Paralelización del cálculo del número π utilizando librerías de precisión múltiple en un clúster de computadores

José I. Aliaga[1] y Miguel Pardo[1]

Resumen—**La Computación de Altas Prestaciones (CAP), o *High Performance Computing* (HPC), es un campo de la Ingeniería Informática que tiene como principal objetivo extraer el mejor rendimiento de los recursos disponibles de un computador para la resolución de un problema informático. Alcanzar este objetivo requiere tener un conocimiento detallado de la arquitectura y del sistema operativo, así como de los algoritmos y los lenguajes de programación que permiten implementar códigos más eficientes.**

En este artículo se presenta la evaluación de dos librerías de precisión múltiple para CPU y cómo éstas se pueden utilizar en procesadores multinúcleo y en multicomputadores o *clusters* de procesadores multinúcleo para el cálculo de la constante numérica π.

Palabras clave—**Computación de Altas Prestaciones; Programación Paralela con OpenMP y MPI; Librerías de precisión múltiple: GMP y MPFR; Número π y Algoritmos Espiga.**

I. Introducción

El número π es una de las constantes numéricas irracionales más estudiadas por la comunidad científica. Este valor es principalmente conocido en geometría por ser la relación entre la longitud y el diámetro de una circunferencia, pero también aparece en otros campos como la estadística y el álgebra. Sus primeras apariciones datan alrededor del 1800 a.C y a lo largo de la historia han surgido nuevas fórmulas, series y desarrollos para el cálculo del número π, que han permitido ampliar el conocimiento sobre esta constante trascendental. En los últimos años, gracias a la computación y a las mejoras en la arquitectura de los procesadores, se ha conseguido calcular y obtener una cantidad ingente de decimales de dicha constante y, hoy día, sigue siendo objeto de estudio e investigación por parte de la comunidad científica [1].

A. Concurrencia y Paralelismo

La Programación Concurrente y Paralela ha adquirido mucha relevancia en los últimos años debido a la aparición y éxito de las arquitecturas *multicore*. En la actualidad es ampliamente utilizada en grandes sistemas, en aplicaciones que requieren rapidez de ejecución, en sistemas de tiempo real y en computación gráfica entre otros.

Es habitual confundir los conceptos de concurrencia y paralelismo, pero es muy importante entender las diferencias existentes entre ambos términos. La concurrencia surge ligada a los sistemas operativos para facilitar el solapamiento de las operaciones e implica la división de las tareas para poder intercalar su ejecución. De esta forma, se permite que varios procesos compartan una única CPU. Por su parte, el paralelismo implica simultaneidad, es decir, dos procesos –o tareas– son ejecutados al mismo tiempo empleando diferentes unidades de procesamiento. Esta técnica permite acelerar la velocidad de cálculo de un proceso si se fragmenta en tareas y se reparte de forma adecuada entre las distintas unidades de procesamiento. Esta diferencia aparece representada de forma visual en la Figura 1.

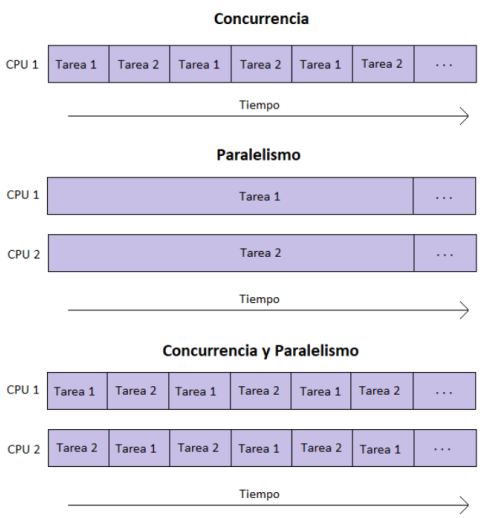

Fig. 1: Concurrencia y Paralelismo.

Las arquitecturas paralelas que se van a utilizar en este artículo son las que Michael J. Flynn clasificó como MIMD (*Multiple Instruction, Multiple Data*) que incluye tanto a los sistemas multinúcleo como a los multiprocesadores. Esta clasificación, también conocida como taxonomía de Flynn, se fundamenta en el número de instrucciones y datos que se ejecutan/tratan en paralelo. La Tabla I muestra los cuatro tipos de arquitecturas que define Flynn y un ejemplo para cada una [2].

En las arquitecturas MIMD, el modelo de programación más común es el denominado SPMD (*Single Program, Multiple Data*), en el que un mismo programa es ejecutado en todos los procesos, por lo que se deben incluir condiciones que permitan que cada proceso actúe de modo diferente. Este paradigma es el utilizado en este artículo para la paralelización del cálculo del número π en las arquitecturas paralelas basadas en MIMD.

[1]Dpto. de Ingeniería y Ciencia de los Computadores, Universitat Jaume I, e-mails: aliaga@uji.es, mipardo@uji.es

Tabla I: Categorización de Flynn.

Nombre	Ejemplos
SISD (Single Instruction, Single Data)	Monoprocesador convencional.
SIMD (Single Instruction, Multiple Data)	Procesadores vectoriales o superescalares.
MISD (Multiple Instruction, Single Data)	No existen arquitecturas de este tipo.
MIMD (Multiple Instruction, Multiple Data)	Sistemas multinúcleo y multiprocesadores.

Las arquitecturas MIMD se pueden subdividir en dos grandes grupos: las arquitecturas multiprocesador y las arquitecturas multicomputador. En las primeras, todos los procesadores están en un único sistema y acceden al mismo espacio de memoria a través de diferentes mecanismos, por lo que también se dice que están fuertemente acopladas; un ejemplo serían los procesadores multinúcleo, que se integran en un único chip y comparten el espacio de memoria. Por su parte, en las segundas cada procesador con su memoria aparece en un único sistema y todos ellos se comunican a través de una red de interconexión, razón por la que también se les denomina sistemas débilmente acoplados. En la actualidad, los grandes sistemas de cómputo integran ambos grupos en un clúster de sistemas multiprocesador.

B. Estructura del artículo

Tras la presente introducción, en la siguiente sección se introducen las librerías de precisión múltiple propuestas para el cálculo de π, y también se presentan los tres algoritmos *Spigot* utilizados para la resolución de la constante. Seguidamente, en la tercera sección se describe el *hardware*, la metodología y las métricas utilizadas junto con los resultados de cada implementación. Finalmente, en la cuarta y última sección se presentan las conclusiones y una perspectiva global de los resultados del estudio.

II. Antecedentes

El alto coste computacional que requiere el cálculo de los decimales del número π, convierte este problema en una aplicación didáctica y muy interesante en el campo de la Computación de Altas Prestaciones. Realizar este cálculo de modo eficiente requiere utilizar un lenguaje de programación que permita tener un control directo de la ejecución de los programas, tanto en lo que se refiere a velocidad de ejecución como en el uso de memoria. Es por ello que, se ha elegido el lenguaje de programación C para implementar los algoritmos desarrollados en este artículo.

A. Librerías de precisión múltiple

Como se mencionaba en secciones previas, este artículo tiene la intención de comparar diferentes librerías de precisión múltiple con distintos algoritmos para el cálculo de π. En este trabajo las librerías utilizadas han sido GMP y MPFR.

GMP (*GNU Multiple Precision Arithmetic Library*) es una librería de cálculo aritmético que proporciona funciones muy optimizadas para operar con números enteros, racionales y en coma flotante empleando cualquier precisión. Este valor de precisión está únicamente limitado por los recursos de memoria disponibles del sistema en el que se ejecutan los programas [3]. Como se puede deducir, los tipos de datos utilizados para este artículo son en coma flotante. En GMP, el principal objetivo de sus estructuras es realizar los cálculos de la forma más eficiente posible, realizando el redondeo de las operaciones truncando hacia el cero. En lo que se refiere a su estructura interna, esta se compone de cuatro campos:

- *_mp_size* almacena el número de *limbs* utilizados, donde un *limb* es la parte de un número de precisión múltiple que cabe en una palabra (normalmente 32 o 64 bits). En GMP, si se desea representar un número negativo, el valor almacenado en este campo será negativo.
- *_mp_prec* representa la precisión de la mantisa, indicando el número de *limbs*.
- *_mp_d* es el puntero al vector de *limbs* almacenados en «*little endian*» (los primeros elementos del vector contienen los *limbs* menos significativos)
- *_mp_exp* indica la posición del punto decimal en el vector *_mp_d*.

Pro su parte, MPFR (*Multiple Precision Floating-Point Reliable Library*) es una librería de C basada en GMP, de libre distribución que proporciona redondeo exacto para todas las operaciones que implementa [4]. A diferencia de la anterior, esta librería soporta diferentes tipos de redondeo como los que se presentan a continuación [5].

- *MPFR_RNDN*: redondeo hacia el más próximo y, en caso de que el número se encuentre en el centro de dos números representables consecutivos, se redondea al que tenga un significante par. Esta ha sido la utilizada para todas las implementaciones con MPFR.
- *MPFR_RNDD*: redondeo hacia el infinito negativo.
- *MPFR_RNDU*: redondeo hacia el infinito positivo.
- *MPFR_RNDZ*: redondeo hacia el cero.
- *MPFR_RNDA*: redondeo opuesto al anterior.

En lo que respecta a la estructura interna de MPFR, esta se compone de:

- *_mpfr_prec* es el campo utilizado para almacenar la precisión de la variable.
- *_mpfr_sign* representa el signo de la variable.
- *_mpfr_exp* almacena el exponente.
- *_mpfr_d* es el puntero al vector de *limbs*.

B. Paralelismo en Arquitecturas MIMD

Dado que se va a trabajar con diferentes arquitecturas MIMD, se presentan las modelos de programación más comunes para estas arquitecturas.

OpenMP (*Open Multi-Processing*) es una API para la programación paralela disponible para los lenguajes de programación C, C++ y Fortran que proporciona una interfaz sencilla para el desarrollo de aplicaciones paralelas sobre multiprocesadores [6].

Por otro lado, MPI son las iniciales de *Message Passing Interface*, y es una interfaz para la programación de paso de mensajes sobre multicomputadores, y está disponible para los mismos lenguajes que OpenMP. La primera versión fue publicada en 1994 con el objetivo de proporcionar una librería para la comunicación entre procesos estándar y portable para cualquier computador [7]. El modelo de programación teórico en el que se basa MPI es MIMD, pero por simplicidad, de la misma forma que con OpenMP, se utiliza el modelo SPMD. De esta forma, todos los procesos ejecutan un único programa empleando directivas condicionales para cada proceso o hebra.

C. Algoritmos Spigot

Existen múltiples algoritmos que proporcionan una aproximación del número π con mayor o menor precisión. La más conocida viene dada por su relación con la longitud de la circunferencia, pero gracias al interés que despierta este campo y a los estudios realizados, se han desarrollado muchos otros métodos para calcular esta constante numérica. Todos los que se presentan en este trabajo se clasifican como algoritmos *Spigot* –o algoritmos Espiga– ya que, hasta ahora, son los métodos conocidos que permiten alcanzar un mayor número de decimales. Los algoritmos *Spigot* son aquellos que generan dígitos periódicamente y de izquierda a derecha, de forma que los dígitos calculados no son reutilizados para la obtención de otros nuevos [8]. Este tipo de algoritmos son habitualmente utilizados para la computación de números trascendentales como π o e.

En este artículo se propone para la resolución de π tres diferentes algoritmos *Spigot*: Bailey-Borwein-Plouffe, Bellard y Chudnovsky.

La serie de Bailey-Borwein-Plouffe fue desarrollada en 1995 y presentada dos años más tarde en la revista *Mathematics of Computation*. Sus autores, Simon Plouffe, David Bailey y Peter Borwein, demuestran que el algoritmo permite calcular el dígito d del número π sin necesidad de calcular los decimales previos con un coste de tiempo lineal [9]. El desarrollo de esta serie supuso un gran impulso para la búsqueda de nuevos algoritmos Spigot más eficientes en el cálculo de números trascendentales. Su expresión es la siguiente:

$$\pi = \sum_{n=0}^{\infty} \frac{1}{16^n}\left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6}\right) \quad (1)$$

Por su parte, el 20 de enero de 1997, Fabrice Bellard publicó una fórmula que aseguraba ser la versión mejorada de la serie (1). Según su autor, esta serie es un 43 % más rápida en el cálculo del enésimo decimal del número π que la serie de Bailey, Borwein y Plouffe [10].

$$64\pi = \sum_{n=0}^{\infty} \frac{-1^n}{1024^n}\left(-\frac{32}{4n+1} - \frac{1}{4n+3} + \frac{256}{10n+1} - \frac{64}{10n+3} - \frac{4}{10n+5} - \frac{4}{10n+7} + \frac{1}{10n+9}\right) \quad (2)$$

Finalmente, el tercer algoritmo que se implementa en este artículo es el desarrollado por los hermanos Chudnovsky. Se trata de una fórmula empleada en múltiples ocasiones para batir el récord de máximos decimales calculados del número π, tanto en ordenadores personales como en supercomputadores. Fabrice Bellard, el autor de la serie (2), consiguió batir el récord calculando 2700 billones de decimales de la constante trascendental con esta expresión [11]:

$$\frac{426880\sqrt{10005}}{\pi} = \sum_{n=0}^{\infty} \frac{(6n)!(545140134n+13591409)}{(n!)^3(3n)!(-640320)^{3n}} \quad (3)$$

III. Resultados experimentales

Antes de presentar los resultados obtenidos es importante describir el hardware en el que los códigos han sido ejecutados y cuáles han sido las métricas utilizadas para comparar y evaluar el rendimiento de los distintos algoritmos y librerías.

A. Descripción del Hardware

Los resultados se han obtenido en un servidor de cálculo facilitado por el Departamento de Ingeniería y Ciencia de los Computadores (DICC) de la Universitat Jaume I. El servidor de cálculo está compuesto por 4 nodos con 16 núcleos AMD EPYC 7351P que trabajan a una frecuencia de 1,2 GHz, y con una memoria RAM de 16 GB. Los nodos se conectan a través de una red Ethernet 1G con una latencia de 3,3 microsegundos y un ancho de banda de 117 MB/s.

Por lo que respecta al software, se ha utilizado la versión 4.4.7 del compilador GCC y la 4.0.2 de OpenMPI. De las librerías numéricas, se ha utilizado la versión 6.2.1 de GMP y la 4.1.0 de MPFR.

B. Metodología

Para poder realizar una comparación «justa» de los diferentes algoritmos, se ha optado por fijar el número de dígitos del número π a calcular en cada prueba. Dado que cada método obtiene los decimales exactos de un modo diferente, ha sido necesario estimar con antelación una expresión que caracterice cada algoritmo, tal y como sigue:

- *Algoritmo de Bailey-Borwein-Plouffe.* Requiere un total de 84 iteraciones para obtener 100 decimales, por lo que se obtiene $1,19$ decimales por iteración. Así pues, la expresión asociada es la siguiente:

$$iteraciones_{BBP} = precisión \cdot 0,84$$

- *Algoritmo de Bellard.* En este caso se obtienen 3 decimales en cada iteración, por lo que la expresión sería:

$$iteraciones_{Bell} = \lfloor \frac{precisión}{3} \rfloor$$

- *Algoritmo de Chudnovsky.* Esta serie permite obtener hasta 14 decimales por iteración, aunque la expresión es un tanto más compleja:

$$iteraciones_{Chud} = \lfloor \frac{precisión + 13}{14} \rfloor$$

Para hallar estos valores ha sido necesario evaluar cada uno de los métodos con diferentes precisiones de forma previa a la obtención de resultados, pero este estudio no forma parte del análisis de costes de cada algoritmo, sino que se considera una etapa previa. Una vez obtenidas las expresiones para cada algoritmo, resulta sencillo fijar una precisión y ejecutar el número de iteraciones necesario para obtener los dígitos deseados.

Finalmente, cabe destacar que los resultados mostrados a continuación son el valor medio de cinco ejecuciones, pero considerando únicamente la fase de cálculo. Es decir, la etapa de validación al finalizar el cálculo no se ha considerado en los tiempos de ejecución ya que es equivalente en todos los casos.

C. Métricas y evaluación del rendimiento

Para la evaluación del rendimiento de los algoritmos, se han empleado un conjunto de métricas que permiten comparar las diferentes implementaciones que se presentan en el artículo. Estas métricas son el tiempo de ejecución (TE), el incremento de velocidad (Sp) y la eficiencia (Ep).

El tiempo de ejecución es la métrica principal y la base para la obtención de otros índices en la evaluación y comparación de los algoritmos. Este parámetro puede ser suficiente para reflejar la mejora de prestaciones de los algoritmos paralelos frente a los secuenciales y para comparar los resultados entre distintos algoritmos secuenciales, pero puede resultar escaso para analizar y valorar la calidad de los algoritmos paralelos.

Por otra parte, el incremento de velocidad, o *speedup*, es una métrica ampliamente utilizada para reflejar la escalabilidad de los algoritmos paralelos. Se dice que un algoritmo es escalable cuando aprovecha todos los recursos del sistema de forma eficiente. El incremento de velocidad se calcula como el cociente entre el tiempo de ejecución del mejor algoritmo secuencial y el tiempo de ejecución de la implementación paralela.

$$Sp = \frac{Tiempo\ del\ mejor\ algoritmo\ secuencial}{Tiempo\ paralelo}$$

Cuanto más próximo sea el valor del incremento de velocidad al número de procesos o hebras que se emplean para el cálculo en paralelo, mayor es la ganancia y el aprovechamiento de los recursos.

La última métrica que se considera en este artículo es la eficiencia, una métrica que pretende expresar el concepto de ganancia respecto al número de unidades de procesamiento que se emplean. Se obtiene como el cociente entre el incremento de velocidad y el número de procesos o hebras empleadas en el cálculo.

$$Ep = \frac{Sp(p)}{p}$$

El resultado de la eficiencia será igual a uno cuando el algoritmo en desarrollo sea completamente paralelizable, es decir que la carga de trabajo se haya repartido perfectamente entre las distintas unidades de procesamiento. Si bien este valor es difícilmente alcanzable, es importante destacar que valores próximos a uno indican una buena escalabilidad del algoritmo paralelo.

D. Paralelización de Bailey-Borwein-Plouffe

El primer algoritmo que se presenta es el de *Bailey-Borwein-Plouffe* (BBP en adelante) debido a su mayor simplicidad. La resolución de la serie consiste en resolver el sumatorio mediante un bucle, acumulando los resultados obtenidos de cada iteración en una variable proporcionando directamente el valor de π.

Para optimizar la resolución de la serie, es necesario, en primer lugar, resolver la potencia de forma recursiva. Es decir, como el algoritmo requiere calcular la potencia 16^n en cada iteración, si se almacena el resultado de la iteración anterior sólo será necesario multiplicarlo este valor por 16 para obtener el resultado, siendo mucho más eficiente que resolver la operación de forma «tradicional». La expresión recursiva para la resolución de la potencia se muestra a continuación:

$$a^b = a^{b-1} \cdot a$$

Pero la expresión anterior presenta un problema, y es que paralelizar la operación genera una dependencia de datos. Por ejemplo, en caso que se resuelva el problema utilizando dos hebras, la segunda tendrá que esperar a la primera para obtener el valor de la potencia de la iteración previa y viceversa. Por tanto, para permitir que las hebras trabajen de forma independiente sin necesidad de esperar y sincronizar sus cálculos se propone una pequeña modificación de la expresión recursiva:

$$a^b = a^{b-t} \cdot a^t,$$

donde t es el número de hebras que se emplean en el cálculo. De esta forma, las hebras ya pueden resolver una iteración de la serie realizando un reparto cíclico de las iteraciones de forma independiente.

En segundo lugar, para agilizar el cálculo también se utilizan operaciones de bits ya que generalmente son más eficientes. En cuatro de las cinco fracciones que aparecen en la serie se requiere calcular $8n$, que se puede expresar como una potencia de dos: $2^3 n$. Por tanto, la operación de multiplicación se puede realizar como un desplazamiento de tres bits hacia la izquierda. Además, aplicar el desplazamiento de bits permite emplear otro operador binario para resolver las expresiones $8n+1$, $8n+4$, $8n+5$ y $8n+6$ porque los tres bits de menor peso resultantes se quedan a cero. Esto permite «rellenar» dichos bits simulando una operación de suma con el operador binario OR.

Las Tablas IIy III muestran los tiempos de ejecución con OpenMP para el cálculo de diferentes precisiones de π con la librería GMP y MPFR, respectivamente. En este caso, es la segunda librería quien realiza los cálculos de forma mucho más rápida. La

Tabla II: Tiempos de ejecución (en segundos) del algoritmo BBP con GMP y OpenMP.

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	2,290	1,091	0,549	0,285	0,189	0,149
20.000	11,050	5,662	2,834	1,431	0,954	0,747
30.000	27,638	14,115	7,067	3,546	2,683	2,323
40.000	59,675	31,035	15,742	8,094	5,577	4,402
50.000	91,129	46,959	23,813	12,214	8,447	6,632

Tabla III: Tiempos de ejecución (en segundos) del algoritmo BBP con MPFR y OpenMP

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	0,350	0,179	0,092	0,049	0,036	0,032
20.000	1,368	0,699	0,351	0,183	0,137	0,128
30.000	3,051	1,546	0,778	0,397	0,283	0,219
40.000	5,394	2,724	1,367	0,690	0,481	0,388
50.000	8,430	4,247	2,135	1,075	0,757	0,673
100.000	33,452	17,194	8,815	4,567	3,133	2,541

Tabla IV: Tiempos de ejecución (en segundos) del algoritmo BBP con GMP y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	2,123	0,283	0,151	0,082	0,063	0,058
20.000	11,052	1,426	0,737	0,371	0,258	0,272
30.000	27,663	3,544	1,781	0,906	0,612	0,642
40.000	59,713	7,688	3,862	1,946	1,349	1,364
50.000	91,281	11,691	5,869	2,984	1,979	2,082
100.000	400,95	51,537	26,041	13,084	9,085	8,353

Tabla V: Tiempos de ejecución (en segundos) del algoritmo BBP con MPFR y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	0,346	0,056	0,032	0,023	0,020	0,015
20.000	1,368	0,184	0,099	0,056	0,047	0,043
30.000	3,045	0,395	0,205	0,129	0,082	0,086
40.000	5,392	0,689	0,380	0,186	0,133	0,331
50.000	8,400	1,070	0,548	0,292	0,198	0,238
100.000	34,065	4,309	2,168	1,102	0,750	0,814

escalabilidad de ambas librerías empleando múltiples hebras se representa en las figuras 2 y 3. Como se puede apreciar, tanto la implementación con la librería GMP como MPFR escalan de forma relativamente óptima, pero es GMP quien muestra ligeramente una pendiente más positiva.

Por otra parte, de la misma forma que para OpenMP, las Tablas IV y V muestran los tiempos de ejecución obtenidos utilizando procesos en vez de hebras. Cabe destacar que son los cuatro nodos los que intervienen en el cálculo donde cada uno genera dieciséis procesos. Los resultados de escalabilidad para GMP y MPRF se representan en las Figuras 4 y5, respectivamente. Al igual que antes, MPFR presenta mejores tiempos de ejecución, pero en lo que respecta a la escalabilidad, MPFR no alcanza los valores obtenidos con GMP.

E. Bellard

La serie de Bellard comparte muchas similitudes con la serie de Bailey-Borwein-Plouffe. De hecho, si se comparan las dos fórmulas, se aprecia que ambas constan de un sumatorio del producto de dos cocientes, en el que uno de ellos se obtiene a partir

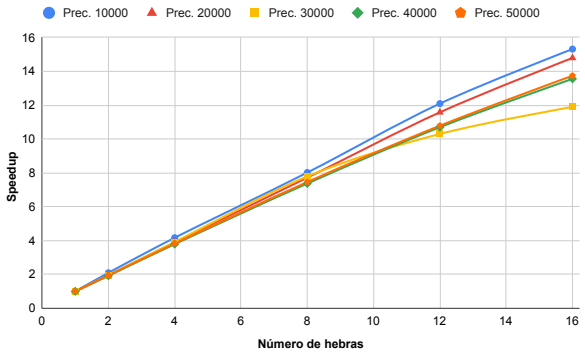

Fig. 2: Escalabilidad del algoritmo BBP con la librería GMP y OpenMP.

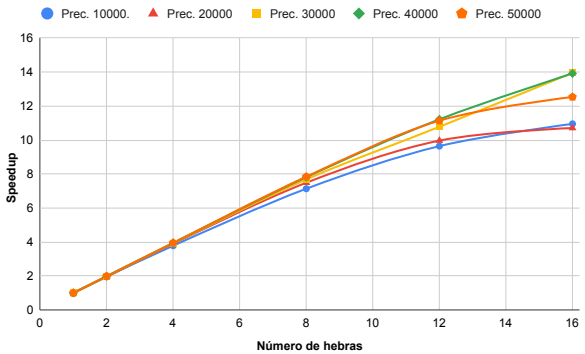

Fig. 3: Escalabilidad del algoritmo BBP con la librería MPFR y OpenMP.

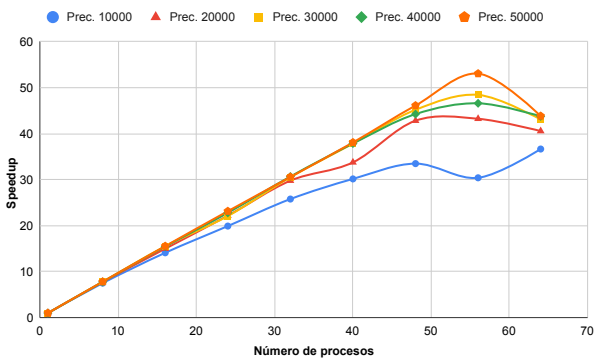

Fig. 4: Escalabilidad del algoritmo BBP con la librería GMP y MPI.

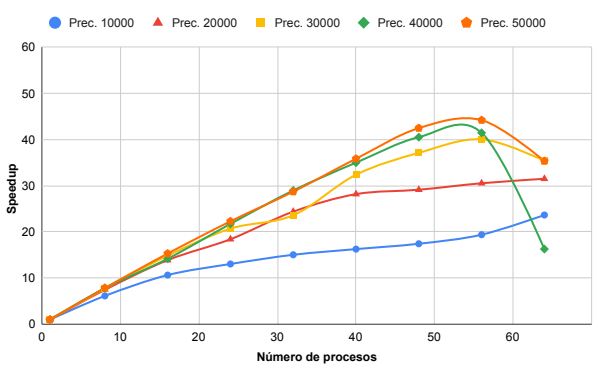

Fig. 5: Escalabilidad del algoritmo BBP con la librería MPFR y MPI.

de sumar y restar un conjunto de cocientes. Además, también se aprecia una operación matemática similar

Tabla VI: Tiempos de ejecución (en segundos) del algoritmo Bellard con GMP y OpenMP.

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	0,936	0,475	0,238	0,121	0,086	0,067
20.000	4,775	2,399	1,201	0,609	0,417	0,356
30.000	11,846	5,938	2,971	1,493	1,003	0,868
40.000	25,475	12,896	6,539	3,343	2,307	2,012
50.000	38,837	19,648	9,939	5,103	3,470	3,016
100.000	168,198	85,185	43,066	22,524	15,201	11,721

Tabla VII: Tiempos de ejecución (en segundos) del algoritmo Bellard con MPFR y OpenMP

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	0,239	0,124	0,063	0,036	0,026	0,023
20.000	0,925	0,473	0,237	0,126	0,100	0,091
30.000	2,061	1,044	0,524	0,271	0,191	0,147
40.000	3,652	1,843	0,924	0,470	0,408	0,301
50.000	5,673	2,858	1,432	0,726	0,508	0,475
100.000	22,556	11,488	5,831	2,991	2,058	1,713

Tabla VIII: Tiempos de ejecución (en segundos) del algoritmo Bellard con GMP y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	0,937	0,128	0,076	0,042	0,035	0,031
20.000	4,786	0,628	0,315	0,166	0,116	0,123
30.000	11,857	1,497	0,772	0,440	0,306	0,294
40.000	25,532	3,225	1,616	0,835	0,564	0,585
50.000	38,825	4,998	2,455	1,241	0,878	0,914
100.000	170,02	21,338	10,849	5,442	3,709	3,838

Tabla IX: Tiempos de ejecución (en segundos) del algoritmo Bellard con MPFR y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	0,236	0,040	0,026	0,021	0,017	0,016
20.000	0,942	0,131	0,075	0,043	0,036	0,034
30.000	2,059	0,271	0,145	0,079	0,062	0,059
40.000	3,650	0,471	0,246	0,145	0,101	0,095
50.000	5,669	0,731	0,376	0,197	0,146	0,182
100.000	22,832	2,906	1,468	0,747	0,512	0,730

a la abordada en la serie anterior: 1024^n. En cambio, esta vez se ha optado por resolver las expresiones del tipo $a \cdot n + b$ de forma recursiva como sigue:

$$R(n) = a \cdot n + b = R(n-1) + a,$$

donde $R(0) = b$. De nuevo, se ha ajustado la expresión para que las hebras puedan trabajar de forma independiente mediante un reparto cíclico de las iteraciones:

$$R(n) = a \cdot n + b = R(n-t) + a \cdot t,$$

donde $R(0) = b$ y t equivale al número de hebras que se emplean en el cálculo.

Por último, sabiendo que la potencia que aparece en la serie (1024^n) es equivalente a 2^{10n}, se puede resolver esta operación mediante un desplazamiento de bits. En este caso, el resultado de la potencia se obtiene desplazando $10n$ veces el número 1, tal y como se expresa a continuación:

$$1024^n = 2^{10n} = 1 \ll (10n)$$

La Tabla VI muestra los tiempos de ejecución implementando la serie de Bellard con la librería GMP, mientras que los resultados multihebra para la librería MPFR se presentan en la Tabla VII. A partir de ambas tablas se pueden obtener las gráficas de escalabilidad, tal y como se presentan en la Figura 6 para GMP y en la Figura 7 para MPFR. Los resultados de los algoritmos multiproceso se muestran en las Tablas VIII y IX. Seguidamente, los resultados de escalabilidad se presentan en las Figuras 8 y 9 para la librería GMP y MPFR, respectivamente. Con este segundo algoritmo se repite el mismo escenario detectado, donde MPFR presenta unos tiempos de ejecución notablemente más rápidos pero GMP consigue aprovechar ligeramente mejor los recursos del sistema.

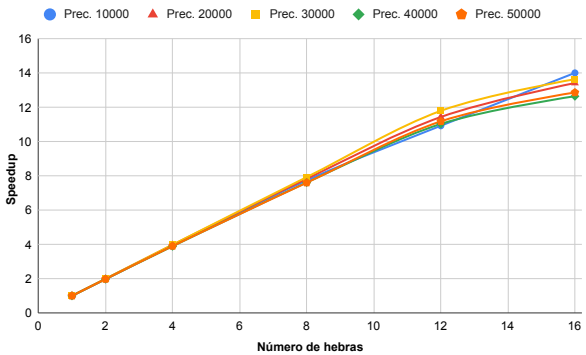

Fig. 6: Escalabilidad del algoritmo Bellard con la librería GMP y OpenMP.

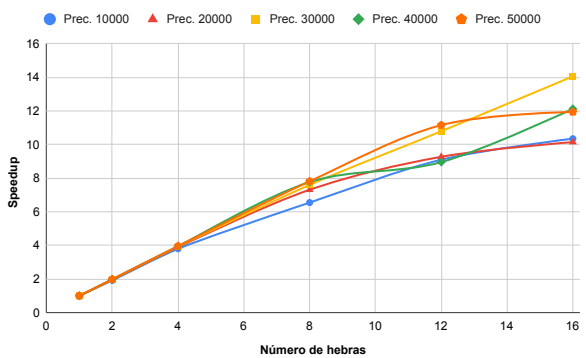

Fig. 7: Escalabilidad del algoritmo Bellard con la librería MPFR y OpenMP.

F. *Chudnovsky*

Este último algoritmo es conocido por haber sido utilizado en múltiples ocasiones para batir el récord mundial de máximos decimales calculados de la constante π. Su potencia radica en que el cálculo de una iteración obtiene un número de decimales considerablemente mayor que BBP y Bellard. Tal y como se puede deducir de la relación obtenida entre iteraciones y precisión deseada, Chudnovsky obtiene de media casi 14 decimales en cada iteración, mientras que BBP obtiene 1,19 decimales y Bellard 3.

La serie 3 presenta ciertas operaciones similares que ya han sido resueltas y abordadas en BBP y Bellard, pero además se plantea un nuevo reto: el cálculo de factoriales. En una primera implementación se calculaban y almacenaban todos los factoria-

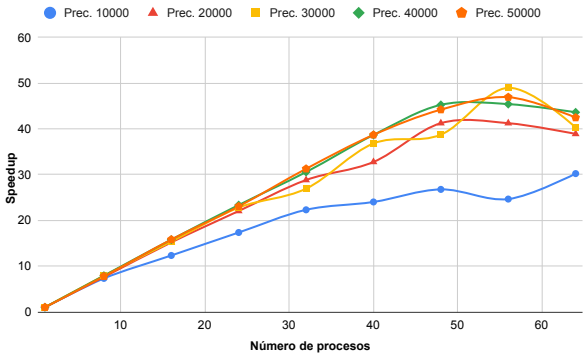

Fig. 8: Escalabilidad del algoritmo Bellard con la librería GMP y MPI.

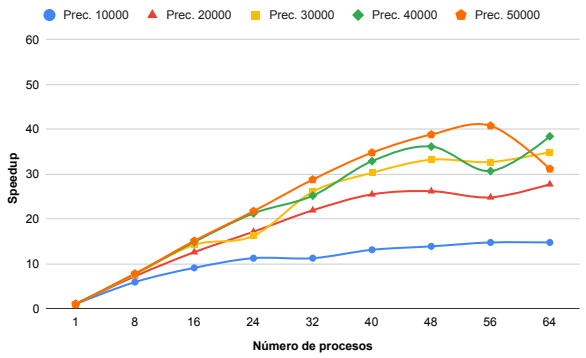

Fig. 9: Escalabilidad del algoritmo Bellard con la librería MPFR y MPI.

les de forma previa al cómputo de las iteraciones. De esta forma, en cada iteración, las hebras o procesos podían acceder al factorial que necesitasen sin depender de otras hebras o procesos, pero simplificando la expresión matemática que agrupa el cálculo de los factoriales se obtienen mejores resultados. La expresión que agrupa el cálculo de los factoriales se puede simplificar como sigue:

$$A(n) = \frac{(6n)!}{(n!)^3 \cdot (3n)!} = \frac{(12n+2)\cdot(12n+6)\cdot(12n+10)}{(n+1)^3} \cdot A(n-1)$$

Las otras dos dependencias que se pueden encontrar en la serie se abordan de la siguiente manera:

$$\begin{aligned} B(n) &= (-640320)^{3n} = (-640320)^3 \cdot B(n-1) \\ C(n) &= 545140134n + 13591409 = 545140134 + C(n-1) \end{aligned}$$

A diferencia de los dos anteriores, este algoritmo realiza un reparto por bloques de las iteraciones, y por ello, cada hebra o proceso deberá calcular de forma directa los primeros valores de las tres dependencias para poder trabajar de forma independiente.

Finalmente, tras realizar un análisis del coste de las iteraciones de cada algoritmo, se observa que, a diferencia de la serie Bailey-Borwein-Plouffe o de Bellard, el tiempo que requiere realizar el cálculo de una iteración no es constante. Como se puede apreciar en la Figura 10, el coste aumenta en cada iteración. La razón más evidente es que la variable que almacena la dependencia B, cuya expresión se ha detallado más arriba, crece de forma lineal, siendo razonable que esto perjudique a la velocidad del cálculo. Este

Tabla X: Tiempos de ejecución (en segundos) del algoritmo Chudnovsky con GMP y OpenMP.

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	0,203	0,113	0,061	0,037	0,027	0,028
20.000	0,980	0,537	0,279	0,150	0,119	0,114
30.000	2,522	1,356	0,698	0,370	0,263	0,227
40.000	4,987	2,660	1,371	0,707	0,548	0,405
50.000	8,520	4,552	2,334	1,195	0,897	0,724
100.000	42,056	21,699	11,010	5,594	3,814	3,031

Tabla XI: Tiempos de ejecución (en segundos) del algoritmo Chudnovsky con MPFR y OpenMP

Prec.	Número de hebras					
	1	2	4	8	12	16
10.000	1,142	0,578	0,293	0,150	0,105	0,082
20.000	6,227	3,152	1,602	0,824	0,669	0,453
30.000	15,694	7,949	4,023	2,355	1,427	1,422
40.000	31,155	15,711	7,967	4,275	3,101	2,157
50.000	52,140	26,288	13,273	6,726	5,118	3,546
100.000	244,573	123,448	62,676	31,897	21,660	19,681

análisis permitiría concluir que el reparto por bloques de tamaño uniforme no sea la mejor opción para este algoritmo, ya que las hebras o procesos de menor índice realizan su cálculo más rápidamente y se mantienen ociosas hasta que las otras hebras o procesos terminen. Por tanto, se ha optado por asignar a cada hebra un número de iteraciones diferente a partir de una estimación del coste temporal de cada una. Cabe tener en cuenta que este problema sólo se presenta con la librería GMP y, por ello con la librería MPFR la serie se implementa con un reparto de las iteraciones por bloques de tamaño uniforme.

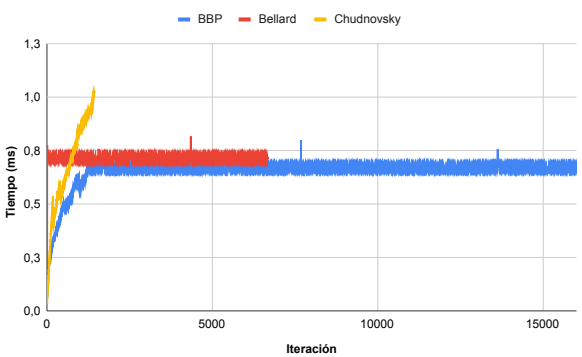

Fig. 10: Coste de las iteraciones de los distintos algoritmos con la librería GMP.

De la misma forma que se ha presentado para los algoritmos previos, las Tablas X y XI y las Figuras 11 y 12 muestran los resultados obtenidos con la implementación multihebra de Chudnovsky con GMP y MPFR. Por su parte, las Tablas XII y XIII muestran los tiempos de ejecución en segundos con MPI utilizando múltiples procesos, mientras que las Figuras 13 y 14 muestran los resultados de escalabilidad. Esta vez, se produce el efecto contrario a lo detectado en los algoritmos previos ya que la versión con GMP obtiene mejores tiempos de ejecución y es MPFR quien obtiene mejores resultados de escalabilidad.

Tabla XII: Tiempos de ejecución (en segundos) del algoritmo Chudnovsky con GMP y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	0,200	0,039	0,027	0,024	0,026	0,026
20.000	0,981	0,148	0,087	0,057	0,067	0,062
30.000	2,516	0,354	0,199	0,125	0,113	0,111
40.000	4,964	0,733	0,373	0,214	0,182	0,166
50.000	8,476	1,160	0,613	0,344	0,279	0,250
100.000	41,536	5,465	2,822	1,463	1,053	1,053

Tabla XIII: Tiempos de ejecución (en segundos) del algoritmo Chudnovsky con MPFR y MPI.

Prec.	Número de procesos					
	1	8	16	32	48	64
10.000	1,138	0,156	0,089	0,052	0,040	0,041
20.000	6,240	0,803	0,415	0,218	0,153	0,166
30.000	15,722	1,996	1,017	0,521	0,361	0,410
40.000	31,195	3,965	1,998	1,096	0,695	0,772
50.000	52,504	6,644	3,372	1,721	1,159	1,261
100.000	246,54	31,380	15,738	7,997	5,422	5,177

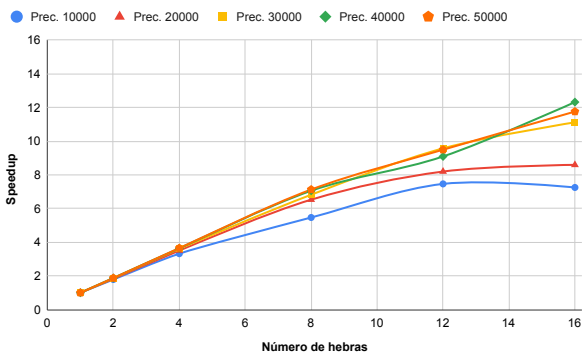

Fig. 11: Escalabilidad del algoritmo Chudnovsky con la librería GMP y OpenMP.

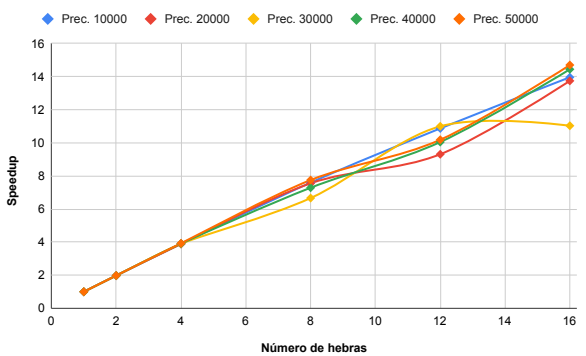

Fig. 12: Escalabilidad del algoritmo Chudnovsky con la librería MPFR y OpenMP.

G. *Paradigma de programación híbrido*

En los apartados anteriores, todos los resultados presentados se han obtenido a partir de la programación paralela con hebras o procesos exclusivamente, pero existe un tercer paradigma de programación paralela que combina ambos recursos. Este se conoce como modelo o paradigma de programación híbrido y consiste en generar procesos que, a su vez, generan hebras para realizar localmente sus cálculos.

Los resultados obtenidos a partir de la combinación de MPI y OpenMP no han sido los esperados. Su análisis ha podido detectar un problema asociado

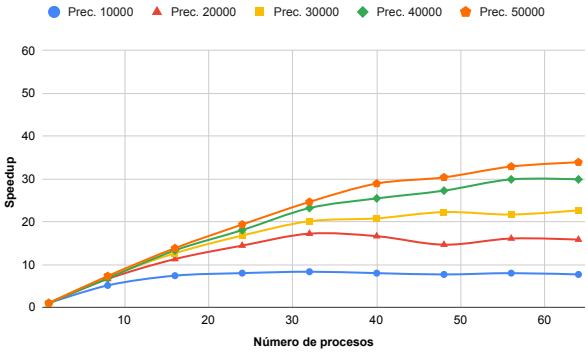

Fig. 13: Escalabilidad del algoritmo Chudnovsky con la librería GMP y MPI.

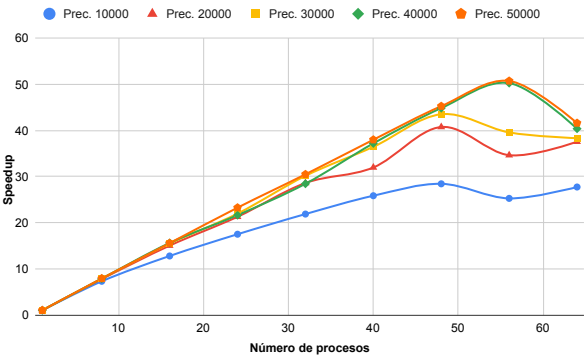

Fig. 14: Escalabilidad del algoritmo Chudnovsky con la librería MPFR y MPI.

a la gestión de recursos, ya que no es posible aprovechar los recursos del sistema de la misma forma que lo hacen los procesos o las hebras trabajando de forma separada. La Tabla XIV y la Figura 15 muestran los tiempos de ejecución y la escalabilidad del algoritmo BBP con MPFR empleando el modelo de programación híbrido. En la Tabla XIV se puede apreciar la caída de las prestaciones cuando se asigna un gran número de hebras por proceso y, por otra parte, la Figura 15 muestra como la escalabilidad no alcanza valores próximos a los obtenidos cuando se utiliza exclusivamente procesos.

Cabe destacar que sólo se ha presentado una tabla de resultados y una gráfica de escalabilidad de los resultados híbridos, pero este efecto se repite en todos los códigos (en mayor o menor medida) sin importar el algoritmo o librería. Por esta razón los resultados híbridos no se han tenido en cuenta en la comparación y valoración general de los resultados.

H. *Resultados globales*

En este último apartado se presenta una visión global de los resultados obtenidos comparando las librerías y los algoritmos *Spigot* utilizados para el cálculo de π.

En primer lugar, la Figura 16 presenta los resultados de las implementaciones secuenciales y multihebra con los distintos algoritmos y librerías. Como se puede observar, destaca la implementación de BBP y Bellard con la librería MPFR y la implementación de Chudnovsky con la librería GMP. Esta misma clasifi-

Tabla XIV: Tiempos de ejecución (en segundos) del algoritmo BBP con MPFR y programación híbrida.

Prec.	Número de procesos												
	1						2			3		4	
	Número de hebras por proceso												
	1	2	4	8	12	16	8	12	16	12	16	12	16
10.000	0,35	0,18	0,10	0,06	0,05	0,04	0,05	0,03	0,03	0,02	0,03	0,02	0,02
20.000	1,37	0,71	0,44	0,28	0,24	0,20	0,17	0,12	0,11	0,09	0,08	0,07	0,08
30.000	3,05	1,55	0,91	0,95	1,15	1,14	0,56	0,60	0,64	0,42	0,43	0,35	0,34
40.000	5,49	2,74	1,69	2,25	2,59	2,83	1,24	1,41	1,40	0,94	0,99	0,72	0,73
50.000	8,40	4,29	2,79	4,14	4,66	4,91	2,26	2,34	2,55	1,67	1,69	1,28	1,31

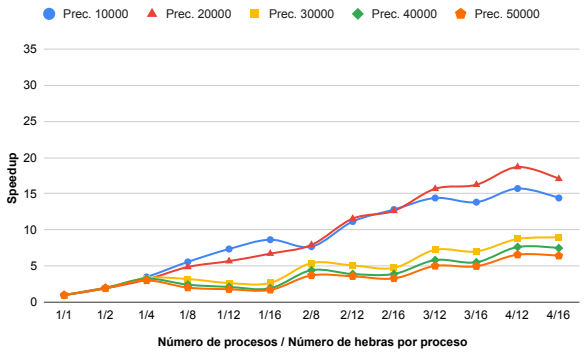

Fig. 15: Escalabilidad del algoritmo BBP con la librería MPFR y el modelo de programación híbrido.

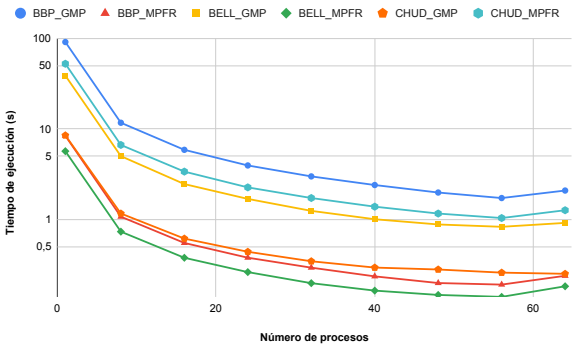

Fig. 17: Tiempos de ejecución de las implementaciones multiproceso (MPI) para obtener 50.000 decimales de π.

cación, se repite para la versión multiproceso (MPI) representada en la Figura 17.

En segundo lugar, también se puede representar cuál ha sido la mejor librería según el algoritmo y paradigma en lo que respecta a los mejores tiempos de ejecución. Esto mismo es lo que se detalla en la Tabla XV, donde se observa que, en cuatro de las seis combinaciones, es la librería MPFR la que realiza el cálculo de la constante π de forma más rápida.

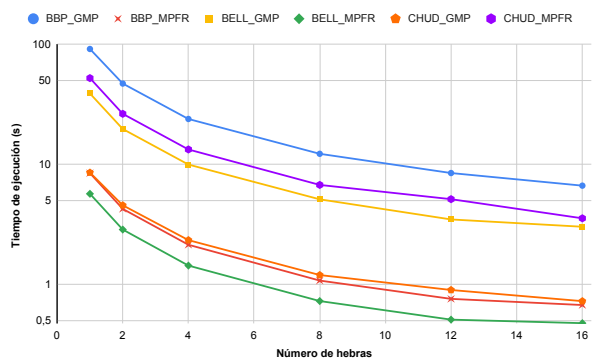

Fig. 16: Tiempos de ejecución de las implementaciones multihebra (OpenMP) para obtener 50.000 decimales de π.

Tabla XV: Mejores tiempos de ejecución obtenidos según algoritmo y paradigma.

Algoritmo	Paradigma	Mejor librería
BBP	OpenMP	MPFR
BBP	MPI	MPFR
Bellard	OpenMP	MPFR
Bellard	MPI	MPFR
Chudnovsky	OpenMP	GMP
Chudnovsky	MPI	GMP

En tercer lugar, a partir de los tiempos de ejecución presentados a lo largo de esta sección, las Figuras 18 y 19 comparan los resultados de escalabilidad de las implementaciones con OpenMP y MPI, respectivamente. En ambas destacan ligeramente las implementaciones de BBP y Bellard con la librería GMP y Chudnovsky con la librería MPFR.

En cuarto lugar, evaluando según la eficiencia se expresa esta misma clasificación tal y como se observa en la Figura 20 para OpenMP y en la Figura 21 para MPI donde la librería GMP obtiene resultados más próximos a uno en dos de los tres algoritmos propuestos en el estudio. En el cómputo global de la escalabilidad y eficiencia, la librería GMP mejora los resultados de la librería MPFR ya que cuatro de las seis implementaciones obtiene de media resultados mejores, tal y como se presenta en la Tabla XVI.

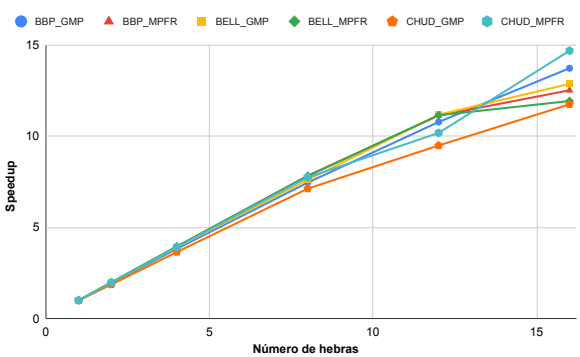

Fig. 18: Escalabilidad de las diferentes implementaciones con OpenMP para la obtención de 50.000 decimales.

IV. Conclusiones

El cálculo de la constante trascendental π supone sin duda un problema muy interesante para introducirse en la Computación de Altas Prestaciones. A

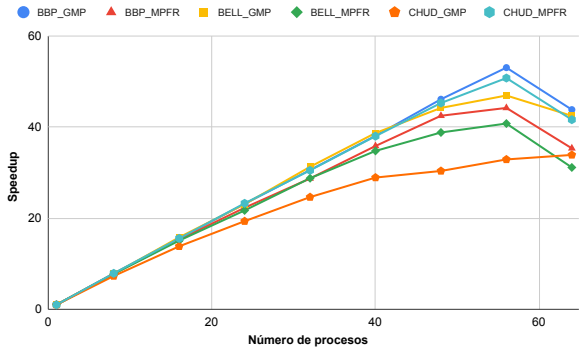

Fig. 19: Escalabilidad de las diferentes implementaciones con MPI para la obtención de 50.000 decimales.

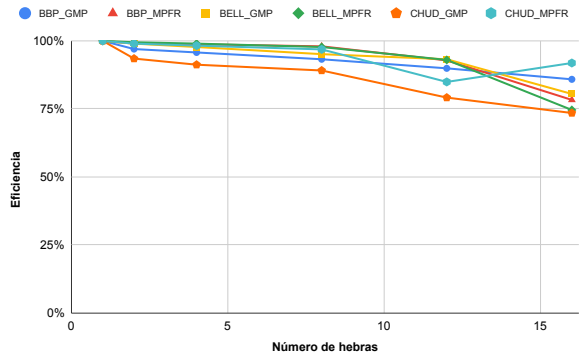

Fig. 20: Eficiencia de las diferentes implementaciones con OpenMP para la obtención de 50.000 decimales.

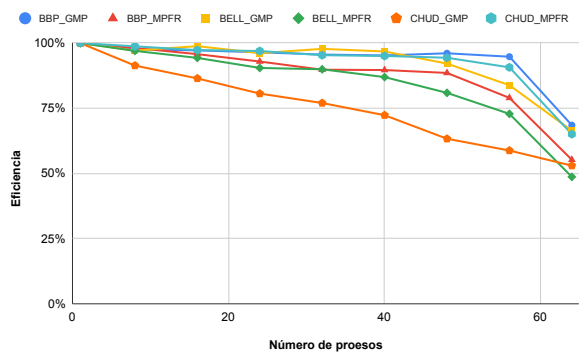

Fig. 21: Eficiencia de las diferentes implementaciones con MPI para la obtención de 50.000 decimales.

Tabla XVI: Mejor escalabilidad y eficiencia media obtenida según algoritmo y paradigma.

Algoritmo	Paradigma	Mejor librería
BBP	OpenMP	GMP
BBP	MPI	GMP
Bellard	OpenMP	GMP
Bellard	MPI	GMP
Chudnovsky	OpenMP	MPFR
Chudnovsky	MPI	MPFR

partir de los resultados obtenidos se pueden deducir las conclusiones que se comentan a continuación.

Por un lado, MPFR obtiene habitualmente mejores tiempos de ejecución. Esta librería asegura que los resultados de cualquier operación son los más cercanos posibles al resultado exacto, y además también proporciona una variabilidad de técnicas de redondeo. Por su parte, GMP obtiene mejores resultados en lo que respecta a eficiencia y escalabilidad de los algoritmos. Esto tiene sentido ya que el principal objetivo de esta librería es realizar los cálculos de la forma más eficiente posible. Además, a pesar de no tratarse de una librería exclusiva para operaciones con números en coma flotante, ofrece un amplio abanico de funciones. Si bien es cierto que GMP no asegura redondeo exacto, en ningún momento durante el transcurso del estudio esto ha supuesto un problema para el cálculo de la constante trascendental. En general, ambas librerías son altamente recomendables para la resolución de este tipo de problemas. Su codificación es muy parecida y cuentan con una detallada y amplia documentación.

Para futuros estudios sería muy interesante analizar y resolver el problema empleando GPU y librerías asociadas a las unidades de procesamiento gráfico. También se podría considerar el uso de la computación o algoritmos *out-of-core* para calcular una mayor cantidad de decimales e intentar batir el récord mundial de decimales calculados de la constante π. Por último, otro aspecto muy interesante sería estudiar el problema que se produce en la gestión de recursos de los algoritmos híbridos y analizar en detalle el motivo de la caída de prestaciones. Sin duda, se trata de un trabajo que abre un gran abanico de posibilidades de estudio, mejora y desarrollo tecnológico.

Agradecimientos

El trabajo realizado por José I. Aliaga ha sido subvencionado a través del proyecto PID2020-113656RB-C21 financiado por MCIN/ AEI/10.13039/501100011033. Por su parte, el trabajo de Miguel Pardo fue subvencionado por una beca de colaboración para la iniciación a la investigación en el Departamento de Ingeniería y Ciencia de los Computadores (convocatoria 2021).

Referencias

[1] Simon Reif Acherman, "El Número PI y su Historia," *Ingeniería y Competitividad*, vol. 2, no. 2, pp. 47, 2011.
[2] Michael J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C, no. 9, pp. 948–960, 1972.
[3] "The GNU MP Bignum Library," `https://gmplib.org/`.
[4] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann, "MPFR," *ACM Transactions on Mathematical Software*, vol. 33, no. 2, pp. 13, 2007.
[5] "The GNU MPFR Library," `https://www.mpfr.org/`.
[6] "OpenMP: Home," `https://www.openmp.org/`.
[7] "Open MPI: Open Source High Performance Computing," `https://www.open-mpi.org/`.
[8] Stanley Rabinowitz and Stan Wagon, "A Spigot Algorithm for the Digits of π," *The American Mathematical Monthly*, vol. 102, no. 3, pp. 195–203, 1995.
[9] David Bailey, Peter Borwein, and Simon Plouffe, "On the rapid computation of various polylogarithmic constants," *Mathematics of Computation*, vol. 66, no. 218, pp. 903–913, 1997.
[10] F Bellard, "A new formula to compute the n'th binary digit of pi," `https://bellard.org/pi/pi_bin/pi_bin.html`, 01 1997.
[11] F Bellard, "Computation of 2700 billion decimal digits of Pi using a Desktop Computer," `https://bellard.org/pi/pi2700e9/pipcrecord.pdf`, 02 2010.