Algoritmos para o Cálculo do Valor de PI

Isadora Ferrão[1], **Lucélia Santos**[1]

[1]Instituto De Ciências Matemáticas e de Computação - Universidade de São Paulo (USP - ICMC)

[2]São Carlos - São Paulo – Brasil

{isadoraferrao,luceliasantos}@usp.br

Abstract. *The number Pi is an irrational number with infinite decimal places applied in scientific and industrial contexts. The mathematical models Gauss Legrende, Borwein and Monte Carlo contribute to the calculation of this number. In this work, these models were implemented in the programming language C and source code were sequentially and parallel (pthreads) to perform the calculation of Pi. Moreover was also implemented an application of the Monte Carlo algorithm from the mathematical model Black Scholes. The results of Pi and the execution time were collected and analyzed.*

Resumo. *O número Pi é um número irracional com infinitas casas decimais aplicado em contextos científico e industrial. Os modelos matemáticos Gauss Legrende, Borwein e Monte Carlo, contribuem para realizar o cálculo desse número. Neste trabalho, esses modelos foram implementados na linguagem de programação C e codificados de forma sequencial e paralela (pthreads) para realizar o cálculo de Pi. Também foi implementado uma aplicação do algoritmo Monte Carlo no modelo matemático Black Scholes. Os resultados do valor de Pi e o tempo de execução, foram coletados e analisados.*

1. Introdução

O número PI (PI = 3,1415926536...) é objeto de estudos devido as suas infinitas casas decimais e suas aplicações, como o cálculo do raio de cilindros e engrenagens, o rolar do movimento das ondas do mar, nas curvas das pistas automobilísticas, entre outras aplicações [Souza 2019, P. 2019, Ferreto 2019]. Como citado, ele é um número irracional, pois não consegue ser explanado em forma de expressão visto que ele tem infinitas casas decimais. Em termos gerais, o PI é usado para calcular a circunferência de círculos a partir do raio ou do diâmetro. Por esses motivos, calculá-lo de forma precisa é fundamental.

Este trabalho possui como objetivo principal calcular o valor de Pi através dos algoritmos de Gauss Legendre, Borwein e Monte Carlo. As três abordagens foram implementadas na linguagem de programação C [Damas 2016], sendo que cada algoritmo foi codificado na versão sequencial e paralela utilizando Pthreads [Buttlar et al. 1996]. O restante do trabalho está organizado como segue. A **Seção 2** detalha o desenvolvimento deste trabalho. **Na Seção 3** são descritos e detalhados os resultados obtidos e na **Seção 4** são apresentadas as considerações finais.

2. Desenvolvimento

A seguir serão apresentadas as etapas de desenvolvimento desse trabalho e alguns detalhes sobre os algoritmos implementados, incluindo as lógicas das abordagens, scripts desenvolvidos e ambiente de testes.

2.1. Etapas de Desenvolvimento

O trabalho é composto por quatro fases de desenvolvimento. A **primeira etapa** é constituída pelo estudo das abordagens que foram implementadas e pela divisão de tarefas entre os membros do grupo. A **segunda fase** é composta pela codificação dos algoritmos. Os códigos Gauss Legendre, Borwein e Monte Carlo foram implementados em duas versões: (1) versão sequencial e (2) versão paralela. Ao total foram desenvolvidos 8 códigos, visto que para Monte Carlo também foi desenvolvido o Modelo de Black Scholes, em ambas as versões. Todos os códigos foram implementados na linguagem C. Para as versões paralelas, foram utilizados o POSIX threads, o qual define uma api padrão para criar e manipular threads [Buttlar et al. 1996].

Na **terceira fase** foram realizados os testes e as análises dos resultados obtidos. Para automação dos testes e também para facilitar a análise dos resultados, foram desenvolvidos códigos em shell script. Para cada um dos códigos, existe um shell script. Para cada um dos códigos foi adaptado um shell script. Por exemplo, para as versões paralelas, os scripts possuem testes automatizados para testar com 2, 4, 8 e 16 threads. Além disso, após o término dos testes são obtidos automaticamente a média, desvio padrão e absoluto de todos os códigos desenvolvidos. A seguir serão explicitados dois exemplos dos scripts shells que foram desenvolvidos para o trabalho.

A Figura 1 ilustra o script de teste desenvolvido para testar o algoritmo de Borwein. Assim como no código de Borwein, os outros algoritmos também utilizam a média de 30 execuções para 1000000000 iterações, a fim de estipular uma média, desvio padrão e desvio absoluto do algoritmo. Para facilitar visualmente e organizacionalmente, o script cria automaticamente uma pasta Resultados e move os resultados para dentro desta pasta. São criados três arquivos automaticamente e alocados para dentro da pasta Resultados: (1) Saida.txt = esse arquivo contém o valor do pi; (2) Tempo.txt = esse arquivo contém o tempo final de cada uma das iterações; (3) Calculo.txt = esse arquivo contém a média, desvio padrão e desvio absoluto dos resultados.

```
#COMPILA
gcc -o programa Borwein.c -lm -lpthread

#VARIAVEIS
ITERACOES=30

if [ ! -d "Resultados" ];then
    mkdir Resultados
else
    rm -r Resultados #limpa para inserir novos resultados
    mkdir Resultados
fi

for (( i=0; i<$ITERACOES; i++ ));
do
    echo "Executando iteracao $i..."
    ./programa < entrada.txt >> Resultados/saida.txt
    #/usr/bin/time -f "%e" ./programa < entrada.txt >> Resultados/saida.txt
done

mv tempo.txt Resultados
gcc -o programa calculos.c -lm
echo "Executando Calculos..."
./programa $ITERACOES Resultados/tempo.txt

mv Calculo.txt Resultados
echo "Execucao finalizada!"

rm programa
```

Figura 1. Script código sequencial

A Figura 2 ilustra o script de teste desenvolvido para a versão paralela de Borwein. Através do script desenvolvido é possível testar a média de 30 execuções para 1000000000 iterações, a fim de estipular uma média, desvio padrão e desvio absoluto do código. Neste caso, ele faz todos os testes com 2 threads, depois com 4 threads, 8 threads e depois com 16 threads. Para cada um dos testes mencionados, são gerados arquivos para facilitar a análise dos resultados. São criados três arquivos automaticamente dentro da pasta Resultados: (1) 2-Saida.txt = esse arquivo contém o valor do Pi; (2) 2-Tempo.txt = esse arquivo contém o tempo final de cada uma das iterações; (3) Calculo.txt = esse arquivo contém a média, desvio padrão e absoluto dos resultados. Diferentemente da versão sequencial, neste caso os arquivos saída e tempo são padronizados para gerar conforme o número da thread. Por exemplo 2-Saida.txt indica os resultados do código rodando com 2 threads, 4-Saida.txt indica os resultados do código rodando com 4 threads, assim por diante.

```
#COMPILA
gcc -o borwein Borwein_paralelo.c -lm -lpthread
gcc -o calc calculos.c -lm

#VARIAVEIS
ITERACOES=30
declare -a threads=("2" "4" "8" "16")

if [ ! -d "Resultados" ];then
    mkdir Resultados
else
    rm -r Resultados #limpa para inserir novos resultados
    mkdir Resultados
fi

for thread in "${threads[@]}"
do
    for (( i=0; i<$ITERACOES; i++ ));
    do
        echo "Executando iteracao $i com $thread threads..."
        ./borwein $thread 10 < entrada.txt >> Resultados/$thread-saida.txt #./programa Num-THREADS E Num-ITERACOES
        #/usr/bin/time -f "%e" ./programa < entrada.txt >> Resultados/saida.txt
    done
    mv tempo.txt $thread-tempo.txt
    mv $thread-tempo.txt Resultados
    echo "Executando Calculos $thread threads..."
    ./calc $ITERACOES Resultados/$thread-tempo.txt
done

mv Calculo.txt Resultados
echo "Execucao finalizada!"

rm borwein
rm calc
```

Figura 2. Script código paralelo

Por fim, a quarta e última fase foi destinada para a documentação do trabalho.

2.2. Software e Hardware utilizado

As implementações dos programas foram feitos na linguagem c em uma máquina Dell Inspiron Special Edition, Intel core i7, sexta geração, memória RAM de 16 GB e 1 TB de HD + 8 GB de SSD. O ambiente de programação é composto por uma máquina virtual VirtualBox (versão 5.2.18) com o sistema operacional Ubuntu 12.04 LTS (64 bits).

No ambiente de execução dos testes foi utilizado uma workstation, a qual dispõe de um processador Xeon E5-2609 v3 com 6 cores (sem Hyper-Threading), 15 MB de cache L3 e 16 GB de memória principal. O compilador utilizado foi o gcc versão 4.8.3.

2.3. Algoritmos para cálculo do valor de Pi

A seguir serão apresentadas as três abordagens para calcular o valor de Pi implementadas neste trabalho, sendo elas, Gauss Legendre, Borwein e Monte Carlo. Posteriormente é apresentada a abordagem Black scholes.

2.3.1. Gauss Legendre

O algoritmo de Gauss Legendre é utilizado para calcular os dígitos de PI. Ele é notável por sua convergência rápida, podendo calcular 45 milhões de dígitos corretos com apenas 25 iterações [Foresee and Hagan 1997]. O algoritmo de Gauss Legendre utiliza cinco funções matemáticas, sendo que quatro delas fazem o uso de recursão. A seguir será explicitado um pouco sobre a parte lógica da abordagem.

Primeiro, atribuímos os casos básicos, tendo os valores conforme apresentado na Figura 3.

$$a_0 = 1 \qquad b_0 = \frac{1}{\sqrt{2}} \qquad t_0 = \frac{1}{4} \qquad p_0 = 1$$

Figura 3. Gauss

Em seguida, começamos a calcular as funções matemática para i iterações, conforme apresentado na Figura 4.

$$a_{n+1} = \frac{a_n + b_n}{2}$$

$$b_{n+1} = \sqrt{a_n b_n}$$

$$t_{n+1} = t_n - p_n(a_n - a_{n+1})^2$$

$$p_{n+1} = 2p_n$$

Figura 4. Fórmula de Gauss

Assim, Pi é calculado conforme apresenta a Figura 5.

$$\pi \approx \frac{(a_n + b_n)^2}{4t_n}$$

Figura 5. Fórmula final do Pi

O algoritmo tem natureza convergente de segunda ordem, o que significa que o número de dígitos corretos duplica a cada iteração do algoritmo. Nas funções matemáticas, sem o valor de índice 'n', não é possível calcular o valor de índice posterior "n+1". Outro ponto importante é que quanto maior for o índice, mais próximo de Pi o resultado será. Consequentemente, mais trabalhoso será o cálculo, pois será necessário calcular o valor dos índices anteriores.

2.3.2. Borwein

O algoritmo de Borwein foi criado pelos matemáticos canadenses Jonathan e Peter Borwein para calcular o valor de 1/Pi. Primeiro, atribuímos os casos básicos, tendo os seguintes conforme apresentado na Figura 6.

$$a_0 = 6 - 4\sqrt{2}$$
$$y_0 = \sqrt{2} - 1$$

Figura 6. Casos básicos de Borwein

Então começamos a calcular os seguintes valores para i iterações conforme apresentado na Figura 7.

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{1/4}}{1 + (1 - y_k^4)^{1/4}}$$
$$a_{k+1} = a_k(1 + y_{k+1})^4 - 2^{2k+3} y_{k+1}(1 + y_{k+1} + y_{k+1}^2)$$

Figura 7. Borwein fórmula

Deste modo temos que ak converge quadraticamente para 1/Pi, ou seja, cada iteração do algoritmo aproximadamente quadruplica o número de dígitos corretos calculados.

2.3.3. Monte Carlo

O método de Monte Carlo é uma técnica que utiliza pontos aleatórios para obtenção de parâmetro de interesse. É um método que utiliza simulações estocásticas indicados à solução de problemas probabilísticos e para calcular resultados determinísticos empregado em diversas áreas, como a matemática, física, engenharia, economia, biologia, entre outras [Azevedo 2019, P. 2019]. No estudo que segue, o cálculo do valor de Pi utiliza o método de Monte Carlo. Para isso, é utilizada a expressão que dependa de Pi como a área da circunferência onde A=pow(Pi,2). Colocando o círculo nas coordenadas x, y temos a representação conforme é ilustrado na Figura 8.

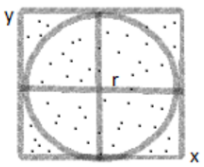

Figura 8. Monte Carlo

São utilizados pontos aleatórios para obtenção de parâmetros de interesse. Quanto maior a área de uma região, maior o número de pontos que podem ser dispostos naquela região. Considerando que a área do quadrado é A=pow(l,2) e l=2r, logo: A= pow((2r),2) = pow(4r,2), quando dividimos a área da circunferência pela área do quadrado obtemos:

Área da Circunferência/Área do Quadrado = pow(Pir,2) / pow(4r,2), simplificando pow(r,2) temos: Área da Circunferência/Área do Quadrado = Pi / 4; isolando o Pi, temos:

Pi = 4 x (Área da circunferência / Área do Quadrado)

Considerando a quantidade de pontos aleatórios em cada região como parâmetro de interesse, a área é diretamente proporcional a quantidade de pontos que pode-se depositar dentro da área, desta forma, fugimos do ciclo vicioso que se apresenta na função Pi = 4 x (Área da circunferência Área do Quadrado), portanto, utilizaremos a quantidade de pontos depositados em cada área independente de Pi para calcular o próprio Pi. Neste caso temos: Pi = 4 x (quantidade de pontos dentro da circunferência quantidade de pontos dentro do quadrado). Para saber se os pontos estão dentro da circunferência, será utilizado como parâmetro o tamanho do raio. Se a distância de um ponto aleatório até o centro da circunferência for do tamanho do raio ou menor que o raio, podemos afirmar que o ponto está no interior ou perto da zona de fronteira da circunferência. Pode-se obter a distância entre dois pontos A e B através do Teorema de Pitágoras, onde: dAB = pow(sqrt((xb- xa),2) + pow((yb- ya),2)). Considerando que um ponto aleatório está contido dentro do círculo de raio igual a 1 temos x2 + y2 ¡= 1.

2.3.4. Black Scholes

Black Scholes é um modelo de cálculo matemático que obtém os valores de preços de opções. Esse modelo orienta condição de ganho sem risco [Olga 2007]. Robert Merton, estendeu o modelo de Black Scholes baseando-se nos conceitos apresentados em processos estocásticos [Kawano et al. 2019], que é quando a variável assume valores aleatórios em cada tempo, esse tempo pode ser discreto (intervalos pré-definidos) ou contínuo (constantemente em mudanças) [Bressan 2018]. Segue conforme a Figura 9, o modelo matemático proposto por Black Scholes Merton [Aquino 2007].

$$c = Se^{(b-r)T}N(d_1) - Ke^{-rT}N(d_2)$$

onde

$$d_1 = \frac{ln(S/K) + (b + \sigma^2/2)T}{\sigma\sqrt{T}} \qquad d_2 = d_1 - \sigma\sqrt{T}$$

$b = r$ representa a fórmula proposta por Black e Scholes (1973).

$b = r - q$ representa a fórmula proposta por Merton (1973) com rendimentos de dividendos contínuos q

Figura 9. Fórmula black scholes

3. Resultados

A seguir serão apresentados os resultados obtidos através da implementação das abordagens citadas anteriormente, bem como as saídas dos algoritmos. Os códigos estão disponíveis em: http://bit.do/eV47h.

3.1. Gauss Legendre

A Figura 10 apresenta a função principal que calcula o Pi na implementação de Gauss na versão sequencial. O código é estruturado em 5 funções matemáticas, sendo que a Figura 10 ilustra a função que calcula o Pi final. A função do Pi busca nas outras funções matemáticas o valor de a, p, t e b. Para cada um desses valores supracitados, existe um cálculo diferente. O número de iterações para o Pi convergir foi de 1000000000, conforme especificações do enunciado do trabalho.

```
double pi(int indice){
    double pi_1, pi_2, pi;
    int iter = 1000000000;
    double k = 0;

    for (k = 0; k < iter; k++){
        pi_1 = (pow(a(indice) + b(indice), 2));
        pi_2 = 4 * t(indice);
        pi = pi_1/pi_2;
    }
    return  pi;
}
```

Figura 10. Código sequencial de Gauss

O valor de Pi final calculado de Gauss sequencial após a média de 30 execuções para 1000000000 iterações foi de 3.141593, a média do tempo de execução obtida foi 55.172064 s, desvio padrão de 0.660210 e desvio absoluto de 1.196638.

A Figura 11 apresenta a implementação do algoritmo de Gauss na versão paralela. O código solicita por parâmetro o número de threads que o usuário deseja e o número de iterações. Para cada thread criada o programa divide o processo em pedaços (chucks) para execução entre as threads disponíveis. Para tratar a sincronização é utilizado o bloqueio mutex, conforme é ilustrado a seguir. A função escolhida para ser paralelizada foi a principal que calcula o pi, visto que as outras funções são dependentes dela para o cálculo final.

```
void *pi(void *threadid){
    double pi_1, pi_2, pi;
    double k = 0;
    long long int chunk;
    double j = 0;

    chunk = NUM_PONTOS/numThreads;

    for(j=0;j<chunk;j++){
        pi_1 = (pow(a(indice) + b(indice), 2));
        pi_2 = 4 * t(indice);
        pi = pi_1/pi_2;
     }
    pthread_mutex_lock(&mutexSum);
    pi_final += pi;
    pthread_mutex_unlock(&mutexSum);

    return NULL;
}
```

Figura 11. Código paralelo de Gauss

Os resultados obtidos com a execução do código paralelo foi a seguinte: (2 threads) O valor de pi foi de 3.141593, a média do tempo de execução obtida foi 54.954094 s, desvio padrão de 0.536066 e desvio absoluto de 0.975479; (4 threads) O valor de pi foi de 3.141593, a média do tempo de execução obtida foi 55.093023 s, desvio padrão de 0.281171 e desvio absoluto de 0.510357; (8 threads) O valor de pi foi de 3.141593, a média do tempo de execução obtida foi 61.407258 s, desvio padrão de 0.081549 e desvio absoluto de0.132801; (16 threads) O valor de pi foi de 3.141593, a média do tempo de execução obtida foi 64.379286 s, desvio padrão de 0.330759 e desvio absoluto de 0.513767.

3.2. Borwein

A Figura 12 apresenta a implementação de Borwein na versão sequencial. Foi implementada uma função principal que calcula a fórmula da abordagem. O número de iterações para o Pi convergir foi de 1000000000, conforme especificações do enunciado do trabalho.

```
#define PRECISAO 33219280 //precisao utilizada para o calculo

double pi(int precisao){
    int iter = 1000000000; //numero de iteracoes para computar a formula
    double pi = 0;
    double k = 0;

   for (k = 0; k < iter; k++){
       pi += (1 / (pow(16, k)) * ((120*k*k + 151*k + 47)/(512*k*k*k*k + 1024*k*k*k + 712*k*k + 194*k + 15)));
   }

   return pi;
}
```

Figura 12. Código de Borwein

O valor de Pi final calculado para o algoritmo de Borwein sequencial após a média de 30 execuções para 1000000000 iterações foi de 3.141595, a média do tempo de execução obtida foi 57.849051 s, desvio padrão de 0.731181 e desvio absoluto de 1.263947.

A Figura 13 apresenta a implementação do algoritmo de Borwein na versão paralela. O código solicita por parâmetro o número de threads que o usuário deseja e o número de iterações. Para cada thread criada o programa divide o processo em pedaços (chucks) para execução entre as threads disponíveis. Para tratar a sincronização é utilizado o bloqueio mutex, conforme é ilustrado a seguir.

```
/* CRIA A THREAD */
for(i=1;i<=numThreads;i++){
    pthread_create(&threads[i], NULL, pi, (void *)i);
}
```

```
float pi_final=0.0;
pthread_mutex_t mutexSum; //variavel mutexsum do tipo pthread_mutex
int indice;
long long total_all=0;

void *pi(void *threadid){

    long long int chunk;
    double pi = 0;
    double j = 0;

    chunk = NUM_PONTOS/numThreads; //divide os pedacos de execucao entre as threads disponiveis

    for(j=0;j<chunk;j++){
        //pi += 2+2;
        pi += (1 / (pow(16, j)) * ((120*j*j + 151*j + 47)/(512*j*j*j*j + 1024*j*j*j + 712*j*j + 194*j + 15)));
    }

    //para tratar a sincronizacao utilizamos o bloqueio mutex

    pthread_mutex_lock(&mutexSum);   //bloqueia um objeto mutex
    pi_final += pi;
    pthread_mutex_unlock(&mutexSum); //libera um objeto mutex

    return NULL;
}
```

Figura 13. Código sequencial de Borwein

Os resultados obtidos com a execução do código paralelo foi a seguinte: (2 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 58.881499 s, desvio padrão de 0.457930 e desvio absoluto de 0.777714; (4 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 59.519210 s, desvio padrão

de 1.345606 e desvio absoluto de 2.260793; (8 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 66.275488 s, desvio padrão de 0.197618 e desvio absoluto de 0.298177; (16 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 69.595750 s, desvio padrão de 0.135079 e desvio absoluto de 0.194091;

Vale ressaltar que para cada um dos algoritmos implementados foi analisado através do comando htop se o código realmente estava rodando sequencialmente e paralelamente com o número de threads especificadas. Por exemplo, a Figura 14 ilustra o código de borwein rodando com 4 threads.

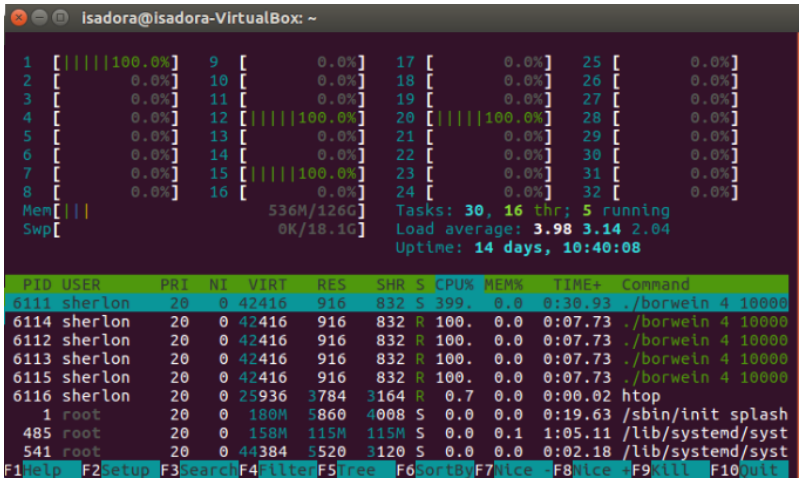

Figura 14. Htop para código paralelo com 4 threads

3.3. Monte Carlo

A Figura 15 apresenta a implementação de Monte Carlo na versão sequencial. Após o algoritmo encontrar de maneira aleatória o número de pontos que estão no círculo e no quadrado, através do Teorema de Pitágoras, esse ponto é verificado se consta dentro do círculo. Somente após essa verificação é que o Pi será calculado através da fórmula matemática apresentada na Seção 2.3.3.

```
do{
    for (i = 1000000000; i; i--, numPontos++){

        x = rand() / (RAND_MAX + 1.0);
        y = rand() / (RAND_MAX + 1.0);

        if (x * x + y * y <= 1){
            numAcertos ++;
        }
    }

    valor = (double) numAcertos/numPontos;
    erro = (sqrt(valor*(1 - valor)/numPontos) * 4);
    valor *=4;
```

Figura 15. Código versão sequencial de Monte Carlo

O valor de pi final calculado para o código de Monte Carlo sequencial após a média de 30 execuções para 1000000000 iterações foi de 3.141604, a média do tempo

de execução obtida foi 25.705757 s, desvio padrão de 3.404330 e desvio absoluto de 13.243453.

A Figura 16 apresenta a implementação do algoritmo de Monte Carlo na versão paralela. A contagem dos pontos no círculo foram paralelizadas agilizando o processo do cálculo de Pi. Nesse contexto, foram utilizadas variáveis locais, desta forma, não houve necessidade de realizar o tratamento de sincronização de variáveis globais. A Thread executa a função ”calcular”que realiza a contagem dos números de pontos no círculo. Logo após essa execução, ocorre a junção dos valores dos pontos no círculo e o valor de Pi é calculado. Após esse processo, as Threads são finalizadas.

```
long thrnum = (long)thread_id;
int tid = (int)thrnum;  //O número da Thread

float *pontos_Circulo = (float *)malloc(sizeof(float));
*pontos_Circulo=0;

float total_iteracao= TOTAL_PONTOS/numThreads;
int contador = 0;

for(contador=0;contador<total_iteracao;contador++){
    float x =  rand() / (RAND_MAX + 1.0);;
    float y =  rand() / (RAND_MAX + 1.0);;

    if (x * x + y * y < 1){
        *pontos_Circulo+=1;
    }
}
```

Figura 16. Código versão paralela de Monte Carlo

Os resultados obtidos com a execução do código paralelo foi a seguinte: (2 threads) O valor de Pi foi de 3.141073, a média do tempo de execução obtida foi 2.354814 s, desvio padrão de 0.748796 e desvio absoluto de 31.798514; (4 threads) O valor de Pi foi de 3.141288, a média do tempo de execução obtida foi 8.498618 s, desvio padrão de 0.856503 e desvio absoluto de 10.078148; (8 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 43.333557 s, desvio padrão de 1.693542 e desvio absoluto de 3.908154; (16 threads) O valor de Pi foi de 3.141350, a média do tempo de execução obtida foi 108.124003 s, desvio padrão de 1.438453 e desvio absoluto de 1.330374;

3.4. Black scholes

A Figura 17 ilustra a parte principal do código de Black Scholes na versão sequencial. Nele são calculados o t, o mean e o intervalo de confiança. Essa abordagem utiliza números aleatórios, para isso, foi desenvolvido um gerador de números aleatórios utilizando a fórmula de distribuição normal. A fórmula de distribuição normal é uma distribuição de probabilidade utilizada para modelar fênomenos.

O intervalo de confiança calculado através da versão sequencial do black scholes com a média de 30 execuções para 1000000000 iterações foi de Intervalo de Confianca = (8.055637, 13.752483), a média do tempo de execução obtida foi 0.000106 s, desvio padrão de 0.000009 e desvio absoluto de 8.621089.

Os resultados obtidos com a execução do código paralelo foi a seguinte: (2 threads) O valor de Pi foi de 3.141073, a média do tempo de execução obtida foi 2.354814

```
trials = (double *) malloc(M*sizeof(double));

struct EstruturaAleatorio state;
initGeradorAleatorio(&state);

/*Calcula: Trials*/

for(i = 0; i < M; i++){
    t = S*exp((r-((1.0/2.0)*pow(sigma, 2.0)))*T + sigma*sqrt(T)*RandomNumber(&state));

    if((t-E) > 0.0)
        trials[i] = exp((-r)*T)*(t-E);
    else
        trials[i] = 0.0;
mean += trials[i];
}

mean = mean/(double)M;
stddev = stdDev(trials, mean, M);
confwidth = 1.96*stddev/sqrt(M);
confmin = mean - confwidth;
confmax = mean + confwidth;
```

Figura 17. Código versão sequencial de black scholes

s, desvio padrão de 0.748796 e desvio absoluto de 31.798514; (4 threads) O valor de Pi foi de 3.141288, a média do tempo de execução obtida foi 0.018529 s, desvio padrão de 0.001066 e desvio absoluto de 5.754485; (8 threads) O valor de Pi foi de 3.141595, a média do tempo de execução obtida foi 0.018072 s, desvio padrão de 0.001207 e desvio absoluto de 6.677341; (16 threads) O valor de Pi foi de 3.141350, a média do tempo de execução obtida foi 0.016997 s, desvio padrão de 0.002026 e desvio absoluto de 11.920817;

A Figura 18 ilustra a parte principal do código de Black Scholes na versão paralela. O código solicita por parâmetro o número de threads que o usuário deseja e o número de iterações. Para cada thread criada o programa divide o processo em pedaços (chucks) para execução entre as threads disponíveis. Para tratar a sincronização é utilizado o bloqueio mutex.

```
chunk = NUM_PONTOS/numThreads; //divide os pedacos de execucao entre as threads disponiveis

trials = (double *) malloc(M*sizeof(double));

struct EstruturaAleatorio state;
initGeradorAleatorio(&state);

/*Calcula: Trials*/
for(j=0;j<chunk;j++){
    for(i = 0; i < M; i++){
        t = S*exp((r-((1.0/2.0)*pow(sigma, 2.0)))*T + sigma*sqrt(T)*RandomNumber(&state));

        if((t-E) > 0.0)
            trials[i] = exp((-r)*T)*(t-E);
        else
            trials[i] = 0.0;
    mean += trials[i];
    }

}

//para tratar a sincronizacao utilizamos o bloqueio mutex

pthread_mutex_lock(&mutexSum);   //bloqueia um objeto mutex
/*Calcula: Média, Desvio Padrão e o Intervalo de Confiança*/

mean = mean/(double)M;
stddev = stdDev(trials, mean, M);
confwidth = 1.96*stddev/sqrt(M);
confmin = mean - confwidth;
confmax = mean + confwidth;

pthread_mutex_unlock(&mutexSum); //libera um objeto mutex
```

Figura 18. Código versão paralela de black scholes

4. Considerações Finais

A elaboração de programas visando desempenho computacional proporciona maior velocidade na obtenção de soluções numéricas. A otimização do código fonte e sua execução paralela possibilitam um desempenho ainda maior quando utilizados juntos. Para um

resultado favoravel é importante levar em consideração conceitos de arquitetura de computadores e sistemas operacionais para utilização ótima da memória cache. Estas tecnicas de otimização quando utilizadas de forma adequada melhoram a eficiência de sistemas computacionais, possibilitando o retorno do resultado de maneira veloz e com menor custo computacional.

Os trabalhos futuros incluem analisar o comportamento de aplicações paralelas tendo como objetivo final listar as melhores técnicas e aplicá-las para atingir um maior desempenho.

Por fim, este trabalho apresentou-se de grande valia para os alunos de sistemas operacionais, visto que foi possível abordar na prática diferentes conceitos aprendidos ao decorrer da disciplina, como exemplo a aplicação de mutex nos códigos paralelos para tratar a sincronização entre threads.

Referências

Aquino, I. O. (2007). Reuso de números aleatórios na simulação de monte carlo para apreçamento de uma carteira de derivativos exóticos. Master's thesis, ICMC/USP, São Paulo.

Azevedo, A. T. (2019). Simulação de monte carlo. `https://www.youtube.com/watch?v=_dqcIbabEJk`. Acesso em 22, junho de 2019.

Bressan, R. (2018). Processos estocástico para finanças. `http://clubedefinancas.com.br/materias/processos-estocasticos-para-financas-uma-introducao/`. Acesso em 22, junho de 2019.

Buttlar, D., Farrell, J., and Nichols, B. (1996). *PThreads Programming: a POSIX Standard for better multiprocessing*. "O'Reilly Media, Inc.".

Damas, L. (2016). *Linguagem C* . Grupo Gen-LTC.

Ferreto, T. (2019). Programação paralela. `https://www.inf.pucrs.br/~gustavo/disciplinas/ppd/material/Prog_Paralela.pdf`. Acesso em 22, junho de 2019.

Foresee, F. D. and Hagan, M. T. (1997). Gauss-newton approximation to bayesian learning. In *Proceedings of International Conference on Neural Networks*, ICNN,97, pages 1930–1935. The organization, IEEE.

Kawano, E., Naue, G., and Bressan, R. F. (2019). O modelo de black-scholes-merton. `http://clubedefinancas.com.br/wp-content/uploads/2018/10/BS.html`. Acesso em 22, junho de 2019.

Olga, L. F. (2007). A teoria da ciência no modelo black-scholes de apreçamento de opções. Master's thesis, FFLCH/USP, São Paulo.

P., S. (2019). Cálculo das constantes elementares clássicas: O caso pi. `http://www.mat.ufrgs.br/~portosil/aplcom1a.html`. Acesso em 22, junho de 2019.

Souza, C. E. (2019). Cálculo de pi através do método monte carlo. `https://www.youtube.com/watch?v=GNAeZqoLfYw`. Acesso em 22, junho de 2019.