Relatório do Projeto de Sistemas Operativos

LEI

Sistemas Operativos

Discentes: Anees Asghar, Diogo Pinto, João Nunes, Pedro Mendes

Docente: Gonçalo Valadão

Ano Letivo: 2021/2022

10 de junho de 2022, Lisboa.

Índice

Índice de Imagens

Índice de Tabelas

Introdução

No âmbito da cadeira de Sistemas Operativos, foi-nos proposto a realização de um projeto em *C*. O mesmo consiste na realização de um algoritmo de programação, conhecido como simulação de *Monte Carlo*, a qual é uma técnica frequentemente utilizada para aproximar uma função ao utilizar escolhas aleatórias na realização dos cálculos.

Neste exemplo específico, *Monte Carlo*, foi utilizado para gerar *n* pontos aleatórios dentro de um quadrado de lado 2 centrado na origem do plano cartesiano, que contem um círculo de raio 1 nele inscrito. De seguida, procedemos aos cálculos necessários para obter o valor aproximado de *Pi*, para isso procedemos a contagem de quantos pontos se encontravam dentro do círculo e, por fim, dividimos essa contagem pelo total de pontos gerados vezes 4.

Para a resolução desse desafio, foi necessário utilizar os conhecimentos adquiridos em contexto de sala de aula, sobre *C*, noções de matemática e, por fim, os conhecimentos sobre alocação de memória e Threads, sendo que o mesmo foi essencial na otimização do código.

Teoria

Assim como foi referido anteriormente, passaremos à explicação teórica de todos os processos necessários para o desenvolvimento deste projeto. Primeiramente ir-se-á enunciar a metodologia que engloba a programação em *C*. De seguida, entraremos em detalhe sobre o que consiste a simulação de *Monte Carlo* e toda a teoria envolvente. Finalmente, passaremos à explicitação do valor π.

Em C, foi bastante importante perceber como funciona a memória e a alocação da mesma. Para isso, recorremos ao livro "Advance Linux Programing" (Mark Mitchell, 2001). Recorremos também ao mesmo livro para perceber o que são Threads e como funcionam.

Por definição, *Threads* são um mecanismo que permite ao programa realizar múltiplas tarefas de uma só vez. Os *Threads* são executados de forma assíncrona, sendo que o *Kernel Linux* interrompe cada *Thread* para permitir a criação de outros. (Mark Mitchell, 2001)

Um *Thread* existe dentro de um processo, ou seja, quando um programa é iniciado, existe a criação de um processo, onde, por sua vez, há a criação de um único *Thread* que executa o programa sequencialmente. Um *Thread* pode criar mais *Threads* dentro do mesmo processo, sendo que todos executam o mesmo programa, mas podem executar partes diferentes do mesmo programa em qualquer instante de tempo (Mark Mitchell, 2001).

Monte Carlo Simulation é uma metodologia muito utilizada para modelar probabilidades de diferentes resultados, que por si só, não seriam tão facilmente previsíveis devido ao facto de as variáveis em estudo serem *random*. (Family, s.d.)

Pi é o nome dado ao valor do rácio entre perímetro e diâmetro de um qualquer círculo. O que significa que para qualquer círculo, é possível dividir o seu perímetro (distancia a volta do círculo) pelo seu diâmetro e obter sempre o mesmo valor, sendo esse valor um número irracional, o que quer dizer que é um número decimal infinito, e que as suas casas decimais não formam nenhum padrão. O símbolo utilizado para *Pi* é "π". (Math.com, 2005)

Apresentação dos resultados

O presente capítulo destina-se à apresentação dos resultados obtidos. Primeiramente, iremos expor pequenos excertos de código e a explicação do mesmo, de seguida vamos apresentar as tabelas resultantes de um estudo estatístico.

Por questões de organização, decidimos dividir o programa em duas funções principais, visíveis abaixo nas imagens 1 e 2, sendo respetivamente a função Monte Carlo, que devolve o valor aproximado de *Pi* para cada cálculo (imagem 1) e a função *main*, que como o nome indica, é a função principal do programa na qual são indicados os números de pontos e números *Threads* a serem utilizados para cada cálculo (imagem 2).

```c
float monte_carlo(int n_points, int n_threads)
{
    double pi;
    int i, n_points_generated, total_n_points_in_circle = 0;
    int n_points_per_thread = ceil((double)n_points / n_threads); // calculate the number of random points each thread has to generate

    pthread_t *thread_ids = malloc(n_threads * sizeof(pthread_t)); // array of size n_threads to store pthread_t
    int **result = malloc(n_threads * sizeof(int *));             // array of size n_threads to store the return values of the threads

    // create threads
    for (i = 0; i < n_threads; i++)
    {
        pthread_create(thread_ids + i, NULL, &routine, (void *)&n_points_per_thread);
    }

    // await threads
    for (i = 0; i < n_threads; i++)
    {
        pthread_join(thread_ids[i], (void **)(result + i));
    }

    // calculate total number of points in circle (across all threads)
    for (i = 0; i < n_threads; i++)
    {
        total_n_points_in_circle += *result[i];
    }

    // compute pi
    n_points_generated = n_points_per_thread * n_threads; // total number of points generated across all threads
    pi = ((double)total_n_points_in_circle / n_points_generated) * 4;

    // free dynamically allocated memory
    free(thread_ids);
    for (i = 0; i < n_threads; i++)
    {
        free(result[i]); // allocated inside thread routine
    }
    free(result);

    return pi;
}
```

Imagem 1 – Função Monte Carlo

```c
int main()
{
    srand(time(NULL));
    int i, j, n_points, n_threads;
    double approx_pi, rel_error, os_time_spent;
    clock_t os_start, os_end;
    struct timeval wc_start, wc_end;
    long wc_seconds, wc_microseconds;
    int thread_options[] = {2, 4, 6, 8};
    int point_options[] = {20000, 100000, 1000000, 10000000};
    // simulate monte carlo and print performance metrics for each set of options
    for (i = 0; i < 4; i++)
    {
        n_points = point_options[i];
        for (j = 0; j < 4; j++)
        {
            os_time_spent = 0.0;
            n_threads = thread_options[j];
            // run and time monte carlo with this set of settings
            os_start = clock();
            gettimeofday(&wc_start, NULL);
            approx_pi = monte_carlo(n_points, n_threads);
            os_end = clock();
            gettimeofday(&wc_end, NULL);
            os_time_spent += (double)(os_end - os_start) / CLOCKS_PER_SEC;
            wc_seconds = (wc_end.tv_sec - wc_start.tv_sec);
            wc_microseconds = ((wc_seconds * 1000000) + wc_end.tv_usec) - (wc_start.tv_usec);
            rel_error = fabs(M_PI - approx_pi) / M_PI; // compute relative error
            // performance metrics
            printf("Number of points: %d\n", n_points);
            printf("Number of threads: %d\n", n_threads);
            printf("Approximation of pi: %.16lf\n", approx_pi);
            printf("Relative Error: %.16lf\n", rel_error);
            printf("Elapsed time (Operating System): %lf seconds\n", os_time_spent);
            printf("Elapsed time (Wall Clock Time): %ld s and %ld ms\n\n", wc_seconds, wc_microseconds);
        }
    }
    return 0;
}
```

Imagem 2- Função Main

Foi também criado uma função utilitária, que dado as coordenadas (x, y) de um ponto, devolve 1 caso que se encontre dentro do círculo, visível abaixo na imagem 3.

```c
// Returns 1 if point (x, y) is on or inside the circle of radius 1, otherwise returns 0
int point_in_circle(float x, float y)
{
    float distance_from_origin = sqrt(x * x + y * y);
    return (distance_from_origin <= 1.0) ? 1 : 0;
}
```

Imagem 3 – Função Ponto dentro do Círculo (point in circle)

Por fim foi criada uma função ao qual se deu o nome de *routine*, que tem como objetivo correr as instruções que cada *Thread* tem de executar, visível na imagem 4.

```c
// Thread routine to generate a specified number of points and return a pointer to the
// total number of those points that fall on or inside the circle of radius 1
void *routine(void *param)
{
    int *p_n_points = (int *)param; // pointer to the number of points to generate
    int i = 0, n_points_in_circle = 0;
    int *p_result = malloc(sizeof(int *)); // allocate space in memory for the result (n_points_in_circle)

    // generate points and increment n_points_in_circle if point falls on or inside the circle
    for (i = 0; i < *p_n_points; i++)
    {
        float x = rand_float(-1.0, 1.0), y = rand_float(-1.0, 1.0);
        if (point_in_circle(x, y))
        {
            n_points_in_circle++;
        }
    }

    *p_result = n_points_in_circle;
    return (void *)p_result;
}
```

Imagem 4- Função Routine

Para finalizar o vigente capítulo, iremos apresentar os resultados obtidos ao executar o programa.

- Para 20000 pontos:

```
Number of points: 20000
Number of threads: 2
Approximation of pi: 3.1498000621795654
Relative Error: 0.0026124992940743
Time elapsed (Wall-Clock Time): 0.005452s

Number of points: 20000
Number of threads: 4
Approximation of pi: 3.1656000614166260
Relative Error: 0.0076417952529270
Time elapsed (Wall-Clock Time): 0.003721s

Number of points: 20000
Number of threads: 6
Approximation of pi: 3.1561686992645264
Relative Error: 0.0046396994397341
Time elapsed (Wall-Clock Time): 0.004319s

Number of points: 20000
Number of threads: 8
Approximation of pi: 3.1447999477386475
Relative Error: 0.0010209134354798
Time elapsed (Wall-Clock Time): 0.003523s
```

Imagem 5 – Resultados obtidos para 20000 pontos

- Para 100000 pontos os resultados obtidos foram:

```
Number of points: 100000
Number of threads: 2
Approximation of pi: 3.1458001136779785
Relative Error: 0.0013392761417931
Time elapsed (Wall-Clock Time): 0.019703s

Number of points: 100000
Number of threads: 4
Approximation of pi: 3.1419599056243896
Relative Error: 0.0001168999533332
Time elapsed (Wall-Clock Time): 0.014736s

Number of points: 100000
Number of threads: 6
Approximation of pi: 3.1390571594238281
Relative Error: 0.0008070728593880
Time elapsed (Wall-Clock Time): 0.015721s

Number of points: 100000
Number of threads: 8
Approximation of pi: 3.1471199989318848
Relative Error: 0.0017594086667397
Time elapsed (Wall-Clock Time): 0.013946s
```

Imagem 6- Resultados obtidos com 100000 pontos

- Para 1000000 pontos:

```
Number of points: 1000000
Number of threads: 2
Approximation of pi: 3.1421360969543457
Relative Error: 0.0001729833955181
Time elapsed (Wall-Clock Time): 0.218066s

Number of points: 1000000
Number of threads: 4
Approximation of pi: 3.1405680179595947
Relative Error: 0.0003261516508283
Time elapsed (Wall-Clock Time): 0.144939s

Number of points: 1000000
Number of threads: 6
Approximation of pi: 3.1422297954559326
Relative Error: 0.0002028085548938
Time elapsed (Wall-Clock Time): 0.179535s

Number of points: 1000000
Number of threads: 8
Approximation of pi: 3.1423959732055664
Relative Error: 0.0002557045754660
Time elapsed (Wall-Clock Time): 0.197566s
```

Imagem 7- Resultados obtidos com 1000000 pontos

- Para 10000000 pontos:

```
Number of points: 10000000
Number of threads: 2
Approximation of pi: 3.1412992477416992
Relative Error: 0.0000933939821124
Time elapsed (Wall-Clock Time): 1.992247s

Number of points: 10000000
Number of threads: 4
Approximation of pi: 3.1417667865753174
Relative Error: 0.0000554282508031
Time elapsed (Wall-Clock Time): 1.287547s

Number of points: 10000000
Number of threads: 6
Approximation of pi: 3.1419200897216797
Relative Error: 0.0001042261578733
Time elapsed (Wall-Clock Time): 1.821982s

Number of points: 10000000
Number of threads: 8
Approximation of pi: 3.1418275833129883
Relative Error: 0.0000747804534514
Time elapsed (Wall-Clock Time): 2.041938s
```

Imagem 8 - Resultados obtidos com 10000000 pontos

Apreciação crítica dos resultados

Neste último capítulo, passaremos à discussão dos resultados obtidos e quais os seus significados. Inicializaremos pelo estudo dos valores obtidos, visíveis nas imagens 5 a 8, e comparar com os resultados expectáveis para cada etapa.

Começando pela análise do valor aproximado de *Pi*, observarmos que, com o aumento da quantidade de *Threads*, o valor de pi aproximado não diferencia significantemente, isto é, com 2,4,6,8 *Threads* todos os valores obtidos são muito próximos de si. Onde se observou maior diferenças nos resultados obtidos foi com o aumento de números de pontos, onde observamos que a qualidade de valores de *Pi* era melhor.

Em contrapartida obtivemos um resultado diferente do espectável relativo ao tempo que o programa demora por *Thread.* Pois apesar de distribuirmos trabalho por um maior número de *Threads*, o tempo por cada não diminui, sendo até visível que em certos casos aumenta.

O resultado obtido poderá estar correlacionado com o facto de ter sido utilizado uma máquina virtual, pois também verificamos, que para cada tipo de processador os resultados obtidos eram diferentes entre si. Para maior certeza nestas conclusões decidimos correr o programa 11 vezes e analisar os dados.

Na análise de dados, começamos por dividir os valores por categorias sendo elas as seguintes: 1- 20000; 2- 100000; 3- 1000000 e 4-10000000. A seguir passamos ao cálculo do desvio padrão do valor de pi aproximado e por fim o desvio padrão do erro relativo por número de pontos, todos estes resultados são visíveis nas tabelas 1 a 4, respetivamente.

Tabela 1- Desvio Padrão para 20000 pontos

Tabela 2- Desvio Padrão para 100000 pontos

Statistics

		erro relat.	pi aprox.

Statistics

		pi aprox.	erro relat.
N	Valid	44	44
	Missing	0	0
Mean		3.1422024803	.00139775072
Median		3.1438400745	.00134726493
Std. Deviation		.00506553594	.00079955043

Tabela 3- Desvio Padrão para 1000000 pontos

Statistics

		pi aprox.	erro relat.
N	Valid	44	44
	Missing	0	0
Mean		3.1413755146	.00041800896
Median		3.1414600611	.00033130212
Std. Deviation		.00161510323	.00030065851

Tabela 4- Desvio Padrão para 10000000 pontos

Statistics

		pi aprox.	erro relat.
N	Valid	44	44
	Missing	0	0
Mean		3.1416045915	.00012694516
Median		3.1416994333	.00010236683
Std. Deviation		.00052598229	.00010749970

Com estas tabelas conseguimos verificar que, independente da quantidade de pontos todos os valores obtidos são muito próximos de si, ou seja, não há um desvio padrão muito elevado, como por exemplo, o desvio padrão para 10000000 e para o valor de pi aproximado é de 0.00052598229.

Verificamos ainda que à medida que a quantidade de pontos aumenta o erro relativo médio apresenta um menor valor, ou seja, a qualidade dos valores obtidos de *Pi* vai aumentando consoante o aumento de pontos.

Por fim, podemos também, concluir que, apesar do programa criar pontos de forma *random*, obtemos alguns valores iguais, visíveis nas tabelas 5 a 12.

Tabela 5- Resultados do erro relativo para 20000 pontos

erro relat.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	.0000703231209159	1	2.3	2.3	2.3
	.0002569947223092	1	2.3	2.3	4.5
	.0003159822580641	1	2.3	2.3	6.8
	.0003206672635719	1	2.3	2.3	9.1
	.0003843398048346	1	2.3	2.3	11.4
	.0005069239908613	1	2.3	2.3	13.6
	.0005752815376318	1	2.3	2.3	15.9
	.0009480580843330	1	2.3	2.3	18.2
	.0010117306255956	1	2.3	2.3	20.5
	.0012119310592678	1	2.3	2.3	22.7
	.0012663448996556	1	2.3	2.3	25.0
	.0015255328690825	1	2.3	2.3	27.3
	.0016575629571157	3	6.8	6.8	34.1
	.0017754975567846	1	2.3	2.3	36.4
	.0018391700980473	1	2.3	2.3	38.6
	.0019121772311757	1	2.3	2.3	40.9
	.0020348373081932	1	2.3	2.3	43.2
	.0021621823907185	1	2.3	2.3	45.5
	.0021668673962263	1	2.3	2.3	47.7
	.0022258549319812	1	2.3	2.3	50.0
	.0022941365877609	1	2.3	2.3	52.3
	.0023531241235158	2	4.5	4.5	56.8
	.0026124992940743	1	2.3	2.3	59.1
	.0027351593710918	2	4.5	4.5	63.6
	.0031218037331849	1	2.3	2.3	65.9
	.0032443879192116	1	2.3	2.3	68.2
	.0034400905485075	1	2.3	2.3	70.5
	.0035037630897702	1	2.3	2.3	72.7
	.0037486633170105	2	4.5	4.5	77.3
	.0037536923583223	1	2.3	2.3	79.5
	.0040130675288808	1	2.3	2.3	81.8
	.0040165787503770	1	2.3	2.3	84.1
	.0042578159741396	1	2.3	2.3	86.4
	.0050316005161113	1	2.3	2.3	88.6
	.0052862906811619	1	2.3	2.3	90.9
	.0053452782169168	1	2.3	2.3	93.2
	.0059865368530698	2	4.5	4.5	97.7
	.0068807806333264	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 6- Resultados do valor de pi aproximado para 20000 pontos

pi aprox.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	3.1199760437011720	1	2.3	2.3	2.3
	3.1247999668121340	1	2.3	2.3	4.5
	3.1289741992950440	1	2.3	2.3	6.8
	3.1298000812530520	1	2.3	2.3	9.1
	3.1314001083374023	1	2.3	2.3	11.4
	3.1329998970031734	2	4.5	4.5	15.9
	3.1342000961303710	2	4.5	4.5	20.5
	3.1345999240875244	1	2.3	2.3	22.7
	3.1347999572753900	1	2.3	2.3	25.0
	3.1352000236511230	1	2.3	2.3	27.3
	3.1368000507354736	1	2.3	2.3	29.5
	3.1400001049041750	1	2.3	2.3	31.8
	3.1405999660491943	1	2.3	2.3	34.1
	3.1413717269897460	1	2.3	2.3	36.4
	3.1424000263214110	1	2.3	2.3	38.6
	3.1426000595092773	1	2.3	2.3	40.9
	3.1428000926971436	1	2.3	2.3	43.2
	3.1433999538421630	1	2.3	2.3	45.5
	3.1445710659027100	1	2.3	2.3	47.7
	3.1447710990905760	1	2.3	2.3	50.0
	3.1454000473022460	1	2.3	2.3	52.3
	3.1455709934234620	1	2.3	2.3	54.5
	3.1468000411987305	3	6.8	6.8	61.4
	3.1471705436706543	1	2.3	2.3	63.6
	3.1473705768585205	1	2.3	2.3	65.9
	3.1475999355316160	1	2.3	2.3	68.2
	3.1484000682830810	1	2.3	2.3	70.5
	3.1487998962402344	1	2.3	2.3	72.7
	3.1498000621795654	1	2.3	2.3	75.0
	3.1514000892639160	1	2.3	2.3	77.3
	3.1524000167846680	1	2.3	2.3	79.5
	3.1526000499725340	1	2.3	2.3	81.8
	3.1533694267272950	2	4.5	4.5	86.4
	3.1542000770568848	1	2.3	2.3	88.6
	3.1549689769744873	1	2.3	2.3	90.9
	3.1573998928070070	1	2.3	2.3	93.2
	3.1582000255584717	1	2.3	2.3	95.5
	3.1603999137878420	2	4.5	4.5	100.0
	Total	44	100.0	100.0	

Tabela 7- Resultados do valor de pi aproximado para 100000 pontos

pi aprox.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	3.1315200328826904	1	2.3	2.3	2.3
	3.1321198940277100	1	2.3	2.3	4.5
	3.1333999633789062	1	2.3	2.3	6.8
	3.1340973377227783	1	2.3	2.3	9.1
	3.1345372200012207	1	2.3	2.3	11.4
	3.1361372470855713	2	4.5	4.5	15.9
	3.1373600959777830	2	4.5	4.5	20.5
	3.1375372409820557	1	2.3	2.3	22.7
	3.1379599571228027	1	2.3	2.3	25.0
	3.1391999721527100	1	2.3	2.3	27.3
	3.1392400264739990	1	2.3	2.3	29.5
	3.1392800807952880	1	2.3	2.3	31.8
	3.1394400596618652	1	2.3	2.3	34.1
	3.1399600505828857	1	2.3	2.3	36.4
	3.1400799751281734	1	2.3	2.3	38.6
	3.1403772830963135	1	2.3	2.3	40.9
	3.1414399147033690	1	2.3	2.3	43.2
	3.1414799690246580	1	2.3	2.3	45.5
	3.1425600051879883	1	2.3	2.3	47.7
	3.1438400745391850	2	4.5	4.5	52.3
	3.1440799236297607	1	2.3	2.3	54.5
	3.1441199779510500	2	4.5	4.5	59.1
	3.1442000865936280	1	2.3	2.3	61.4
	3.1442399024963380	1	2.3	2.3	63.6
	3.1455371379852295	1	2.3	2.3	65.9
	3.1455600261688232	1	2.3	2.3	68.2
	3.1459999084472656	1	2.3	2.3	70.5
	3.1460170745849610	1	2.3	2.3	72.7
	3.1462571620941160	1	2.3	2.3	75.0
	3.1465370655059814	1	2.3	2.3	77.3
	3.1467199325561523	1	2.3	2.3	79.5
	3.1470398902893060	1	2.3	2.3	81.8
	3.1473600864410400	2	4.5	4.5	86.4
	3.1476800441741943	1	2.3	2.3	88.6
	3.1477999687194824	1	2.3	2.3	90.9
	3.1484799385070800	1	2.3	2.3	93.2
	3.1486170291900635	1	2.3	2.3	95.5
	3.1492800712585450	1	2.3	2.3	97.7
	3.1510000228881836	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 8- Resultados do erro relativo para 100000 pontos

erro relat.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	.0000358686111028	1	2.3	2.3	2.3
	.0000486182975535	1	2.3	2.3	4.5
	.0003079175771212	1	2.3	2.3	6.8
	.0003868644434506	1	2.3	2.3	9.1
	.0004815005089507	1	2.3	2.3	11.4
	.0005196736773120	1	2.3	2.3	13.6
	.0006851919281986	1	2.3	2.3	15.9
	.0007153763066079	2	4.5	4.5	20.5
	.0007361147830106	1	2.3	2.3	22.7
	.0007488644694613	1	2.3	2.3	25.0
	.0007616141559120	1	2.3	2.3	27.3
	.0007917226433304	1	2.3	2.3	29.5
	.0008044723297811	2	4.5	4.5	34.1
	.0008299717026825	1	2.3	2.3	36.4
	.0008426454981424	1	2.3	2.3	38.6
	.0011563231989480	1	2.3	2.3	40.9
	.0012555683789651	1	2.3	2.3	43.2
	.0012628539140798	1	2.3	2.3	45.5
	.0012908779255972	1	2.3	2.3	47.7
	.0013472649317452	2	4.5	4.5	52.3
	.0014028727920650	1	2.3	2.3	54.5
	.0014083369434010	1	2.3	2.3	56.8
	.0014847591711144	1	2.3	2.3	59.1
	.0015738551942877	1	2.3	2.3	61.4
	.0016320635842143	1	2.3	2.3	63.6
	.0017339092938383	1	2.3	2.3	65.9
	.0017365098234452	2	4.5	4.5	70.5
	.0018358308944531	2	4.5	4.5	75.0
	.0019376766040770	1	2.3	2.3	77.3
	.0019758497724383	1	2.3	2.3	79.5
	.0021922908781370	1	2.3	2.3	81.8
	.0022359281978343	1	2.3	2.3	84.1
	.0022458142625558	1	2.3	2.3	86.4
	.0023858331405410	1	2.3	2.3	88.6
	.0024469810431876	1	2.3	2.3	90.9
	.0026078142885665	1	2.3	2.3	93.2
	.0029944586506596	1	2.3	2.3	95.5
	.0030152730180531	1	2.3	2.3	97.7
	.0032062147508503	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 9- Resultados do valor aproximado de pi para 1000000 pontos

pi aprox.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	3.1381158828735350	1	2.3	2.3	2.3
	3.1385858058929443	1	2.3	2.3	4.5
	3.1390440464019775	1	2.3	2.3	6.8
	3.1390480995178223	2	4.5	4.5	11.4
	3.1393599510192870	1	2.3	2.3	13.6
	3.1393640041351320	1	2.3	2.3	15.9
	3.1393778324127197	1	2.3	2.3	18.2
	3.1396040916442870	1	2.3	2.3	20.5
	3.1397640705108643	1	2.3	2.3	22.7
	3.1399199962615967	1	2.3	2.3	25.0
	3.1403079032897950	1	2.3	2.3	27.3
	3.1404016017913814	1	2.3	2.3	29.5
	3.1405479907989500	1	2.3	2.3	31.8
	3.1408159732818604	1	2.3	2.3	34.1
	3.1409456729888916	2	4.5	4.5	38.6
	3.1410319805145264	1	2.3	2.3	40.9
	3.1412799358367920	1	2.3	2.3	43.2
	3.1412999629974365	1	2.3	2.3	45.5
	3.1413559913635254	1	2.3	2.3	47.7
	3.1414000988006590	1	2.3	2.3	50.0
	3.1415200233459473	1	2.3	2.3	52.3
	3.1415319442749023	2	4.5	4.5	56.8
	3.1417520046234130	1	2.3	2.3	59.1
	3.1419336795806885	1	2.3	2.3	61.4
	3.1420600414276123	1	2.3	2.3	63.6
	3.1421039104461670	1	2.3	2.3	65.9
	3.1421816349029540	1	2.3	2.3	68.2
	3.1422696113586426	1	2.3	2.3	70.5
	3.1423616409301760	1	2.3	2.3	72.7
	3.1424920558929443	1	2.3	2.3	75.0
	3.1425759792327880	1	2.3	2.3	77.3
	3.1426200866699220	1	2.3	2.3	79.5
	3.1426296234130860	1	2.3	2.3	81.8
	3.1429240703582764	1	2.3	2.3	84.1
	3.1430120468139650	1	2.3	2.3	86.4
	3.1434016227722170	1	2.3	2.3	88.6
	3.1436800956726074	1	2.3	2.3	90.9
	3.1439640522003174	1	2.3	2.3	93.2
	3.1439960002899170	2	4.5	4.5	97.7
	3.1444199085235596	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 10- Resultados do erro relativo para 1000000 pontos

erro relat.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	.0000193243751132	2	4.5	4.5	4.5
	.0000231189246521	1	2.3	2.3	6.8
	.0000507230093748	1	2.3	2.3	9.1
	.0000612920930134	1	2.3	2.3	11.4
	.0000753319263073	1	2.3	2.3	13.6
	.0000931663091401	1	2.3	2.3	15.9
	.0000995411523654	1	2.3	2.3	18.2
	.0001085519443476	1	2.3	2.3	20.5
	.0001487741694599	1	2.3	2.3	22.7
	.0001627381117631	1	2.3	2.3	25.0
	.0001784677827745	1	2.3	2.3	27.3
	.0001874785747567	1	2.3	2.3	29.5
	.0002059403214361	2	4.5	4.5	34.1
	.0002154823503537	1	2.3	2.3	36.4
	.0002447762727940	1	2.3	2.3	38.6
	.0002472250204193	1	2.3	2.3	40.9
	.0002862886447495	1	2.3	2.3	43.2
	.0003130022735033	1	2.3	2.3	45.5
	.0003270421067972	1	2.3	2.3	47.7
	.0003300777464284	1	2.3	2.3	50.0
	.0003325264940537	1	2.3	2.3	52.3
	.0003791235623913	1	2.3	2.3	54.5
	.0004089487217670	1	2.3	2.3	56.8
	.0004238031200391	1	2.3	2.3	59.1
	.0004518068956361	1	2.3	2.3	61.4
	.0005324233637627	1	2.3	2.3	63.6
	.0005758127745673	1	2.3	2.3	65.9
	.0005820560717315	1	2.3	2.3	68.2
	.0006329789265434	1	2.3	2.3	70.5
	.0006644534517959	1	2.3	2.3	72.7
	.0007049994767917	1	2.3	2.3	75.0
	.0007094011542568	1	2.3	2.3	77.3
	.0007106913011000	1	2.3	2.3	79.5
	.0007548396218124	1	2.3	2.3	81.8
	.0007650090145766	2	4.5	4.5	86.4
	.0008099567170375	2	4.5	4.5	90.9
	.0008112468638808	1	2.3	2.3	93.2
	.0008999431961798	1	2.3	2.3	95.5
	.0009571093481559	1	2.3	2.3	97.7
	.0011066904909792	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 11- Resultados do valor aproximado de pi para 10000000 pontos

pi aprox.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	3.1402847766876220	2	4.5	4.5	4.5
	3.1405587196350098	1	2.3	2.3	6.8
	3.1407775878906250	1	2.3	2.3	9.1
	3.1409816741943360	2	4.5	4.5	13.6
	3.1410419940948486	1	2.3	2.3	15.9
	3.1410756111145020	1	2.3	2.3	18.2
	3.1411306858062744	1	2.3	2.3	20.5
	3.1411921977996826	1	2.3	2.3	22.7
	3.1412100791931152	1	2.3	2.3	25.0
	3.1414074897766113	1	2.3	2.3	27.3
	3.1414332389831543	1	2.3	2.3	29.5
	3.1414945125579834	1	2.3	2.3	31.8
	3.1415235996246340	1	2.3	2.3	34.1
	3.1416127681732178	1	2.3	2.3	36.4
	3.1416432857513428	1	2.3	2.3	38.6
	3.1416528224945070	1	2.3	2.3	40.9
	3.1416740417480470	2	4.5	4.5	45.5
	3.1416893005371100	1	2.3	2.3	47.7
	3.1416988372802734	1	2.3	2.3	50.0
	3.1417000293731690	1	2.3	2.3	52.3
	3.1417067050933840	1	2.3	2.3	54.5
	3.1417315006256104	1	2.3	2.3	56.8
	3.1417400836944580	1	2.3	2.3	59.1
	3.1417653560638428	2	4.5	4.5	63.6
	3.1418321132659910	1	2.3	2.3	65.9
	3.1418399810791016	2	4.5	4.5	70.5
	3.1418659687042236	1	2.3	2.3	72.7
	3.1418747901916504	1	2.3	2.3	75.0
	3.1419537067413330	1	2.3	2.3	77.3
	3.1419658660888670	1	2.3	2.3	79.5
	3.1419963836669920	1	2.3	2.3	81.8
	3.1420807838439940	1	2.3	2.3	84.1
	3.1421082019805910	1	2.3	2.3	86.4
	3.1421260833740230	1	2.3	2.3	88.6
	3.1421351432800293	1	2.3	2.3	90.9
	3.1422028541564940	1	2.3	2.3	93.2
	3.1422069072723390	1	2.3	2.3	95.5
	3.1422870159149170	1	2.3	2.3	97.7
	3.1428534984588623	1	2.3	2.3	100.0
	Total	44	100.0	100.0	

Tabela 12- Resultados do erro relativo para 10000000 pontos

erro relat.

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	.0000064026707605	1	2.3	2.3	2.3
	.0000161167175801	1	2.3	2.3	4.5
	.0000191523572112	1	2.3	2.3	6.8
	.0000219805597904	1	2.3	2.3	9.1
	.0000259066553905	2	4.5	4.5	13.6
	.0000307636788002	1	2.3	2.3	15.9
	.0000312392606653	1	2.3	2.3	18.2
	.0000337993184314	1	2.3	2.3	20.5
	.0000341787733853	1	2.3	2.3	22.7
	.0000363037211270	1	2.3	2.3	25.0
	.0000441963841679	1	2.3	2.3	27.3
	.0000469284598359	1	2.3	2.3	29.5
	.0000507432452952	1	2.3	2.3	31.8
	.0000549729048584	2	4.5	4.5	36.4
	.0000589394722993	1	2.3	2.3	38.6
	.0000762223822762	1	2.3	2.3	40.9
	.0000787267849719	2	4.5	4.5	45.5
	.0000869989029667	1	2.3	2.3	47.7
	.0000898068696255	1	2.3	2.3	50.0
	.0001149267875730	1	2.3	2.3	52.3
	.0001187972281026	1	2.3	2.3	54.5
	.0001217772126634	1	2.3	2.3	56.8
	.0001274690369717	1	2.3	2.3	59.1
	.0001285112749222	1	2.3	2.3	61.4
	.0001470489125924	1	2.3	2.3	63.6
	.0001553766856576	1	2.3	2.3	65.9
	.0001641041495971	1	2.3	2.3	68.2
	.0001645797314621	1	2.3	2.3	70.5
	.0001697959739054	1	2.3	2.3	72.7
	.0001726798315550	1	2.3	2.3	75.0
	.0001752803611618	1	2.3	2.3	77.3
	.0001942328729359	1	2.3	2.3	79.5
	.0001944807818286	2	4.5	4.5	84.1
	.0001955230197791	1	2.3	2.3	86.4
	.0002210223926805	1	2.3	2.3	88.6
	.0002594434699345	1	2.3	2.3	90.9
	.0003291113994686	1	2.3	2.3	93.2
	.0004013393867688	1	2.3	2.3	95.5
	.0004163101478725	2	4.5	4.5	100.0
	Total	44	100.0	100.0	

Conclusão

Com este trabalho podemos concluir que a simulação de *Monte Carlo* é mais efetiva à medida que a quantidade de pontos vai aumentando, sendo que se tivéssemos dado um maior número de pontos podíamos então obter um valor de pi mais próximo ao real.

Vimos também que as *Threads* não afetam na qualidade de valores, ou seja, com o aumento de *Threads* o resultado obtido para o valor aproximado de pi não diferencia muito entre si.

Por fim obtivemos resultados fora do espectável em relação ao tempo obtido, pois era de se esperar que, com o aumento de *Threads* o tempo fosse diminuindo, podendo isso ser causado pelo facto de ter se usado uma máquina virtual para a realização deste projeto. Levantado a questão, se utilizada um computador com *Linux* invés de uma máquina virtual será que os tempos obtidos já seriam os espectáveis.

Referências bibliográficas

Family, D. M. (s.d.). *Monte Carlo Simulation*. Obtido em 2022, de Investopedia: https://www.investopedia.com/terms/m/montecarlosimulation.asp

Mark Mitchell, J. O. (2001). *Advanced Linux Programming.* Indianapolis: New Riders.

Math.com. (2005). *Math.com*. Obtido em 2022, de PI: http://www.math.com/tables/constants/pi.htm