Práctica 1: Programa MPI simple e caracterización do rendemento

Fundamentos de Sistemas Paralelos

Daniel Fuentes Rodríguez
daniel.fuentes.rodriguez@rai.usc.es

Tomás García Barreiro
tomas.garcia.barreiro@rai.usc.es

Índice

1. Introdución

Nesta memoria mostraremos os códigos e técnicas usados para o cálculo de π a través da fórmula inferior usando paralelización con MPI a través tanto do cálculo dunha integral por trapecios como da integración de Monte Carlo.

$$\pi = \left(\int_{-\infty}^{\infty} e^{-x^2} dx \right)^2 \text{ (Colaboradores de Wikipedia, 2023a)}$$

2. Código

O código, dispoñibles nos anexos 2 e 3, pódese dividir en catro seccións diferenciadas.

A primeira é a inicialización dos nodos, a segunda é o cálculo da integral por cada nodo, a terceira é o paso de mensaxes entre nodos e a última é o gardado dos datos nun ficheiro CSV e a finalización dos nodos.

Neste apartado, explicaremos o funcionamento de todas as seccións, centrándonos principalmente na segunda e terceira, ó ser as que máis interese teñen.

2.1. Inicialización e Determinación do Traballo para os Nodos

En primeiro lugar, definimos un primeiro apartado do código que inicialice variables e determine o traballo de cada nodo.

Nesta primeira sección será onde se inicialicen os límites da integral para o cálculo de π.

Á súa vez, é importante destacar que se estableceu un valor de referencia de π aportado polo profesor. Ademais inicialízase tamén o número de iteracións a realizar para cada nodo. Deste xeito, poderemos escalar o número de iteracións segundo vexamos convinte.

No caso de estar usando o método dos trapecios, estableceremos o rango de traballo en x de cada nodo dividindo o rango establecido de maneira equitativa.

En consecuencia, cada nodo executará o número de iteracións establecido no seu rango concreto. Desta maneira, mediante o paso de mensaxes de resultados entre os nodos, poderemos agrupar o espectro completo e obter o valor de π.

Se en cambio, estamos facendo o cálculo seguindo o método de Monte Carlo, non definiremos un rango de x onde traballará cada nodo, os puntos xeraranse de forma aleatoria na área total definida de forma xeral. Neste caso, xeraremos tantos puntos como número de iteracións se definan.

2.2. Cálculo da Integral

Neste apartado abordaremos principalmente o cálculo do valor de π mediante a integral reflexada na sección anterior.

Como podemos observar na fórmula presentada previamente, no noso caso teremos que obter a área comprendida entre $-\infty$ e ∞ da campá de Gauss. Sin embargo, isto non é posible, polo que teremos que delimitar este rango.

Para que sexa máis sinxelo a visualización da función a tratar, móstrase a continuación a súa representación gráfica no rango [−10,10]:

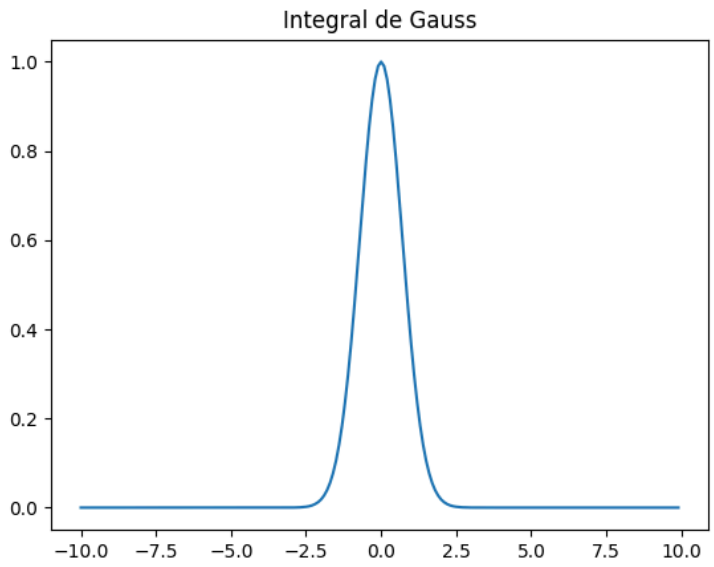

Figura 1: Integral de Gauss no rango [-10,10]

Como podemos ver no gráfico, conforme máis nos afastemos do (0,0), menor será a área a obter, polo que menor importancia terá o cálculo da mencionada área respecto da precisión de π.

Imos considerar dous rangos para o cálculo da integral en función dos diferentes métodos.

En primeiro lugar, o establecido na gráfica presentada, é dicir, o rango [−10,10], será o empregado polo cálculo integral baseado en trapecios.

Por outro lado, utilizaremos o rango [−5,5] para o cálculo integral baseado no método de Monte Carlo. Tamén é importante destacar que neste método teremos que limitar a altura sendo o seu valor máximo $y = 10$.

Tomáronse estos valores porque deste xeito teremos unha área total de 100 unidades onde a área representada pola gráfica da función será $\pi\%$. Desta maneira, obteremos o valor de π mediante cálculos máis simples dos que teríamos co rango de trapecios.

Á súa vez, consideramos que é importante destacar a importancia do número de iteracións definido no apartado anterior. Como sabemos, conforme maior sexa este número, maior será a precisión no cálculo da área integrada na función.

Representando os métodos a empregar, no caso dos trapecios, canto maior sexa o número de trapecios, mellor será a estimación en canto á área real da gráfica.

Igual pasará co método de Monte Carlo, obteremos mellores resultados conforme maior sexa a nube de puntos utilizada.

En consecuencia, cremos que terá maior relevancia o número de iteracións a realizar polos métodos de estimación empregados fronte ao rango establecido no cálculo integral. Aínda así, todas estas son suposicións iniciais que se veran amosadas nos resultados, onde mediante táboas e gráficos buscaremos darlle sentido ás hipóteses presentadas.

2.3. Integración por Trapecios

Nesta sección, tomaremos como referencia o **Anexo 2**, no que daremos sentido ao código presentado. Neste caso, usaremos o método de cálculo integral de áreas por trapecios. Centrarémosnos en determinar o espazo comprendido baixo a gráfica presentada anteriormente mediante a aproximación por trapecios [ver Bibliografía].

Como comentamos anteriormente, estes trapecios poderán estimar mellor á gráfica conforme o número de trapecios tenda a ∞.

Neste caso, teremos tantos trapecios como iteracións se queiran establecer, onde a base de cada trapecio será o rango total de traballo para o nodo entre o número de iteracións. Deste xeito, cada nodo dividirá o seu rango en tantos trapecios como iteracións se definan.

Para o cálculo dos trapecios, estableceremos as alturas y_1 e y_2, sendo respectivamente a altura no comezo do trapecio e no final. Para o cálculo de ambas aplicaremos a función definida como `gaussian()` que representa a campá de Gauss que buscamos integrar.

Unha vez obtidas as alturas, faremos o cálculo do trapecio mediante o cálculo equivalente dun rectángulo e un triángulo. Por un lado, o rectángulo estará definido pola base de todo o trapecio e a altura será o valor de y_1.

Por outro lado, o triángulo definirase con base a do trapecio e altura a diferenza absoluta entre y_1 e y_2 (altura ao comezo e ao final).

Definirase desta maneira o trapecio como o rectángulo ao que lle engadiremos ou quitaremos a área do triángulo en función de se y_1 é maior ou menor que y_2.

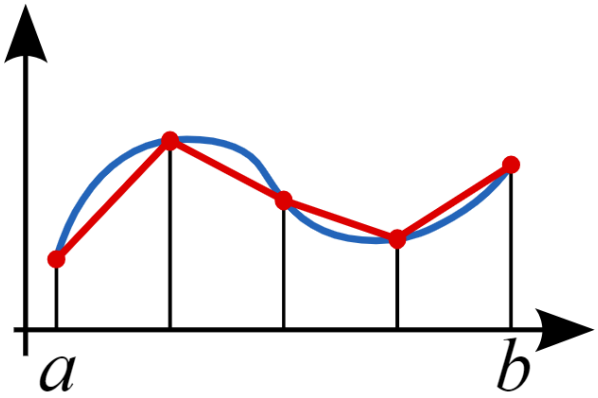

Figura 2: Regla do Trapecio

No caso de ter unha altura inicial inferior, teremos que engadir este triángulo como representación da pendente aumentada respecto da altura inicial. En caso contrario, teremos que substraer a diferenza de altura representando a pendente negativa do trapecio.

2.4. Integración por Monte Carlo

A integración por Monte Carlo consiste na xeración de puntos aleatorios dentro dunha área definida de antemán. Estes puntos clasificaranse en dous grupos, aqueles que caeron fóra da área delimitada pola función e aqueles que caeron dentro. Unha vez finalizada a xeración, podemos calcular a área como a relación entre os puntos que caeron dentro e o número total de puntos xerados.

Este método presenta un problema inicial, que é que precisa definir os límites tanto en x coma en y, creando un rectángulo dentro do cal serán xerados tódolos puntos.

No noso caso, o rango de x será [−5,5], mentres que o de y será [0,10]. Desta forma, como xa explicamos anteriormente, a área delimitada pola función será de, aproximadamente, un $\pi\%$ da área total definida.

Para a xeración destes puntos aleatorios o único que facemos é definir unha *seed* diferente para cada nodo e, despois, usamos a función `rand()` de C, xunto cuns poucos cálculos, para conseguir os valores de x e y dentro do cadrado.

Unha vez feito isto, chamamos á función `gaussian()`, pasándolle o valor de x e vendo se o valor que devolve en y é maior que o calculado aleatoriamente.

Se este é o caso, aumentamos en 1 o valor da variable que leva a conta dos puntos aleatorios xerados dentro da función.

Executamos estes pasos tantas veces como iteracións teñamos definidas e, unha vez finalizadas, pasamos o número de puntos xerados dentro da función ata ter no nodo 0 o total.

Cando teña isto, o nodo 0 atopará $\sqrt{\pi}$ simplemente calculando a relación entre o número de puntos dentro da función e o número de puntos total que, como cada nodo executa o mesmo número de iteracións, será sinxelo de atopar.

Cabe destacar que o que se calcula é a relación con respecto ó xeral, polo que o resultado será un número menor de 1 e haberá que multiplicalo por 100 para obter o valor buscado.

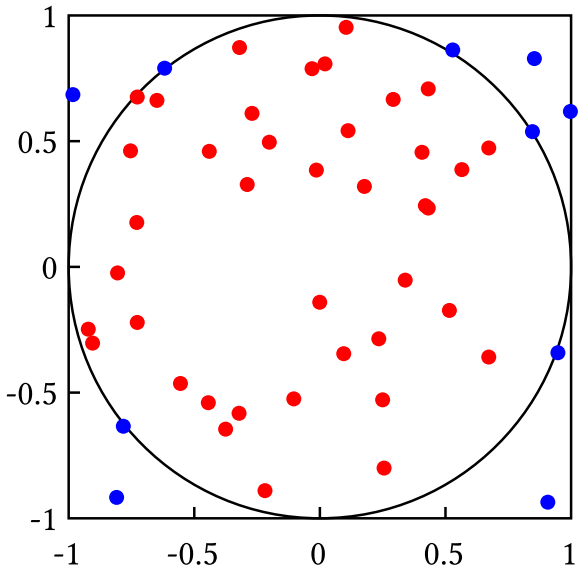

Figura 3: Método de Monte Carlo

2.5. Paso de mensaxes

O paso de mensaxes é a función primaria de MPI e, neste caso, tiñamos que resolver o problema de pasar os datos calculados por tódolos nodos para xuntalos nun e conseguir o valor de π.

En primeiro lugar, especificaremos un aspecto a ter en conta no paso de mensaxes, os rangos. Como sabemos, o traballo con MPI require identificar cada nodo cun rango único, para así poder comunicarse entre eles de forma sinxela. No noso caso, o grupo por defecto de MPI, `MPI_COMM_WORLD`, foi máis que suficiente, xa que tódolos procesos executan o mesmo código e o único que queremos é diferencialos ó final de todo, para agrupar tódolos datos nun só nodo.

A solución sinxela sería pasar os datos do nodo n ó nodo $n-1$, sumalos en $n-1$, pasalos a $n-2$ e así sucesivamente ata chegar ó nodo 0. Esta forma de pasar os datos presenta un grave problema, xa que o tempo que se tarda dende o paso da primeira mensaxe ata o cálculo final de π é $O(n)$.

Por iso mesmo, o paso de mensaxes no código é na forma dunha árbore binaria, ou sexa, o nodo n envíalle o seu dato ó nodo 0, o $n-1$ ó 1, e así ata o nodo $n/2$, que lle envía os seus datos a $n/2-1$. Na seguinte iteración, consideramos n como $n/2-1$, e repetimos o mesmo proceso que anteriormente, ata que o único nodo con tódolos datos sexa o 0.

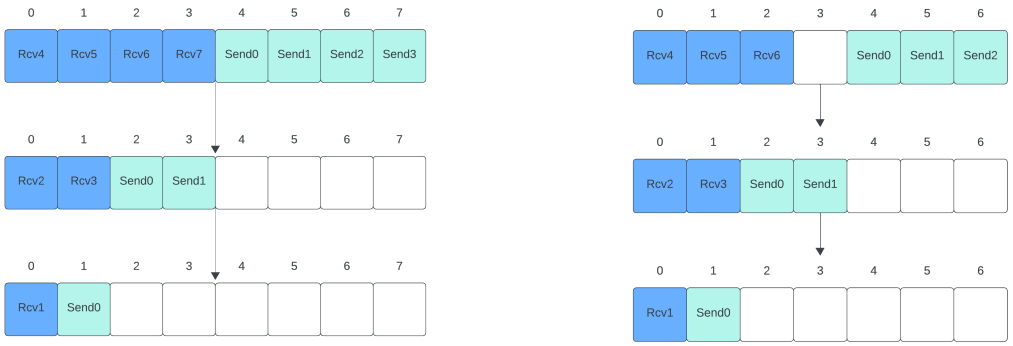

Figura 4: Paso de mensaxes con 8 e 7 nodos

Como podemos ver na imaxe, no caso onde $n/2$ sexa un valor impar, na primeira iteración este nodo non enviará nin recibirá nada. Deste xeito, comezará a enviar mensaxes a partires da segunda iteración, donde o número de nodos traballando no paso de mensaxes sempre será par.

Se ben esta implementación garantiza unha complexidade temporal $O(\log n)$, atopamos un problema ó poder usar só `MPI_Send()` e `MPI_Recv()`, xa que un nodo pode mandar a súa mensaxe antes de invocar o outro a función para recibir.

Neste caso, arranxamos este problema engadindo un `nanosleep()` dun milisegundo antes do *send* dos nodos que envían, o que significa que a mellora temporal real non será tan grande como a teórica.

Unha forma de arranxar este problema sería implementando o envío dun ACK por parte do nodo que recibe, permitindo borrar o `nanosleep()` que temos e diminuíndo así o tempo de execución do noso programa. Para isto, usaríanse as funcións `MPI_Isend()` e `MPI_Irecv()`, non bloqueantes, o que permitiría ó nodo que envía mandar a mensaxe e agardar un tempo t a recepción dun ACK. Se pasado ese tempo non o recibe, volvería outra vez mandar a mensaxe.

2.6. Tratado dos datos

Unha vez remata a execución do código, queda un ficheiro CSV que é preciso tratar. Por un lado, porque garda os datos co *locale* español, polo que usa a coma decimal en vez do punto. Por outro lado, porque garda o número de nodos MPI nos que se executa en cada iteración, pero non o número de nodos do CESGA, que tal vez pode ser interesante ter en conta para a posterior análise.

Por iso mesmo, creáronse dous ficheros que axudan no tratamento dos datos.

O primeiro deles é un *script* de Python [**Anexo 5**] que recibe como *input* un ou varios CSV, nos que remplaza as comas decimais por puntos e despois xunta ordenando por nodos do CESGA e, dentro de cada un deles, por nodos MPI.

Este código resulta interesante se se ten, como é o noso caso, varios ficheiros de datos para cada método, xa que os xunta rapidamente e dun xeito sinxelo para a súa posterior análise.

Por outro lado, utilizouse un *Jupyter Notebook* no que se tomaron os datos xa tratados e se definiron gráficos e táboas que se amosarán ao longo do apartado seguinte. Debido á complexidade da inclusión como Anexo do código, destacaremos que este *Notebook* se atopa incluído dentro dos códigos entregados co informe, igual que o resto dos anexos.

3. Análise dos Resultados

Antes da análise dos resultados obtidos tras tratar e traballar os datos obtidos dende o Finisterrae III, é importante destacar unha diferenciación léxica.

Falaremos dos «Nodos MPI» co termo de CPUs, sendo un termo diferenciado do uso de «Nodos», que empregaremos para os nodos do CESGA, compostos por máis dunha CPU.

Para a análise de resultados, divideremos esta sección en 3 apartados diferenciados.

En primeiro lugar, trataremos os resultados obtidos para a integración por trapecios. Despois, falaremos dos resultados obtivos para a integración polo método de Monte Carlo. Por último, faremos unha comparativa de ambos métodos respecto da calidade con diferente número de iteracións. Para este último análise, tomouse o mesmo valor de CPUs e Nodos, para obter resultados no mesmo contexto.

3.1. Integración por Trapecios

Empezando pola integración por trapecios, como vemos na figura inferior, os datos relativos á execución do código cun só nodo MPI (polo tanto, sen usar o paso de mensaxes) presentan uns resultados anómalos comparados co resto.

Neste caso, podemos ver que a calidade amosada respecto do cálculo integro de todo o rango de valores fronte á división do traballo en diferentes nodos, é moi superior.

Isto pode deberse a unha combinación de dous factores. Por unha banda, é un número de iteracións pequeno pero razonable para obter un cálculo bo de π, polo que non debería haber problemas coa precisión do cálculo con punto flotante. Por outra, simple sorte, facendo que ó escollermos ese rango específico con ese número de iteración o resultado calculado sexa excepcionalmente bo.

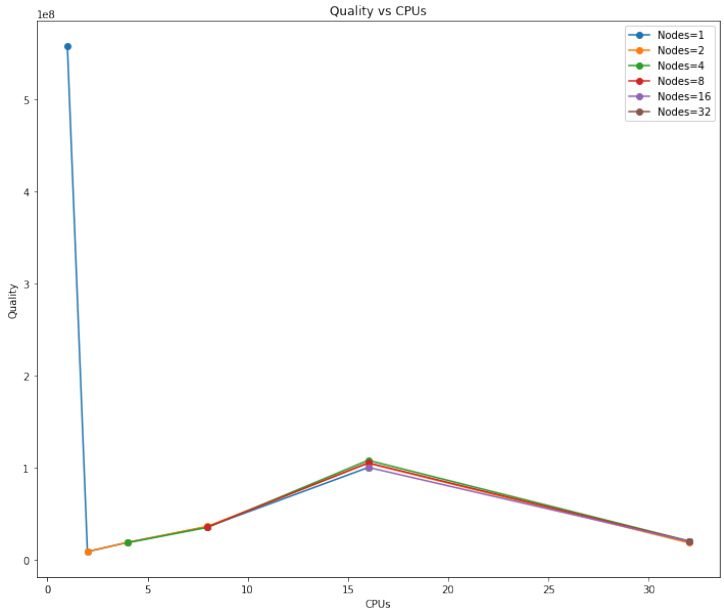

Figura 5: Calidade segundo o número de CPUs na integración por trapecios

Neste caso, faremos énfase nos resultados obtidos mediante o uso de varios nodos MPI. É dicir, centrándonos no paso de mensaxes e a paralelización, sendo os aspectos fundamentais da práctica.

Observando o gráfico inferior, é posible observar que os mellores resultados son con 16 CPUs. Sin embargo, a calidade segundo o número de nodos non varía moito pero, polo que podemos observar, o par conformado por 4 nodos e 16 CPUs, amosa os mellores resultados en canto a calidade.

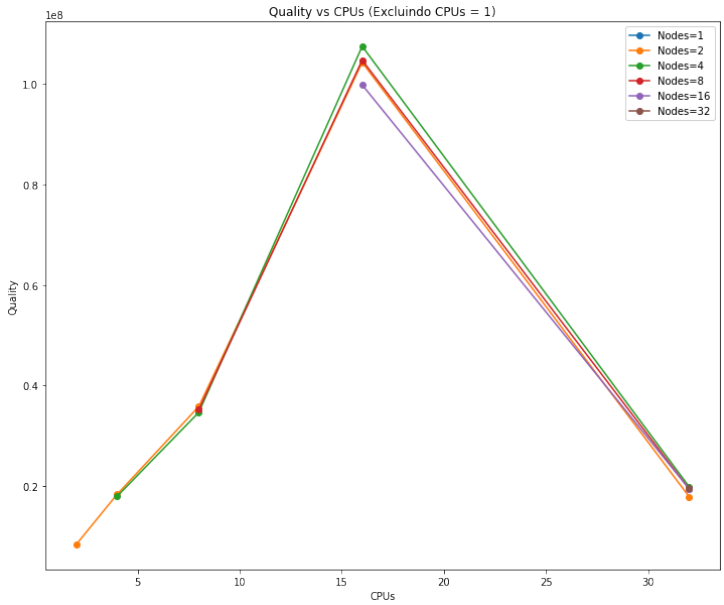

Figura 6: Calidade segundo o número de CPUs na integración por trapecios

Se nos fixamos nos tempos de execución obtidos para diferentes números de nodos, podemos observar as tendencias crecentes conforme aumentamos o número de CPUs. Isto é lóxico dado que como comentamos anteriormente, o uso de `nanosleep()` xunto co paso de mensaxes no doble de nodos implica un tempo de espera relativamente grande.

En consecuencia, observamos como o paso de mensaxes representa un aspecto moi importante no coste temporal da execución, sobre todo para un menor número de nodos do supercomputador.

Faría falta un maior número de CPUs para poder observar as tendencias claras para un maior número de nodos do supercomputador. Neste caso, polos resultados obtidos, non queda moi claro os motivos reais da vantaxe obtida en tempos usando 4 nodos respecto do resto.

Como hipótese, podemos engadir que pode existir unha influencia no reparto de CPUs nos nodos dentro do supercomputador, sendo a disposición o que beneficie o resultado para 4 nodos.

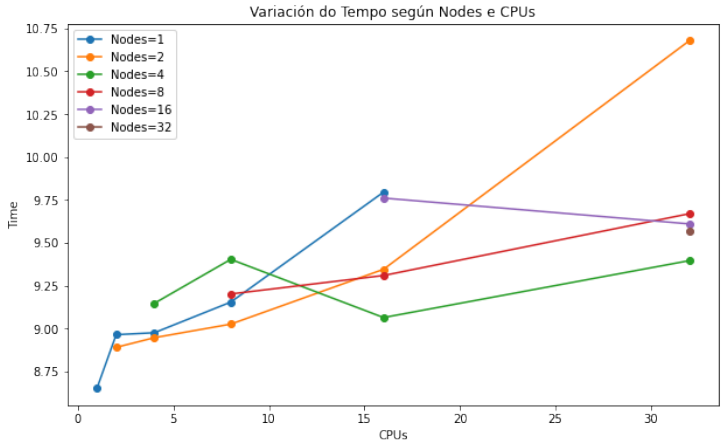

Figura 7: Tempo de Execución segundo o número de CPUs na integración por trapecios

É importante destacar que os mellores resultados obtidos, quitando o caso de traballar cun só nodo MPI, obtivéronse con 4 Nodos e 16 CPUs. Esta información xunto aos resultados e tendencias obtidas poden ser de interese para o análise de Monte Carlo e a determinación con mais precisión de hipóteses que nos expliquen o rendemento obtido.

3.2. Integración por Monte Carlo

Neste apartado, analizaremos en detalle os resultados obtidos para a integración polo método de Monte Carlo. Como apunte inicial e fundamental que poida influír nos resultados obtidos, é importante recordar o feito de que Monte Carlo é un método que emprega números aleatorios. Deste xeito, podemos obter resultados que se vexan influenciados polo azar.

Comezando co análise, na gráfica seguinte, podemos ver a calidade obtida para diferentes nodos e CPUs. Neste caso, e como anticipabamos anteriormente, observamos que para 4 nodos e 8 CPUs, obtivemos un resultado atípicamente moi bo.

Curiosamente, coincide co mesmo valor de nodos para os cales no apartado anterior observamos unha mellora temporal atípica.

Se seguimos observando o gráfico, podemos observar unha posible tendencia descendente da calidade se pasamos de 16 a 32 CPUs. Da mesma forma que comentamos no apartado anterior, esta tendencia pode non cumplirse para un maior número de CPUs, sendo precisa a obtención de resultados para un número máis grande de CPUs.

De maneira xeral, podemos engadir que a mellor calidade obtida obsérvase para 8 e 16 CPUs, sendo valores similares aos obtidos con Trapecios.

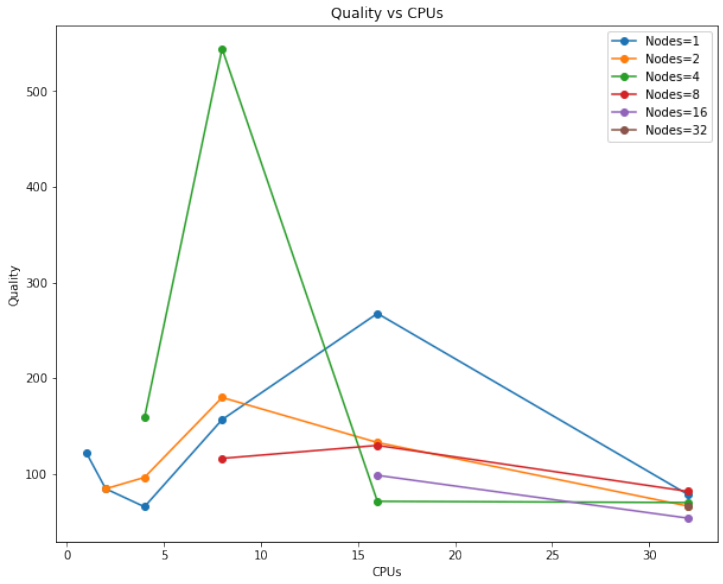

Figura 8: Calidade do Resultado segundo o número de CPUs na integración por trapecios

Na imaxe anterior, podiamos observar en detalle os resultados en termos medios. A continuación, amosamos o mesmo gráfico engadindo a desviación típica para poder determinar a variabilidade posible dos resultados obtidos.

Observamos, en consecuencia, como existe unha desviación moi acentuada para os pares (4 nodos, 8 CPUs) e (1 nodo, 16 CPUs). Istos resultados poden ser unha mostra clara da capacidade do azar para

manipular os resultados. Mostra diso é a tendencia na desviación amosada conforme aumentamos o número de CPUs.

Sin embargo, non todo ten que ser azar, existindo a posibilidade de que para 4 nodos existan influencias non contempladas que nos permitan obter mellores resultados.

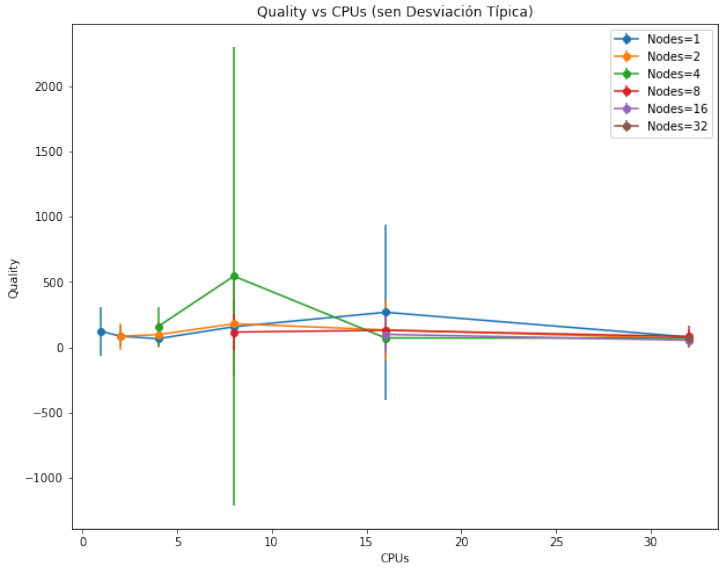

Figura 9: Calidade do Resultado (sen Desviacións) segundo o número de CPUs

Podemos ver como neste caso, a influencia do erro non é tan clara no resultado obtido. Neste caso, podemos ver como para 4 nodos o erro obtido en 4 CPUs é o mais baixo. Sin embargo, para 8 CPUs, observamos que o erro obtido non é tan destacable sendo a principal vantaxe o coste temporal.

É importante destacar que, a diferencia da integración por Trapecios, a falta de precisión en punto flotante non xoga un factor determinante no erro obtido.

Loxicamente, se traballamos con conteos de números aleatorios, é lóxico pensar que a influencia do erro da máquina é despreciable. Principalmente, polo feito de que o cálculo en punto flotante asociado ao valor de π, realizase no nodo 0, unha vez completado o paso de mensaxes.

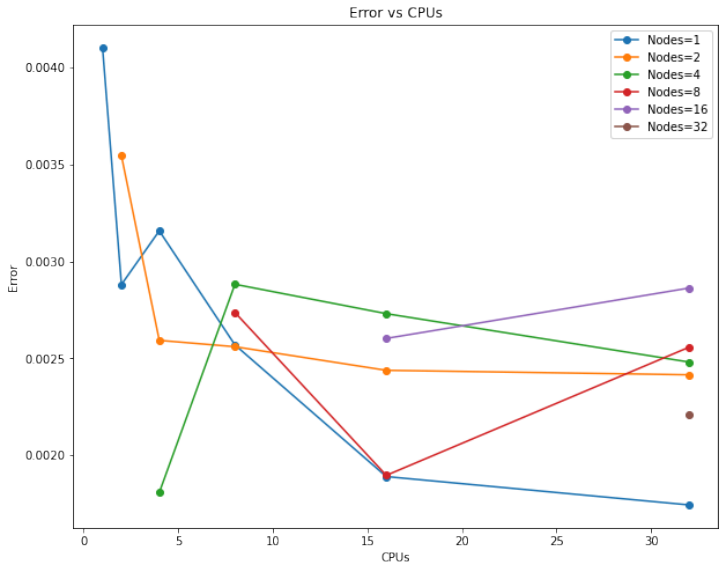

Figura 10: Erro respecto a Pi segundo o número de CPUs na integración por Monte Carlo

Por último, podemos observar no seguinte gráfico cal é a tendencia amosada para o tempo de execución. Neste caso, amósase claramente como para nodos entre 1 e 4, obtemos unha clara tendencia ascendente no tempo requerido para un maior número de CPUs.

Isto pode ser, de novo, pola influencia do coste temporal no paso de mensaxes explicado.

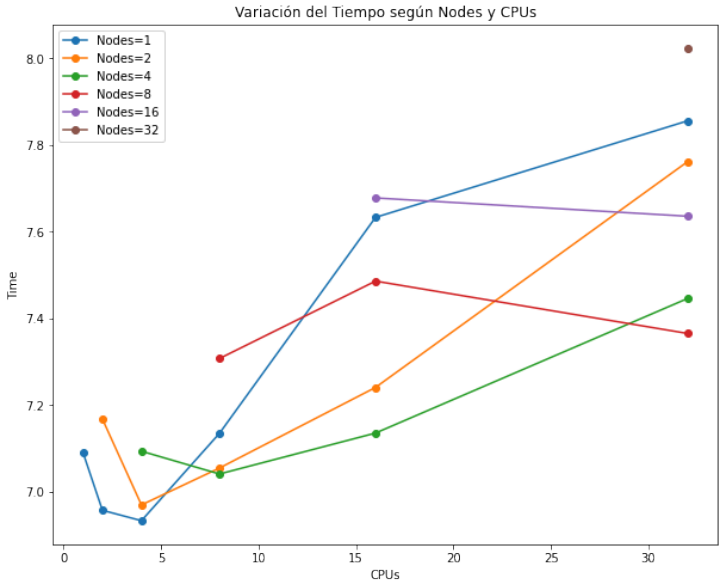

Figura 11: Tempo de Execución segundo o número de CPUs na integración por Monte Carlo

A modo de conclusión xeral obtida, podemos ver que para Monte Carlo síguese a cumplir unha curiosa e posible coincidencia na influencia positiva do uso de 4 nodos. Neste caso, amósanse resultados mellores para 8 CPUs, sendo diferente do obtido en Trapecios (16 CPUs).

3.3. Comparativa por Número de Iteracións

Para a análise de ambos métodos de integración, tomaremos o obtido nos apartados anteriores como referencia e estableceremos un mesmo valor de CPUs e Nodos para a obtención de resultados.

Neste caso, pensamos que pola extraña pero positiva influencia obtida, 4 nodos sería o valor máis correcto. Por outro lado, para o caso do número de CPUs, consideramos que 16 CPUs sería o valor máis correcto, dado que para Trapecios era o mellor resultado.

Non tomamos 8 CPUs dado que pensamos que o resultado obtido en Monte Carlo ten unha clara influencia do azar así como para 16 CPUs os resultados amosado en Monte Carlo son relativamente bos.

Neste caso, a comparativa a realizar farase variando o número de iteracións a realizar, permitindo ver a influencia para ambos métodos de integración da escalabilidade do problema. Aquí, igual que en apartados anteriores, amosaremos a influencia en calidade basándonos no coste temporal e no erro obtido.

En primeiro lugar, amosamos a continuación as 2 táboas obtidas de resultados para diferentes números de Iteracións:

iters	time	err	qual
1	0.924159	2.273799e-02	4.866482e+01
10	0.566680	4.440892e-16	3.975504e+15
100	0.565613	3.552714e-15	4.979422e+14
1000	0.557858	5.151435e-14	3.481680e+13
10000	0.565268	1.869616e-13	9.466448e+12
100000	0.570877	3.556710e-12	4.926733e+11
1000000	0.776519	4.797673e-11	2.833021e+10
10000000	1.587247	1.659015e-10	3.852795e+09
100000000	9.408985	1.026328e-09	1.035734e+08
1000000000	92.857799	8.842800e-08	1.229300e+05

Tabla 1: Calidade do eesultado segundo o número de iteracións na integración por trapecios

iters	time	err	qual
1	0.911423	9.101468	0.302457
10	0.569292	6.023768	0.967785
100	0.548642	0.709098	3.369901
1000	0.569393	0.234194	28.220084
10000	0.561791	0.194563	16.371165
100000	0.574807	0.042993	73.362728
1000000	0.763283	0.006675	440.809529
10000000	1.418241	0.002471	556.060693
100000000	7.752029	0.002621	59.585481
1000000000	71.834936	0.002534	5.659918

Tabla 2: Calidade do eesultado segundo o número de iteracións na integración por Monte Carlo

Sin embargo, para realizar unha explicación mais precisa dos resultados obtidos, apoiarémosnos nas seguintes gráficas. En primeiro lugar, como se pode observar no gráfico seguinte, falaremos da calidade obtida para Monte Carlo.

Como se pode observar, existe unha clara vantaxe en calidade entre 10^6 e 10^7 respecto do resto de valores. Esta vantaxe, pola contra, pode ser claramente un resultado obtido plenamente pola aleatoriedade do algoritmo.

Como aspecto interesante, podemos ver que a calidade é prácticamente nula para un número moi pequeno de iteracións. Lóxico, dado que se necesita unha gran cantidade de puntos aleatorios para poder obter resultados cada vez máis estables. En consecuencia, cunha nube de puntos moi pobre, os resultados amosados son nefastos.

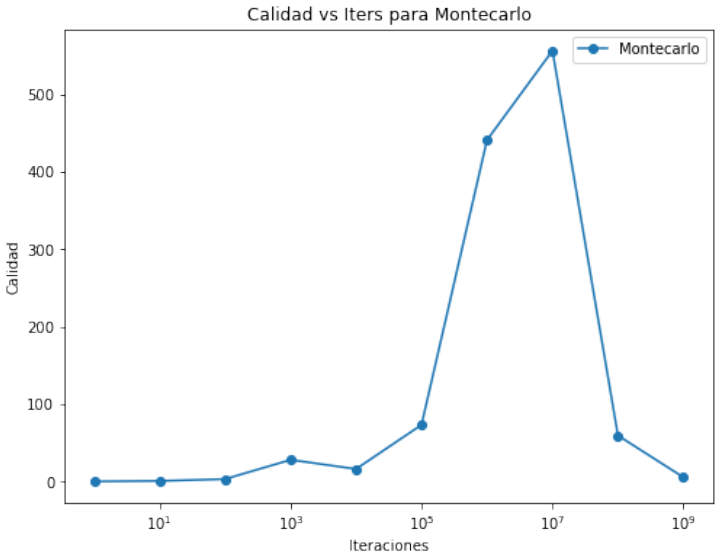

Figura 12: Calidade do resultado segundo o número de iteracións na integración por Monte Carlo

Amosando a continuación os resultados obtidos para a integración por trapecios, podemos ver que a calidade obtida con respecto de Monte Carlo e moito mellor. Simplemente con observar a escala utilizada para ambos gráficos xa podemos entender a magnitude de superioridade deste método de integración.

Ademais, referenciando as táboas amosadas previamente, podemos ver nelas como as ordes de magnitude da calidade son moi superiores respecto de Monte Carlo.

Por outro lado, podemos ver que para 10 iteracións, o resultado obtido é da orde de 10^{15} de calidade. Isto, o máis seguro, é que sexa tamén cuestión de sorte ó escoller o rango de cálculo e o número de iteracións totais, 160, xa que pouco sentido ten que cunha cantidade tan pequena de iteracións o resultado sexa tan bo.

Aínda así, non cabe descartar a posibilidade de que en verdade menos iteracións beneficien ó cálculo, xa que podemos ver polos datos que, mentres as iteracións aumentan de forma exponencial, o erro fai xusto o contrario, diminuir tamén a pasos axigantados (con 100 iteracións por nodo é de $3 * 10^{-15}$ e con 1000 xa pasa a $5 * 10^{-14}$).

Sexa certo ou non que un menor número de iteracións é a mellor opción, o que queda claro é que o método de integración por trapecios é moito mellor que o de integración por Monte Carlo.

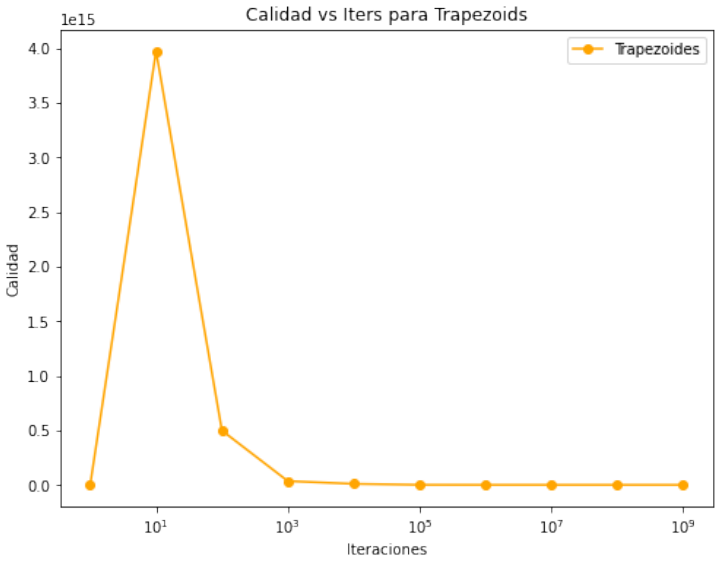

Figura 13: Calidade do Resultado segundo o número de iteracións na integración por trapecios

A modo de comparativa final, podemos observar no seguinte gráfico como os resultados obtidos en calidade en Trapecios son inalcanzables de ningún xeito por Monte Carlo.

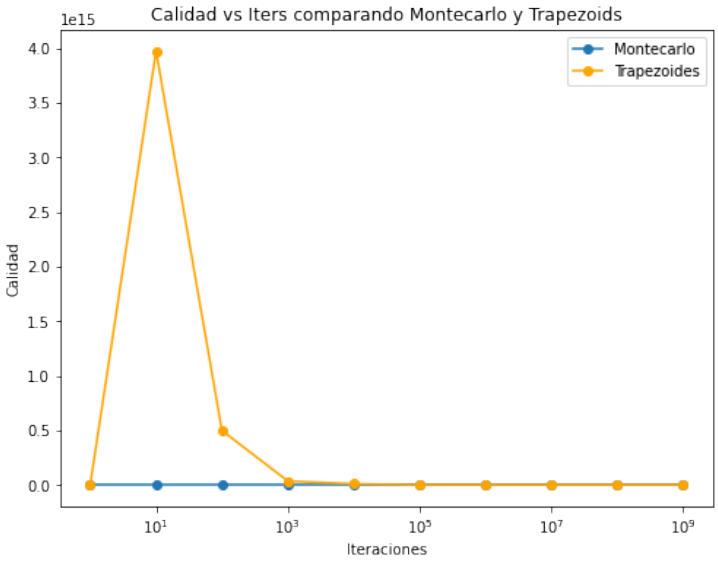

Figura 14: Comparativa da Calidade do Resultado segundo o número de iteracións

4. Conclusións

Tras a realización desta práctica, podemos sacar en claro dous aspectos.

Por unha banda, que a integración por Monte Carlo é inferior á integración por trapecios en varias ordes de magnitude, ademais de que a integración por trapecios se beneficia de ser determinista, vai dar sempre o mesmo resultado, mentres que con Monte Carlo pode que nunha execución a *seed* funcione ó teu favor, otorgando un resultado excelente, e na seguinte fagan o contrario e xeren tódolos puntos fóra da área delimitada.

Por outra banda, queda tamén claro que o número de execucións realizadas non é, nin por asomo, o mínimo necesario para sacar unhas conclusións certeras, xa que se ben podemos falar dos resultados cun número limitado de nodos, non podemos dicir o mesmo cando os nodos superen o centenar.

Sin embargo, non debemos olvidar a principal conclusión a obter de todo este traballo, que é a gran vantaxe que ofrece a paralelización. Mediante o uso de MPI para situacións como a amosada neste informe, podemos ver como mediante o cómputo en paralelo e a sincronización dos nodos, podemos aumentar enormemente o número de execucións total sen perder tempo de execución ao estar paralelizado.

5. Anexos

5.1. Bibliografía

Colaboradores de Wikipedia. (2023b, octubre 19). *Regla del Trapecio*. https://es.wikipedia.org/wiki/Regla_del_trapecio

Colaboradores de Wikipedia. (2023a, octubre 28). *Gaussian Integral*. https://en.wikipedia.org/wiki/Gaussian_integral

Colaboradores de Wikipedia. (2023c, noviembre 7). *Monte Carlo integration*. https://en.wikipedia.org/wiki/Monte_Carlo_integration

5.2. Código MPI: Cálculo Integral por Trapecios

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <locale.h>

// absolute substraction
double abs_subs(double a, double b) {
    return a > b ? a - b : b - a;
}

double gaussian(double x) {
    return exp(-pow(x, 2));
}

// calculates integral by using trapezoids
// and calculates trapezoids by calculating a square and adding/substracting a triangle
double trapezoids(double x, double step) {
    double y1 = gaussian(x);
    double y2 = gaussian(x + step);

    double square_area = y1 * step;
    double triangle_area = abs_subs(y1, y2) * step / 2;
    return y1 > y2 ? square_area - triangle_area : square_area + triangle_area;
}

int main(int argc, char *argv[]) {
    int node = 0, npes;
    int neg_x = -10, pos_x = 10;        // range of function to calculate (from x -10 to x 10)
    long int num_iter = argc > 1 ? atoi(argv[1]) : 100000000;      // number of iterations each node does
    double reference = 3.14159265358979323846264338327950288419716939937510582097494460;

    struct timeval t_prev, t_init, t_final;
    struct timespec sleep_time = {0,1000000};   // Sleep for 1ms
    double overhead,total_time;

    gettimeofday(&t_prev,NULL);
    gettimeofday(&t_init,NULL);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    double size = (pos_x - neg_x) / (double)npes;   // how many units each node is going to calculate
    MPI_Comm_rank(MPI_COMM_WORLD, &node);

    // ------------------------------
    //          pi calculus
    // ------------------------------

    double start_x = neg_x + size*node;     // starting position for each node
    double step = size/num_iter;            // how much will x increase each iteration
    double total = 0;
    for (int i = 0; i < num_iter; i++) {
        total += trapezoids(start_x, step);
        start_x += step;
    }

    // ------------------------------
    //         message passing
    // ------------------------------

    // basically the first n/2 nodes receive and the others send
    // when n is odd, the node in the middle does nothing
    // next iteration, only the first n/2 nodes are taken into account
    // the first half receives, the second half sends
    // etc
```

```
    double msg;
    int node_step;
    int iter = npes % 2 ? npes/2 + 1 : npes/2;  // number of iterations. if npes is odd, one more than half
    int tot_nodes = npes;                       // node total each iter
    int jump;
    int flag = 0;

    for (int i = 0; i < iter; i++) {
        msg = 0;
        node_step = flag ? tot_nodes / 2 + 1 : tot_nodes / 2;
        jump = node_step;

        if (flag) flag = 0;

        if (node_step % 2 && tot_nodes % 2) {
            jump++;
            flag = 1;
        }

        if (node < node_step) {
            MPI_Recv(&msg, 1, MPI_DOUBLE, node + jump, MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
            total += msg;
        } else if (node - jump >= 0 && node < tot_nodes) {
            nanosleep(&sleep_time,&sleep_time);
            MPI_Send(&total, 1, MPI_DOUBLE, node - jump, 0, MPI_COMM_WORLD);
        }

        tot_nodes = node_step;
    }

    gettimeofday(&t_final,NULL);
    overhead = (t_init.tv_sec-t_prev.tv_sec+(t_init.tv_usec-t_prev.tv_usec)/1.e6);
    total_time = (t_final.tv_sec-t_init.tv_sec+(t_final.tv_usec-t_init.tv_usec)/1.e6)-overhead;

    if (!node){
        double pi = total*total;
        double error = abs_subs(pi,reference);
        double quality = 1/(error*total_time);

        printf("---------------------- RESULTS -----------------------\n");
        printf("PI == %.50f\n",pi);
        printf("Difference btwn reference == %.50f\n",error);
        printf("Time == %.50f\n", total_time);
        printf("Quality == %.50f\n",quality);
        printf("------------------------------------------------------\n");

        // Create CSV file
        FILE* fp = fopen("results_trapezoids.csv", "a");
        if (fp == NULL) {
            printf("Error opening file results_trapezoids.csv\n");
            exit(EXIT_FAILURE);
        }
        setlocale(LC_ALL, "es_ES.utf8");
        if (argc > 1)
            fprintf(fp,"%d;%.50f;%.50f;%.50f;%ld\n", npes, total_time, error, quality, num_iter);
        else
            fprintf(fp,"%d;%.50f;%.50f;%.50f\n", npes, total_time, error, quality);
        // Closing CSV file
        fclose(fp);

    }
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

5.3. Código MPI: Cálculo Integral por Monte Carlo

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>
#include <locale.h>

// absolute substraction
double abs_subs(double a, double b) {
    return a > b ? a - b : b - a;
}

double gaussian(double x) {
    return exp(-pow(x, 2));
}

// calculates integral using monte carlo
long int montecarlo(long int num_iter, int x_range, int y_range) {
    double x, y;
    long int total = 0;

    for (int i = 0; i < num_iter; i++) {
        x = (rand() % (x_range * 10000) - x_range*5000)/(double)10000;
        y = (rand() % (y_range * 10000))/(double)10000;

        if (y < gaussian(x)) total++;
    }
    return total;
}

int main(int argc, char *argv[]) {
    int node = 0, npes;
    int x_range = 10, y_range = 10; // x is [-5,5]; y is [0,10]
    long int num_iter = argc > 1 ? atoi(argv[1]) : 100000000;      // number of iterations each node does
    double reference = 3.14159265358979323846264338327950288419716939937510582097494446;

    struct timeval t_prev, t_init, t_final;
    struct timespec sleep_time = {0,1000000};   // Sleep for 1ms
    double overhead,total_time;

    gettimeofday(&t_prev,NULL);
    gettimeofday(&t_init,NULL);

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &npes);
    MPI_Comm_rank(MPI_COMM_WORLD, &node);

    // ------------------------------
    //          pi calculus
    // ------------------------------

    srand(time(NULL)/(node+1));
    long int total = montecarlo(num_iter, x_range, y_range);

    // ------------------------------
    //         message passing
    // ------------------------------

    // basically the first n/2 nodes receive and the others send
    // when n is odd, the node in the middle does nothing
    // next iteration, only the first n/2 nodes are taken into account
    // the first half receives, the second half sends
    // etc

    long int msg;
    int node_step;
```

```
    int iter = npes % 2 ? npes/2 + 1 : npes/2;  // number of iterations. if npes is odd, one more than half
    int tot_nodes = npes;                       // node total each iter
    int jump;
    int flag = 0;

    for (int i = 0; i < iter; i++) {
        msg = 0;
        node_step = flag ? tot_nodes / 2 + 1 : tot_nodes / 2;
        jump = node_step;

        if (flag) flag = 0;

        if (node_step % 2 && tot_nodes % 2) {
            jump++;
            flag = 1;
        }

        if (node < node_step) {
            MPI_Recv(&msg, 1, MPI_LONG, node + jump, MPI_ANY_TAG, MPI_COMM_WORLD, NULL);
            total += msg;
        } else if (node - jump >= 0 && node < tot_nodes) {
            nanosleep(&sleep_time,&sleep_time);
            MPI_Send(&total, 1, MPI_LONG, node - jump, 0, MPI_COMM_WORLD);
        }

        tot_nodes = node_step;
    }

    gettimeofday(&t_final,NULL);
    overhead = (t_init.tv_sec-t_prev.tv_sec+(t_init.tv_usec-t_prev.tv_usec)/1.e6);
    total_time = (t_final.tv_sec-t_init.tv_sec+(t_final.tv_usec-t_init.tv_usec)/1.e6)-overhead;

    if (!node){
        double root_pi = total / (double)(num_iter * npes) * 100;
        double pi = root_pi * root_pi;
        double error = abs_subs(pi,reference);
        double quality = 1/(error*total_time);

        printf("---------------------- RESULTS -----------------------\n");
        printf("PI == %.50f\n",pi);
        printf("Difference btwn reference == %.50f\n",error);
        printf("Time == %.50f\n", total_time);
        printf("Quality == %.50f\n",quality);
        printf("------------------------------------------------------\n");

        // Create CSV file
        FILE* fp = fopen("results_montecarlo.csv", "a");
        if (fp == NULL) {
            printf("Error opening file results.csv\n");
            exit(EXIT_FAILURE);
        }
        setlocale(LC_ALL, "es_ES.utf8");
        if (argc > 1)
            fprintf(fp,"%d;%.50f;%.50f;%.50f;%ld\n", npes, total_time, error, quality, num_iter);
        else
            fprintf(fp,"%d;%.50f;%.50f;%.50f\n", npes, total_time, error, quality);
        // Closing CSV file
        fclose(fp);

    }
    MPI_Finalize();

    return EXIT_SUCCESS;
}
```

5.4. Código Bash: Execución dos códigos

5.4.1. Aumentando o número de nodos

```
#!/bin/bash

#SBATCH -J ex1
#SBATCH -o ex1_%j.out
#SBATCH -e ex1_%j.err
#SBATCH -N 32
#SBATCH -n 256
#SBATCH -t 01:00:00
#SBATCH --mem=4G

module load cesga/2020 gcc openmpi/4.1.1_ft3

for nN in {1,2,4,8,16,32}; do
  for np in {1,2,4,8,16,32}; do
    echo $np $nN
    if [ $np -ge $nN ]; then
      echo "Executing..."
      for i in {1..5}; do
        srun -N $nN -n $np $1
      done
    fi
  done
done

echo "done"
```

5.4.2. Aumentando o número de iteracións

```
#!/bin/bash

#SBATCH -J ex1
#SBATCH -o ex1_%j.out
#SBATCH -e ex1_%j.err
#SBATCH -N 4
#SBATCH -n 16
#SBATCH -t 02:30:00
#SBATCH --mem=4G

module load cesga/2020 gcc openmpi/4.1.1_ft3

for ((i = 1; i <= 1000000000; i *= 10)); do
    for j in {1..10}; do
        srun -N 4 -n 16 $1 $i
    done
done

echo "done"
```

5.5. Código Python: Unificar CSVs

```
import csv
import argparse
from pathlib import Path

# reads csv, generates another with better formating and number of nodes
def csv_rdr_wrtr(input, output):
    with open(input, newline='') as incsv, open(output, 'w', newline='') as outcsv:
        rdr = csv.reader(incsv, delimiter=';')
        wrtr = csv.writer(outcsv, delimiter=',')
        prev_cpus = 9999
        nodes = 0
        for row in rdr:
            if int(row[0]) < prev_cpus:
                nodes = row[0]
            prev_cpus = int(row[0])
            to_write = [nodes, row[0], row[1].replace(',','.'),
                        row[2].replace(',','.'), row[3].replace(',','.').strip()]
            if row[4]: to_write.append(row[4])
            wrtr.writerow(to_write)

# merges various csvs into one
def merge_csv(inputs, output):
    data = []
    for inp in inputs:
        with open(inp, newline='') as incsv:
            rdr = csv.reader(incsv, delimiter=',')
            for row in rdr:
                data.append({
                    'nodes': int(row[0]),
                    'cpus': int(row[1]),
                    'time': row[2],
                    'err': row[3],
                    'qual': row[4]
                    })
                if row[5]: data[-1]['iter'] = row[5]
        Path.unlink(inp)
    ordered = sorted(data, key=lambda d: (d['nodes'], d['cpus']))
    with open(output, 'w', newline='') as outcsv:
        wrtr = csv.writer(outcsv, delimiter=',')
        for item in ordered:
            wrtr.writerow(item.values())


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('input_files', help='input files, separated by commas')
    parser.add_argument('-o', '--output', help='output file, defaults to ./output.csv')
    args = parser.parse_args()
    input = args.input_files.split(',')
    if args.output:
        output = args.output
    else:
        output = './output.csv'
    temp_out = []
    print(input, output)
    i = 0
    for inp in input:
        temp_out.append('./temp_'+str(i)+'.csv')
        csv_rdr_wrtr(inp, temp_out[i])
        i+=1
    merge_csv(temp_out, output)

if __name__ == "__main__":
    main()
```