambos algoritmos obtienen valores muy similares y estables hasta llegar a 20 procesadores, donde vuelve a caer la eficiencia, siendo más notoria esta caída en el algoritmo 1. Sin embargo y al igual que ocurre con las medidas del speedup, se puede apreciar una ligera tendencia a la mejora de la eficiencia del algoritmo 1 al contrario que con el algoritmo 2 que continúa decayendo (**Figura 4.7**).

Por último, si bien con ambos algoritmos obtenemos valores muy similares, podemos concluir que el algoritmo 2 tiene un mejor rendimiento tanto en lo referente al tiempo de ejecución como de aprovechamiento de los recursos cuando se ejecuta con una cantidad de procesadores entre 2 y 16. Sin embargo, el algoritmo 1 parece tener una tendencia a mejorar su rendimiento pasados los 32 procesadores. Para comprobar esto, se debería realizar el mismo experimentos con mayor número de procesadores.

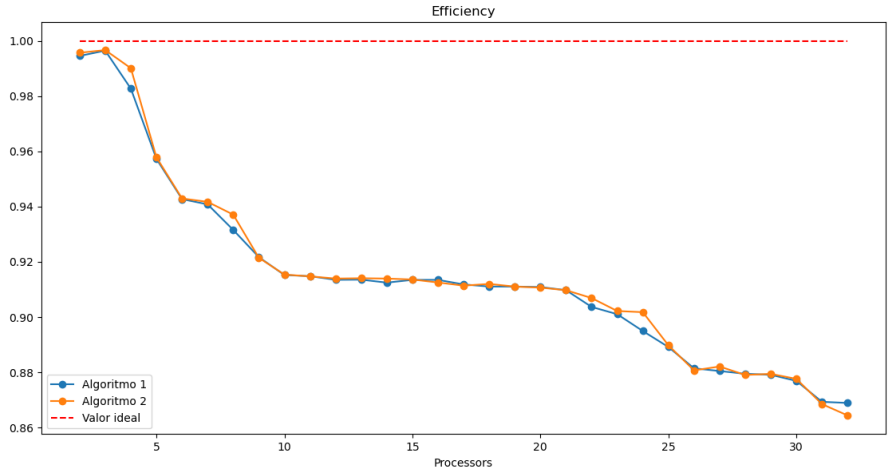


Figura 4.7: Comparación de la eficiencia de ambos algoritmos

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusión

Se hace evidente la importancia de la Computación de Alto Rendimiento cuando hablamos del análisis de grandes cantidades de datos o de la resolución de problemas muy complejos, que de otra manera no seríamos capaces de resolver. El correcto uso de esta herramienta es determinante para obtener los mejores resultados y es por ello por lo que el análisis del rendimiento se vuelve un paso imprescindible a la hora de trabajar en este sector.

Uno de los principales objetivos de este proyecto era cambiar el formato de salida de los datos del sistema CALL. Antes de realizar este cambio, los datos se imprimían directamente en un fichero XML, por lo que tras extraer los datos y antes de poder trabajar con ellos, se debía realizar un proceso de conversión de tipos, puesto que estos eran almacenados como cadenas de caracteres. Al hacer el cambio a un formato que nos permite almacenar los datos según la cualidad de estos, se pretende facilitar el trabajo futuro en la creación de scripts que permitan visualizar dichos datos.

Otro de los objetivos planteados fue la implementación de diferentes scripts donde se trabajara con los datos aportados por el sistema CALL, mostrando algunas de las métricas más utilizadas en el análisis del rendimiento. El propósito de este ejercicio era demostrar la importancia que tiene una herramienta de visualización a la hora de determinar el correcto funcionamiento de un algoritmo que es ejecutado en un computador de altas prestaciones. El correcto uso de esta herramienta nos permite vislumbrar más clara y rápidamente que está ocurriendo en nuestro sistema y cómo se está comportando el algoritmo.

5.2. Líneas futuras

Como línea futura se puede plantear la creación de una librería amplía para Python sobre visualización de métricas de rendimiento, que cuente con una gran variedad de formatos y que aporten mayor información al análisis posterior. Algunas de las métricas que se pueden incluir en esta librería a parte de las ya nombradas en este proyecto pueden ser el coste del trabajo realizado por todo el sistema para la ejecución de un algoritmo, la escalabilidad del mismo, el consumo de recursos (por ejemplo energético), etc.

Otra posible línea a seguir sería el desarrollo de una aplicación que permita introducir los datos obtenidos del rendimiento de un programa y que tenga predefinida los gráficos que se usan habitualmente en el análisis del rendimiento, obteniendo una rápida visualización del problema, sin necesidad de crear múltiples scripts para ello. Esta aplicación debe permitir además personalizar dicha salida, añadiendo otras medidas o métricas que el usuario quiera representar y que no estén presentes. El uso de JSON como interfaz para el intercambio de datos posibilita el desarrollo de estas aplicaciones en formato Web, por ejemplo.

Capítulo 6

Summary and Conclusions

The importance of High Performance Computing becomes evident when we talk about the analysis of large amounts of data or the resolution of very complex problems that we would not be able to solve otherwise. To solve these problems, so-called Supercomputers are used, which can have thousands of processors.

The use of these systems involves a high amount of energy and environmental cost, something to take into account when designing the algorithms that will be executed in them, since the longer they take, the more expense is generated. Performance analysis tools like CALL allow us to reduce these expenses, providing us with information regarding the execution of the code, and then optimize the process.

This project addresses the improvement of the CALL system data output and its visualization in the form of graphs that represent the metrics commonly used in performance analysis. To achieve this, it was necessary to modify the code that was previously responsible for generating the output file. Then, a series of experiments were carried out on the execution of two different algorithms and some graphs were made with data extracted during their execution.

Capítulo 7

Presupuesto

En este capítulo se pretende realizar un presupuesto sobre el trabajo realizado. Puesto que todas las herramientas utilizadas a lo largo de la elaboración de este proyecto son gratuitas, se hará el presupuesto en base al tiempo empleado en cada una de las tareas.

7.1. Presupuesto

Tareas	Horas	Presupuesto
Estudio de las herramientas utilizadas	50 horas	15€/hora
Cambio en el código	80 horas	15€/hora
Implementación de scripts de visualización.	48 horas	15€/hora
Estudio experimental: análisis de rendimiento.	50 horas	15€/hora
Documentación	60 horas	15€/hora
Total	288 horas	4320€

Tabla 7.1: Presupuesto del proyecto.

Para realizar este proyecto hicieron falta 268 horas con un precio final de 2630€.

Apéndice A

Fichero XML cpiprueba.dat

```
<cll_data
 CALL_VERSION="1.0"
 PROGRAM="cpiprueba.c"
 NODE_NAME="verode21"
 SYSNAME="Linux"
 RELEASE="5.4.0-0.bpo.2-amd64"
 VERSION="#1 SMP Debian 5.4.8-1~bpo10+1 (2020-01-07)"
 PARALLEL_ID="MPI" >

  <cll_experiment
   EXPERIMENT="ker"
   BEGIN_LINE=" 26"
   END_LINE=" 42"
   FORMULA="p 0 "
   INFORMULA="ker[0]"
   MAXTESTS="131072"
   DIMENSION="1"
   NUMIDENTS="0"
   IDENTS=""
   OBSERVABLES="PAPI_L2_TCM PAPI_REAL_USEC "
   COMPONENTS="1 "
   NUMTESTS="1 1 " >

    <machines>
      <node>verode21</node>
      <node>verode21</node>
    </machines>

    <headers>
      <h>CPU</h>
      <h>NCPUS</h>
      <h>PAPI_REAL_USEC  </h>
      <h>   PAPI_L2_TCM  </h>
    </headers>
```

```
35
36      <sample>
37        <row>
38          <c>0</c>
39          <c>2</c>
40          <c>         701805  </c>
41          <c>           1180  </c>
42        </row>
43        <row>
44          <c>1</c>
45          <c>2</c>
46          <c>         701353  </c>
47          <c>            612  </c>
48        </row>
49      </sample>
50
51    </cll_experiment>
52  </cll_data>
```

Apéndice B

Fichero cll_exp.h

```
/* cll_exp.h
 *
 * OUTPUT FORMAT: XML (cll_data.dtd)
 *
 */


#ifndef __CLL_EXPERIMENT_HH__
#define __CLL_EXPERIMENT_HH__

#ifdef __cplusplus
extern "C" {
#endif

#define cll_open_file() { \
  if (!cll_fdata) {  \
    if (!(cll_output_name)) { \
      cll_fdata = stdout; \
    } \
    else { \
      (cll_output_name) = cll_foutname((cll_output_name)); \
      fprintf(stderr,"FILE: %s\n",(cll_output_name)); \
      if (!(cll_fdata = fopen((cll_output_name) , "w"))) { \
        fprintf(stdout, "error: cannot open output file\n"); \
        exit(1); \
      } \
    } \
  } \
}

struct json_object *json;
struct json_object *cll_experiment;
struct json_object *headers;
struct json_object *sample;
```

```
35 struct json_object *samples;
36 #define start_json() {\
37   json = json_object_new_object(); \
38 }
39
40 #define cll_print_header(experiment,driver) { \
41   if(!cll_first_report) { \
42     struct utsname machine; \
43     uname(&machine); \
44     json_object *call_version = json_object_new_string(cll_version);\
45     json_object *program = json_object_new_string(cll_programname);\
46     json_object *nodename = json_object_new_string(machine.nodename);\
47     json_object *sysname = json_object_new_string(machine.sysname);\
48     json_object *release = json_object_new_string(machine.release);\
49     json_object *version = json_object_new_string(machine.version);\
50     json_object *parallel_id = json_object_new_string(driver);\
51 \
52     json_object_object_add(json, "CALL_VERSION", call_version);\
53     json_object_object_add(json, "PROGRAM", program);\
54     json_object_object_add(json, "NODE_NAME", nodename);\
55     json_object_object_add(json, "SYSNAME", sysname);\
56     json_object_object_add(json, "RELEASE", release);\
57     json_object_object_add(json, "VERSION", version);\
58     json_object_object_add(json, "PARALLEL_ID", parallel_id);\
59   \
60     cll_first_report++; \
61   } \
62 }
63
64 #define cll_print_exp_header(experiment)  { \
65   unsigned *tmp; \
66   char buffer [50];\
67   int i; \
68   cll_experiment = json_object_new_object();\
69   json_object *experiments = json_object_new_string( (experiment).name );\
70   \
71   json_object *begins = json_object_new_array();  \
72   for(tmp = (experiment).begin_lines; *tmp; tmp++){\
73     json_object *begin_line = json_object_new_int(*tmp);\
74     json_object_array_add(begins,begin_line);\
75   }\
76   \
77   json_object *ends = json_object_new_array();\
78   for(tmp = (experiment).end_lines; *tmp; tmp++){ \
79   \
80     json_object *end_line = json_object_new_int(*tmp);\
81     json_object_array_add(ends,end_line);\
```

```
82    }\
83    \
84    json_object *formula = json_object_new_string((experiment).formula);\
85    json_object *informula = json_object_new_string((experiment).informula);\
86    json_object *maxtest = json_object_new_int((experiment).maxtests);\
87    json_object *dimension = json_object_new_int((experiment).dimension);\
88    json_object *numidents = json_object_new_int((experiment).numidents);\
89  \
90    json_object *idents = json_object_new_array();\
91    for (i = (experiment).numidents -1; i>=0; i--) { \
92      json_object *ident = json_object_new_string((experiment).idents[i]);\
93      json_object_array_add(idents,ident);\
94    }\
95  \
96    json_object *observables = json_object_new_array();\
97    for (i = (experiment).numobservables - 1; i >= 0; i--) {\
98      json_object *observable = json_object_new_string((experiment).observables[i]);\
99  
100     json_object_array_add(observables,observable);\
101   }\
102 \
103   json_object *components = json_object_new_array(); \
104   for(i = 0; i < (experiment).dimension; i++){ \
105     json_object *component = json_object_new_string((experiment).components[i]);\
106     json_object_array_add(components,component);\
107   }\
108 \
109   json_object_object_add(cll_experiment, "EXPERIMENT", experiments);\
110   json_object_object_add(cll_experiment, "BEGIN_LINE", begins);\
111   json_object_object_add(cll_experiment, "END_LINE", ends);\
112   json_object_object_add(cll_experiment, "FORMULA", formula);\
113   json_object_object_add(cll_experiment, "INFORMULA", informula);\
114   json_object_object_add(cll_experiment, "MAXTESTS", maxtest);\
115   json_object_object_add(cll_experiment, "DIMENSION", dimension);\
116   json_object_object_add(cll_experiment, "NUMIDENTS", numidents);\
117   json_object_object_add(cll_experiment, "IDENTS", idents);\
118   json_object_object_add(cll_experiment, "OBSERVABLES", observables);\
119   json_object_object_add(cll_experiment, "COMPONENTS", components);\
120 \
121 }
122 
123 #define cll_print_NUMTESTS(cll_nt, len) { \
124   int cll_i;\
125   json_object *numtests = json_object_new_array();\
126   for(cll_i = 0; cll_i < (len); cll_i++){\
127     json_object *numtest = json_object_new_int( (cll_nt)[cll_i]);\
128     json_object_array_add(numtests,numtest);\
```

```
129  }\
130    json_object_object_add(cll_experiment, "NUMTESTS", numtests);\
131    \
132 }
133
134
135 #define cll_print_field_title(str) { \
136    json_object *string = json_object_new_string(str);\
137    json_object_array_add(headers,string);\
138 }
139
140
141 #define cll_print_field_value(str) { \
142    json_object *string = json_object_new_int(str);\
143    json_object_array_add(sample,string);\
144 }
145
146
147 #define cll_print_exp_fixed_fields(processor,ncpus) { \
148      json_object *pro = json_object_new_int(processor);\
149      json_object *ncpu = json_object_new_int(ncpus);\
150      json_object_array_add(sample,pro);\
151      json_object_array_add(sample,ncpu);\
152 }
153
154
155 #define cll_print_exp_fields_titles(experiment) { \
156    int cll_i;\
157    headers = json_object_new_array();\
158    json_object *cpu = json_object_new_string("CPU");\
159    json_object *ncpus = json_object_new_string("NCPUS");\
160    json_object_array_add(headers,cpu);\
161    json_object_array_add(headers,ncpus);\
162    \
163    for (cll_i = (experiment).numidents - 1; cll_i >= 0; cll_i--) { \
164    json_object *experiments = json_object_new_string( (experiment).idents[cll_i] );\
165      json_object_array_add(headers,experiments);\
166  } \
167    CLL_USERDEF_TITLES(); \
168    json_object_object_add(cll_experiment, "headers", headers);\
169    samples = json_object_new_array(); \
170 }
171
172
173 #define cll_print_experiment_samples(numidents,instances,numberoftests,processor,ncp
174    int cll_i, cll_k; \
175    for (cll_i = 0; cll_i < (numberoftests); cll_i++){ \
```

```
176     sample = json_object_new_array();\
177     cll_print_exp_fixed_fields(processor,ncpus);\
178     for(cll_k = 0; cll_k <numidents; cll_k++){\
179        json_object *instance = json_object_new_int(instances[cll_i].id[cll_k]);\
180        json_object_array_add(sample,instance);\
181     }\
182     CLL_USERDEF_VALUES(instances[cll_i]); \
183     json_object_array_add(samples,sample);\
184   } \
185 }
186
187
188 #define cll_print_end_of_experiment(experiment) { \
189   fpos_t cll_fdata_end; \
190   fgetpos(cll_fdata, &cll_fdata_end); \
191   fsetpos(cll_fdata, &cll_fdata_end); \
192 }
193
194 #ifdef __cplusplus
195 }
196 #endif
197
198
199 #endif /* __CLL_EXPERIMENT_HH__ */
```

Apéndice C

Fichero cll_MPI.h

```
1 #ifndef __CLL_MPI_H__
2 #define __CLL_MPI_H__ /* avoid multiple inclusion */
3
4
5 /* OBSOLETE: #define IS_MAIN_CPU !cll_myid() */
6 #define CLL_SYNC   MPI_Barrier(MPI_COMM_WORLD)
7 #define CLL_SYNC_(x) MPI_Barrier((x))
8
9 #include "mpi.h"
10 /* MPI interface for call */
11
12 #ifdef __cplusplus
13 extern "C" {
14 #endif
15
16 unsigned * cll_get_NUMTESTS(unsigned experiment_numtests, int numproc) {
17  unsigned * all_nt = (unsigned *)malloc(numproc*sizeof(unsigned));
18
19   /* Warning! the type is MPI_UNSIGNED since numtest has being declared as that */
20   MPI_Gather(&experiment_numtests,1,MPI_UNSIGNED,all_nt,1, \
21              MPI_UNSIGNED,0,MPI_COMM_WORLD);
22   return all_nt;
23 }
24
25 #define CLL_MPI_NODENAMES_SIZE 40
26
27 #define cll_print_mpi_nodes(myid,numproc){\
28   int cll_mpi_proc; \
29   struct utsname cll_mpi_machine;\
30   char cll_mpi_nodename[CLL_MPI_NODENAMES_SIZE];\
31 \
32   json_object *machines = json_object_new_array();\
33   if(!myid) { \
34     uname(&cll_mpi_machine); \
```

```
35      json_object *machine =
36      json_object_new_string(cll_mpi_machine.nodename);\
37
38      json_object_array_add(machines,machine);\
39      for(cll_mpi_proc=1; cll_mpi_proc<numproc; cll_mpi_proc++) {\
40        MPI_Status cll_status;\
41        MPI_Recv(cll_mpi_nodename, CLL_MPI_NODENAMES_SIZE*sizeof(char),
42                 MPI_BYTE, cll_mpi_proc, cll_mpi_proc,
43                 MPI_COMM_WORLD, &cll_status); \
44        MPI_Barrier(MPI_COMM_WORLD); /* barrier */ \
45
46        json_object *machine_2 =
47        json_object_new_string(cll_mpi_machine.nodename);\
48
49        json_object_array_add(machines,machine_2);\
50      }\
51      json_object_object_add(cll_experiment, "machines", machines);\
52    }\
53    else {\
54      uname(&cll_mpi_machine);\
55      \
56      sprintf(cll_mpi_nodename,"%s", cll_mpi_machine.nodename); \
57      for(cll_mpi_proc = 1; cll_mpi_proc < numproc; cll_mpi_proc++) {  \
58        /* send data from processor proc to 0*/ \
59        if (myid == cll_mpi_proc) \
60          MPI_Send(cll_mpi_nodename,
61                   CLL_MPI_NODENAMES_SIZE*sizeof(char),
62                   MPI_BYTE, 0, myid, MPI_COMM_WORLD); \
63
64        MPI_Barrier(MPI_COMM_WORLD); /* barrier */ \
65      } \
66    }\
67  }
68
69  #define end_of_json() {\
70    json_object_object_add(cll_experiment, "sample", samples);\
71    json_object_object_add(json, "cll_experiment", cll_experiment);\
72    fprintf(cll_fdata, json_object_to_json_string_ext(json,
73            JSON_C_TO_STRING_PRETTY));\
74  }
75
76
77  #define cll_report(experiment)                          \
78    {                                 \
79      int i;                                              \
80      int myid; \
81      unsigned * cll_numtests; \
```

```
82      int proc, numproc; \
83  \
84      /* fprintf(stderr,"cll_report MPI\n"); */ \
85  \
86      MPI_Comm_rank(MPI_COMM_WORLD,&myid); \
87      MPI_Comm_size(MPI_COMM_WORLD,&numproc); \
88      cll_numtests = cll_get_NUMTESTS((experiment).numtests, numproc);
    \
89  \
90      if (!myid) { \
91        cll_open_file(); \
92        start_json();\
93     cll_print_header((experiment),"MPI"); \
94     cll_print_exp_header((experiment)); \
95     cll_print_NUMTESTS(cll_numtests,numproc); \
96        cll_print_mpi_nodes(myid, numproc);\
97        cll_print_exp_fields_titles((experiment));\
98        cll_print_experiment_samples((experiment).numidents,
99                                     (experiment).instance,
100                                    (experiment).numtests,
101                                    myid, numproc);\
102
103       for(proc = 1; proc < numproc; proc++) {  \
104         MPI_Status cll_status; \
105         /* receive & print data from processor proc */ \
106         MPI_Recv((experiment).instance,
107                  cll_numtests[proc]*sizeof((experiment).instance[0]),
108                  MPI_BYTE, proc, proc,
109                  MPI_COMM_WORLD, &cll_status); \
110
111         cll_print_experiment_samples((experiment).numidents,
112         (experiment).instance,
113         cll_numtests[proc],
114         proc, numproc); \
115
116         MPI_Barrier(MPI_COMM_WORLD); /* barrier */ \
117       } \
118       end_of_json();\
119       cll_print_end_of_experiment(experiment);\
120     }\
121     else { \
122       cll_print_mpi_nodes(myid, numproc); \
123       for(proc = 1; proc < numproc; proc++) {  \
124         /* send data from processor proc to 0*/ \
125         if (myid == proc) \
126           MPI_Send((experiment).instance,
127                    (experiment).numtests*sizeof((experiment).instance[0]),
```

```
128                     MPI_BYTE, 0, proc, MPI_COMM_WORLD);\
129
130           MPI_Barrier(MPI_COMM_WORLD); /* barrier */ \
131         } \
132       }\
133 }
134
135
136 #ifdef __cplusplus
137 }
138 #endif
139
140
141 #endif /* __CLL_MPI_H__ */
```

Apéndice D

Fichero plot_1.py

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import json, os


path_to_json = 'data/other_data/'
json_files = [pos_json for pos_json in os.listdir(path_to_json)
if pos_json.endswith('40.json')]

sequential_sample = []
with open(os.path.join(path_to_json, 'sequential.json')) as sequential_file:
    json_data = json.load(sequential_file)
    sequential_sample.append(json_data['cll_experiment']['sample'])

sequential_nums = []

for p in sequential_sample:
    for j in p:
        sequential_nums.append(j[2])


sample = []
num_tests = []
for js in (json_files):
    with open(os.path.join(path_to_json, js)) as json_file:
        json_data = json.load(json_file)
        sample.append(json_data['cll_experiment']['sample'])
        num_tests.append(json_data['cll_experiment']['NUMTESTS'])


        number_cpus = []
    for p in sample:
        for i in p:
```

```
            number_cpus.append(i[1])


total_cpus = []
for i in range(32):
    total_cpus.append(i+1)


number_cpus = list(dict.fromkeys(number_cpus))

cpus = []
nums = []
tcms = []

for i in number_cpus:
    cpu = []
    num = []
    tcm = []

    for p in sample:
        for j in p:
            if j[1] == i:
                cpu.append(j[0])
                num.append(j[2])
                tcm.append(j[3])

    cpu = list(dict.fromkeys(cpu))
    cpus.append(cpu)
    nums.append(num)
    tcms.append(tcm)


sequential = np.mean(sequential_nums)


number_cpus.sort()

N = len(number_cpus)
parallel = []
for i in range (N):

    parallel.append(np.mean(nums[i]))


```

```
82 parallel.sort(reverse=True)
83
84
85 speedup = []
86 efficiency = []
87 for i in range(len(parallel)):
88     speedup.append(sequential / parallel[i])
89     efficiency.append(sequential / (number_cpus[i]*parallel[i]))
90
91
92
93 num_ef = []
94 for i in number_cpus:
95     num_ef.append(1)
96
97 plt.figure(1)
98 plt.plot(number_cpus,speedup, linestyle='-', marker='o')
99 plt.plot(number_cpus, number_cpus, "r--")
100
101
102 plt.title('Speedup')
103 plt.xlabel('Processors')
104
105 plt.figure(2)
106 plt.plot(number_cpus,efficiency, linestyle='-', marker='o')
107 plt.plot(number_cpus,num_ef, "r--")
108 plt.title('Efficiency')
109 plt.xlabel('Processors')
110
111
112 plt.show()
```

Apéndice E

Fichero boxplot_1.py

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import json, os

path_to_json = 'data/other_data/'
json_files = [pos_json for pos_json in os.listdir(path_to_json)
              if pos_json.endswith('40.json')]


sequential_sample = []
with open(os.path.join(path_to_json, 'sequential.json')) as sequential_file:
    json_data = json.load(sequential_file)
    sequential_sample.append(json_data['cll_experiment']['sample'])

sequential_nums = []

for p in sequential_sample:
    for j in p:
        sequential_nums.append(j[2])


sample = []
num_tests = []
for js in (json_files):
    with open(os.path.join(path_to_json, js)) as json_file:
        json_data = json.load(json_file)
        sample.append(json_data['cll_experiment']['sample'])
        num_tests.append(json_data['cll_experiment']['NUMTESTS'])


        number_cpus = []
    for p in sample:
        for i in p:
```

```
            number_cpus.append(i[1])


total_cpus = []
for i in range(32):
    total_cpus.append(i+1)


number_cpus = list(dict.fromkeys(number_cpus))

cpus = []
nums = []
tcms = []

for i in number_cpus:
    cpu = []
    num = []
    tcm = []

    for p in sample:
        for j in p:
            if j[1] == i:
                cpu.append(j[0])
                num.append(j[2])
                tcm.append(j[3])

    cpu = list(dict.fromkeys(cpu))
    cpus.append(cpu)
    nums.append(num)
    tcms.append(tcm)


sequential = np.mean(sequential_nums)


number_cpus.sort()

N = len(number_cpus)
parallel = []
for i in range (N):

    parallel.append(np.mean(nums[i]))



parallel.sort(reverse=True)

```

```
82
83 speedup = []
84
85 for i in range(len(parallel)):
86     speedup.append(sequential / parallel[i])
87
88
89
90 fig, ax = plt.subplots()
91 speedup_boxplot = []
92 for i in range (N):
93     for j in nums[i]:
94         dummy = []
95         dummy.append(sequential / j)
96
97     speedup_boxplot.append(dummy)
98
99 speedup_boxplot.sort()
100 ax.boxplot(speedup_boxplot, showfliers=True)
101 plt.plot(number_cpus, number_cpus, "r--")
102
103 plt.show()
```

Apéndice F

Fichero cpiprueba.c

```
1 #include "mpi.h"
2 #include <stdio.h>
3 #include <math.h>
4
5 // CALL DEFS
6 #pragma cll parallel MPI
7 #pragma cll uses PAPI
8
9 #define MAX_NUM_INTERVALS 10000000
10 double f( double );
11 double f( double a )
12 {
13     return (4.0 / (1.0 + a*a));
14 }
15
16 int main( int argc, char *argv[]) {
17     int n, myid, numprocs, i;
18     double PI25DT = 3.14159265358979323846264 3;
19     double mypi, pi, h, sum, x;
20
21     MPI_Init(&argc,&argv);
22     MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
23     MPI_Comm_rank(MPI_COMM_WORLD,&myid);
24
25 #pragma cll times 40
26 #pragma cll sync ker PAPI_REAL_USEC,PAPI_L2_TCM=ker[0]
27     n = 100000000;
28     h   = 1.0 / (double) n;
29     sum = 0.0;
30     for (i = myid + 1; i <= n; i += numprocs)
31     {
32         x = h * ((double)i - 0.5);
33         sum += f(x);
34     }
```

```
35      mypi = h * sum;
36
37      printf ("Working at process ID: %d\n", myid);
38
39      MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
40
41
42  #pragma cll end ker
43
44
45      if (myid == 0)
46      {
47          printf("pi is approximately %.16f, Error is %.16f\n",
48                 pi, fabs(pi - PI25DT));
49      }
50
51  #pragma cll end times
52  #pragma cll report ker
53
54      MPI_Finalize();
55
56      return 0;
57  }
```

Apéndice G

Fichero cpiprueba2.c

```
1 #include <mpi.h>
2 #include <stdio.h>
3 #include <math.h>
4
5 // CALL DEFS
6 #pragma cll parallel MPI
7 #pragma cll uses PAPI
8 #define MAX_NUM_INTERVALS 10000000
9
10 double fabs(double);
11
12 double f(double a) {
13     return (4.0 / (1.0 + a * a));
14     }
15
16 int main(int argc, char *argv[]) {
17     int  n, myid, numprocs, i;
18     double PI25DT = 3.14159265358979323846264 3;
19     double mypi, pi, h, sum, x;
20     int source, dest, tag = 100;
21     MPI_Status status;
22
23     MPI_Init(&argc,&argv);
24     MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
25     MPI_Comm_rank(MPI_COMM_WORLD, &myid);
26
27 #pragma cll times 40
28 #pragma cll sync ker PAPI_REAL_USEC,PAPI_L2_TCM=ker[0]
29     printf("\nProcess %d of %d", myid, numprocs);
30     n = 100000000;
31     h = 1.0 / (double) n;
32     sum = 0.0; // Suma de intervalos
33     for (i = myid + 1; i <= n; i += numprocs) {
34         x = h * ((double)i - 0.5);
```

```
35          sum += f(x);
36      }
37      mypi = h * sum;
38      printf("\nProcessor: %d, mypi: %.16f\n", myid, mypi);
39
40      if(myid == 0) {
41          pi = mypi;
42          for (i = 1; i < numprocs; i++) {
43              source = i;         // Recibe resultados del resto de procesos
44              MPI_Recv(&mypi,
45                       1,
46                       MPI_DOUBLE,
47                       source,
48                       tag,
49                       MPI_COMM_WORLD,
50                       &status);
51              pi += mypi;
52          }
53          printf("\npi es aproximadamente %.16f,
54                  el error cometido es %.16f", pi, fabs(pi - PI25DT));
55      } else {
56          dest = 0;
57          // Envía cálculo local al proceso 0
58          MPI_Send(&mypi, 1, MPI_DOUBLE, dest, tag, MPI_COMM_WORLD);
59      }
60  #pragma cll end ker
61  #pragma cll end times
62  #pragma cll report ker
63
64      MPI_Finalize();
65  }
```

Bibliografía

[1] Artículo sobre estadísticas y datos de facebook. https://kinsta.com/es/blog/estadisticas-facebook/.

[2] Demo boxplot matplotlib. https://matplotlib.org/3.3.1/gallery/pyplots/boxplot_demo_pyplot.html#sphx-glr-gallery-pyplots-boxplot-demo-pyplot-py.

[3] Documentation simple plot matplotlib. https://matplotlib.org/gallery/lines_bars_and_markers/simple_plot.html.

[4] Fórmula de leibniz para el cáculo de π. https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80.

[5] Herramienta para el análisis del rendimiento eztrace. http://eztrace.gforge.inria.fr/.

[6] Herramienta para el análisis del rendimiento paraver. https://www.bsc.es/discover-bsc/organisation/scientific-structure/performance-tools.

[7] Herramienta para el análisis del rendimiento tau. https://www.cs.uoregon.edu/research/tau/home.php.

[8] Herramienta para el análisis del rendimiento vampir 9.9. https://vampir.eu/.

[9] Herramienta qsub. http://gridscheduler.sourceforge.net/htmlman/htmlman1/qsub.html.

[10] Json-c github. https://github.com/json-c/json-c.

[11] Json org. https://www.json.org/json-es.html.

[12] Librería json para python. https://docs.python.org/3/library/json.html.

[13] Librería numpy para python. https://numpy.org/.

[14] Librería os para python. https://docs.python.org/3/library/os.html.

[15] Matplotlib. https://matplotlib.org/#.

[16] Mpi forum. https://www.mpi-forum.org/.

[17] Módulo pyplot de matplotlib para la elaboración de gráficas. https://matplotlib.org/api/pyplot_api.html.

[18] Python españa. https://es.python.org/.

[19] Superordenador facebook. https://www.top500.org/system/179068/.

[20] Francisco Almeida, Domingo Giménez, José Miguel Mantas, and Antonio M. Vidal. *Introducción a la programación paralela*. PARANINFO - Cengage Learning, 2008.

[21] Jesús Alberto Gonzalez Martinez, Marcela Printista, Germán Rodríguez Herrera, Casiano Rodríguez León, and Franciso Sande Gonzalez. *CALL User and Referene Guides*. 2003.

[22] Innovadores. La supercomputación española, al rescate contra el covid-19. https://innovadores.larazon.es/es/la-supercomputacion-espanola-al-rescate-contra-el-covid-19/, 2020.

[23] D. Terpstra, H. Jagode, H. You, and J Dongarra. *Collecting Performance Data with PAPI-C*. 2010.