Universidade de São Paulo
Bacharelado em Engenharia de Computação
SSC0640-Sistemas Operacionais

Cálculo Aproximado de π e do algoritmo Black Scholes com programação sequencial e paralela

Professor: Júlio Cézar Estrella

Integrantes:
Juan Carlos Elias Obando Valdivia, 7487156
Bruno Ribeiro Helou, 10276852

26/06/2019

Sumário

1. Introdução

O número Pi (π), originalmente definido como a razão entre o perímetro de uma circunferência e seu diametro, é uma constante matemática utilizada nas mais diversas áreas, como engenharia, economia e computação. Na computação, foram criados diversos métodos numéricos para realizar o cálculo do número π de maneira precisa, utilizando-se das ferramentas da programação iterativa e paralela, a fim de dimensionar essa precisão e torná-la o mais eficiente o possível.
Neste trabalho, são apresentadas as implementações de três algoritmos para estimar o número π. Os algoritmos da literatura: Gauss-Legendre, Borwein e Monte Carlo serão comparados em termos de tempo de execução e precisão com relação à suas implementações sequencial e paralela.
Em seguida serão discutidos os resultados para os valores do algoritmo de Black Scholes sequencial e paralelo, utilizado para determinar o preço de derivativos de ações no mercado financeiro.

2. Ferramentas utilizadas

2.1. Hardware

Para estabelecer uma condição igual para todas as implementações, todos os algoritmos serão compilados e executados no mesmo sistema computacional, cujas especificações seguem abaixo:

- Processador: Intel Core i7-8558U 4.00GHz (8 Cores);
- Placa-Mãe: ASUS UX430UAR v1.0;
- Chipset: Intel Xeon E3-1200 v6/7th;
- Memória: 16384Mb;
- Disco: 512 GB Sandisk SD9SN8W5;
- Graphics: Intel UHD 620;
- OS: Ubuntu 18.04;

2.2. Software

Todos os algortimos foram implementados na linguagem C. Foi utilizada a biblioteca GMP (GNU Multiple Precision Arithmetic Library) [Granlund 2002] para manipular Big Numbers. Para a paralelização dos algoritmos, foi utilizado a a API Posix Pthreads que facilita a manipulação, sincronização e criação de threads [Butenhof 1997].

3. Algoritmo de Gauss-Legendre

O algoritmo de Gauss-Legendre é um método de aproximações sucessivas utilizado para estimar o valor de π. Este algoritmo converge rapidamente, mas necessita de uma grande quantidade de memória para a sua execução [GL].

O algoritmo inicializa as seguintes variáveis:

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

A cada iteração atualiza as variáveis da seguinte forma:

$$a_{n+1} = \frac{a_n + b_n}{2} \qquad b_{n+1} = \sqrt{a_n b_n} \qquad t_{n+1} = t_n - p_n(a_n - a_{n+1})^2 \qquad p_{n+1} = 2p_n. \tag{2}$$

Após calcular esses valores, utiliza-se a próxima equação para realizar o cálculo aproximado de π:

$$\pi \approx \frac{(a_{n+1} + b_{n+1})^2}{4t_{n+1}}. \tag{3}$$

3.1. Implementação sequencial do algoritmo

Foi implementado o algoritmo na forma iterativa utilizando as fórmulas acima e utilizando GMP para todas essas variáveis. O número de iterações utilizado foi 10^5 devido ao grande uso da memória por este método.

Na Tabela 1 são apresentados os tempos de 5 execuções do algoritmo Gauss-Legendre sequencial e a média.

Tabela 1. Tempo de execução do algoritmo Gauss-Legendre sequencial

Execução	Tempo (s)
Execução 1	50.24
Execução 2	54.72
Execução 3	53.92
Execução 4	51.68
Execução 5	53.59
Média	52.83

3.2. Implementação paralela do algoritmo

Uma vez que as variáveis a_{n+1} e b_{n+1} usadas no algoritmo de Gauss-Legendre não dependem uma da outra, em cada iteração foram criadas duas threads, uma para calcular a_{n+1} e outra para calcular b_{n+1}. O algoritmo aguardar o fim da execução dessas duas threads para calcular t_{n+1} e p_{n+1}.

Na Tabela 2 são apresentados os tempos de 5 execuções do algoritmo de Gauss-Legendre paralelo e a média.

Tabela 2. Tempo de execução do algoritmo Gauss-Legendre paralelo

Execução	Tempo (s)
Execução 1	60.79
Execução 2	65.13
Execução 3	62.95
Execução 4	58.02
Execução 5	65.75
Média	62.53

3.3. Comparação dos resultados

A seguir é mostrada a proporção entre o tempo de execução do algoritmo Gauss-Legendre paralelo e sequencial:

$$speed_{up} = \frac{t_{sequencial}}{t_{paralelo}} = \frac{62.53}{52.83} = 0.84 \tag{4}$$

4. Algoritmo de Borwein

O método de Borwein é um algoritmo desenvolvido por Jonathan Borwein e Peter Borwein para calcular o valor de $1/\pi$ [Bor]. Para isso, na Convergência Quadrática em 1985, são utilizadas as variáveis a_k e y_k, inicializadas da seguinte forma:

$$a_0 = 2(\sqrt{2}-1)^2 \qquad y_0 = \sqrt{2}-1 \tag{5}$$

A cada iteração o algoritmo atualiza essas variáveis da seguinte forma:

$$y_{k+1} = \frac{1-(1-y_k^4)^{1/4}}{1+(1-y_k^4)^{1/4}} \qquad a_{k+1} = a_k(1+y_{k+1})^4 - 2^{2k+3}y_{k+1}(1+y_{k+1}+y_{k+1}^2) \tag{6}$$

Então a_k converge quarticamente para $1/\pi$; ou seja, cada iteração quadruplica o número de dígitos corretos.

4.1. Implementação sequencial do algoritmo

Foi implementado o algoritmo na forma iterativa utilizando as fórmulas acima e utilizando GMP para todas essas variáveis. O número de iterações utilizado foi 10^5 devido ao grande uso da memória por este método.

Na Tabela 3 são apresentados os tempos de 5 execuções do algoritmo de Borwein sequencial e a média.

4.2. Implementação paralela do algoritmo

Foram criadas duas threads, uma para calcular a_{k+1} e outra para calcular y_{k+1}. Uma vez que o cálculo da variável a_{k+1} precisa de y_{k+1}, e o cálculo de y_{k+1} não precisa de a_{k+1}, foi implementada a estratégia produtor/consumidor. O produtor é o método que calcula y_{k+1}. A seguir mostramos a parte mais importante desse método:

Tabela 3. Tempo de execução do Borwein sequencial

Execução	Tempo (s)
Execução 1	73.58
Execução 2	75.72
Execução 3	79.61
Execução 4	77.57
Execução 5	76.55
Média	76.61

```
void* calcula_y_next (void *arg_y) {
    ...
    sem_wait(&buffer_full);
    pthread_mutex_lock(&mutex);
    mpf_div(*(my_data->y_next), num, den);
    pthread_mutex_unlock(&mutex);
    sem_post(&buffer_empty);
    ...
}
```

O consumidor é o método que calcula a_{k+1}. A seguir mostramos a parte mais importante desse método:

```
void* calcula_a_next (void *arg_a) {
    ...
    sem_wait(&buffer_empty);
    pthread_mutex_lock(&mutex);
    mpf_set(ynext, *(my_data->y_next));
    pthread_mutex_unlock(&mutex);
    sem_post(&buffer_full);
    ...
}
```

Na Tabela 4 são apresentados os tempos de 5 execuções do algoritmo de Borwein paralelo e a média.

Tabela 4. Tempo de execução do Borwein paralelo

Execução	Tempo (s)
Execução 1	97.72
Execução 2	100.04
Execução 3	99.61
Execução 4	96.76
Execução 5	99.73
Média	98.77

4.3. Comparação dos resultados

A seguir é mostrada a proporção entre o tempo de execução do algoritmo de Borwein paralelo e sequencial:

$$speed_{up} = \frac{t_{sequencial}}{t_{paralelo}} = \frac{78.61}{98.77} = 0.79 \tag{7}$$

5. Algoritmo de Monte Carlo

O método de Monte Carlo é um termo utilizado para se referir a qualquer método que resolve um problema gerando números aleatórios e observando se uma dada fração desses números satisfaz uma propriedade previamente estabelecida.

No caso de calcular π, a ideia é colocar a circunferência dentro de um quadrado e sortear pontos aleatórios dentro da figura. Para uma grande amostra de pontos, ele deve convergir para o resultado esperado, ou seja, para π [MC].

Seja P a probabilidade do ponto estar dentro da circunferência, definido por:

$$P = \frac{A_{circ}}{A_{quad}} \tag{8}$$

em que A_{quad} é a área do quadrado e A_{circ} a área do círculo. Se utilizamos um círculo de rádio unitário, $A_{circ} = \pi$ e $A_{quad} = 4$. Logo, $\pi = P * 4$.

Para calcular P, são sorteados randomicamente dois números representando as coordenadas x e y de um ponto. Para definir se esse ponto está dentro, utiliza-se a condição que $x^2 + y^2 < 1$. Assim, para n pontos, temos:

$$\pi \approx \frac{n_{dentro}}{n} * 4 \tag{9}$$

Em que n_{dentro} representa o número de pontos que estão dentro do círculo e que chamaremos de acertos.

5.1. Implementação sequencial do algoritmo

Na implementação sequencial deste algoritmo, o pseudocódigo 1 foi tomado como base. Foram realizadas 10^9 iterações. Neste caso foi utilizado GMP apenas para a linha 9.

Na Tabela 5 são apresentados os tempos de 5 execuções do algoritmo de Monte Carlo sequencial e a média.

Tabela 5. Tempo de execução do algoritmo de Monte Carlo sequencial

Execução	Tempo (s)
Execução 1	1.57
Execução 2	1.60
Execução 3	1.57
Execução 4	1.56
Execução 5	1.57
Média	1.57

Algorithm 1 Algoritmo que estima π utilizando o método de Monte Carlo

Entrada: O número n de iterações para o Algoritmo que estima π utilizando o método de Monte Carlo

Saída : Valor estimado de π após n iterações

```
acertos = 0
para (i = 0 até n) faça
    x = rand(0, 1)
    y = rand(0, 1)
    se (x² + y² < 1) então
        acertos = acertos + 1
    fim
fim
π = (4 * acertos)/n
retorna π
```

5.2. Implementação paralela do algoritmo

Visto que o algoritmo de Monte Carlo é bastante paralelizável devido ao fato de possuir poucas dependências, os cálculos foram divididos entre K threads. Neste trabalho foi escolhido K=4. Dessa forma, para $n = 10^8$ iterações, cada thread realizaria $n/4$ iterações do algoritmo paralelamente. Quando as threads terminarem, a estimativa de π para cada thread é calculada. Assim, basta somar a quantidade de acertos obtida por cada thread e dividir pelo número de threads utilizadas.

Na Tabela 6 são apresentados os tempos de 5 execuções do algoritmo de Monte Carlo paralelo e a média.

Tabela 6. Tempo de execução do algoritmo de Monte Carlo paralelo

Execução	Tempo (s)
Execução 1	1.11
Execução 2	1.08
Execução 3	1.15
Execução 4	1.16
Execução 5	1.08
Média	1.11

5.3. Comparação dos resultados

A seguir é mostrada a proporção entre o tempo de execução do algoritmo de Monte Carlo paralelo e sequencial:

$$speed_{up} = \frac{t_{sequencial}}{t_{paralelo}} = \frac{1.57}{1.11} = 1.41 \tag{10}$$

6. Algoritmo de Black Scholes

O algoritmo de Black Scholes é utilizado para determinar o preço de derivativos de ações no mercado financeiro [Pontello 2010].

6.1. Implementação sequencial do algoritmo

Na implementação sequencial deste algoritmo, o pseudocódigo 2 foi tomado como base.

Algorithm 2 Algoritmo Black Scholes

Entrada: S: valor da ação, E: preço de exercício da opção, r: taxa de juros livre de risco (SELIC), σ: volatilidade da ação, T : tempo de validade da opção, M: número de iterações

Saída : Intervalo de confiança

para *(*$i = 0$ *até* $M - 1$*)* **faça**
  $t = S \cdot exp((r - 1/2\sigma^2) \cdot T + \sigma\sqrt{T} \cdot randomNumber()$
  $trials[i] = exp(-r \cdot T) \cdot max\{t - E, 0\}$
fim
$mean = mean(trials)$
$stddev = stddev(trials, mean)$
$confwidth$=$1.96 \cdot stddev/\sqrt{(M)})$
$confmin = mean - confwidth$
$confmax = mean + confwidth$
retorna $confmin, confmax$

Na Tabela 7 são apresentados os tempos de 5 execuções do algoritmo de Black Scholes sequencial e a média.

Tabela 7. Tempo de execução do algoritmo Black Scholes sequencial para entrada_blackscholes.txt

Execução	Tempo (s)
Execução 1	0.01
Execução 2	0.01
Execução 3	0.01
Execução 4	0.01
Execução 5	0.01
Média	0.01

6.2. Implementação paralela do algoritmo

Visto que o algoritmo Black Scholes com Monte Carlo é bastante paralizável devido ao fato de possuir poucas dependências, os cálculos foram divididos entre K threads. Neste trabalho foi escolhido K=4. Dessa forma, para M iterações, cada thread realizaria $M/4$ iterações do algoritmo paralelamente. Após as threads terminarem, basta juntar as threads para calcular a média, desvio padrão e o intervalo de confiança médio.

Na Tabela 8 são apresentados os tempos de 5 execuções do algoritmo de Black Scholes paralelo e a média.

6.3. Comparação dos resultados

A seguir é mostrada a proporção entre o tempo de execução do algoritmo de Black Scholes paralelo e sequencial:

Tabela 8. Tempo de execução do algoritmo Black Scholes paralelo

Execução	Tempo (s)
Execução 1	0.00
Execução 2	0.00
Execução 3	0.00
Execução 4	0.00
Execução 5	0.00
Média	0.00

$$speed_{up} = \frac{t_{sequencial}}{t_{paralelo}} = \frac{0.01}{0.00} = \infty \tag{11}$$

7. Problemas e dificuldades enfrentados

Visto que foi nossa primeira experiência com programação paralela, as maiores dificuldades enfrentadas foram pensar em como paralelizar os algoritmos sequenciais. Tivemos que aprender como criar, manipular e sincronizar as threads usando Posix Pthreads. Para isso os conceitos aprendidos durante as aulas teóricas foram fundamentais para o desenvolvimento do projeto.
Outra dificuldade enfrentada foi o uso da biblioteca GMP, visto que aprender a sua sintaxe demandou um grande estudo prévio.

8. Forma de execução dos programas

No $README.md$ em $https://github.com/jceoval/SO2019-GRUPO16$ está indicado detalhadamente a forma de execução dos programas.

9. Conclusão

Os algoritmos de Monte Carlo e Black Scholes apresentarem ganhos consideráveis quando implementados paralelamente. Pelo fato destes algoritmos serem poucos dependentes de estados anteriores, eles são altamente paralelizavéis.
Já os algoritmos de Borwein e Gauss Legendre, por possuírem alta dependência entre seus estados passados, eles dificultam o ganho na paralelização.

Referências

Borwein's algorithm. https://en.wikipedia.org/wiki/Borwein_algorithm. Accessed: 2019-06-01.

Gauss-Legendre algorithm. https://en.wikipedia.org/wiki/Gauss-Legendre_algorithm. Accessed: 2019-06-01.

Module for Monte Carlo Pi. http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html. Accessed: 2019-06-01.

Butenhof, D. R. (1997). *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Granlund, T. (2002). GNU multiple precision arithmetic library 4.1.2. http://swox.com/gmp/.

Pontello, B. V. (2010). Apreçando opções via método de Monte-Carlo. Trabalho de conclusão de curso.