Comparação de *speedup* de programas paralelos e sequenciais

F. N. Candiani *M. L. Junior* *R. H. F. Minami*

Technical Report - PC2014-grupo09-turmab - Relatório Técnico
March - 2014 - Março

UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO

Comparação de *speedup* de programas paralelos e sequenciais

Fernando N. Candiani* Marcius L. Junior† Rafael H. F. Minami‡

Sumário

1 Introdução **3**
1.1 Monte Carlo . . . 3
1.2 Black-Scholes . . . 3

2 Método Monte Carlo para cálculo de π **3**
2.1 Algoritmo . . . 4
2.2 Como rodar o programa . . . 4
2.2.1 Sequencial . . . 4
2.2.2 Paralelo . . . 5
2.3 Dificuldades e Soluções . . . 5
2.4 Resultados Obtidos . . . 5

3 Modelo de Black-Scholes **6**
3.1 Algoritmo . . . 7
3.2 Como rodar o programa . . . 8
3.2.1 Sequencial . . . 8
3.2.2 Paralelo . . . 8
3.3 Dificuldades e Soluções . . . 8
3.4 Resultados Obtidos . . . 8

4 Metodologia de execução dos experimentos **10**
4.1 Dificuldades e Soluções . . . 10

5 Cálculo da média e da variância dos dados **11**

6 Como calcular o *Speedup* **11**

7 Hardware utilizado **11**

8 Conclusão **12**

Appendices **13**

*Inst. de Ciências Matemáticas e Computação, USP. N USP: 7239131 fncandiani@usp.br
†Inst. de Ciências Matemáticas e Computação, USP. N USP: 7277433 marcius@usp.br
‡Inst. de Ciências Matemáticas e Computação, USP. N USP: 7573187 rafahiroki@usp.br

Lista de Figuras

1 Método de Monte Carlo aplicado para encontrar o valor aproximado de π 4
2 Gráfico *threads* x tempo(s) para o algoritmo de cálculo de π. 6
3 Algoritmo do modelo de Black Scholes 7
4 Gráfico *threads* x tempo(s) para o algoritmo de *Black-Scholes*. 10

Lista de Tabelas

1 Média e Variância para os dados do *Monte Carlo* sequencial. 6
2 Média e Variância para os dados do *Monte Carlo* paralelo. 7
3 Média e Variância para os dados do *Black-Scholes* sequencial. 9
4 Média e Variância para os dados do *Black-Scholes* paralelo. 9

Lista de Códigos

1 Script para execução dos experimentos 13
2 Cálculo da média e variância dos dados. 15
3 Método de Monte Carlo sequencial. 16
4 Método de Monte Carlo paralelo. 17
5 Método de Black-Scholes sequencial. 19
6 Método de Black-Scholes paralelo. 21

Resumo

In this assignment we used the Monte Carlo simulation to calculate the π number and also the Black Scholes model. In each case, we build an program that use different programming paradigms; one sequencial and the other parallel. After executing each program multiples times we calculated the speedup so we could compare the improvement of the usage of parallel paradigm rather than sequencial. This article shows the results obtained during the experiments and problems regarding the implementation of the parallel algorithm and its execution.

1 Introdução

Nesse trabalho utilizamos a simulação de Monte Carlo para calcular o número π e também o modelo de Black-Scholes. Foram desenvolvidas duas versões de cada algoritmo, uma sequencial e outra paralela utilizando *threads*. Neste relatório nós mostraremos os resultados obtidos e faremos a comparação entre a versão sequencial e a paralela dos dois algoritmos. Todos os códigos e saídas dos programas, bem como os métodos de compilação, podem ser encontrados dentro do nosso repositório do *Google Code*[1].

1.1 Monte Carlo

Definimos por um método de Monte Carlo (MMC), um método qualquer de uma classe de método estatísticos, que inferem de amostragens massivas para obter resultados numéricos, isto é, repetem exaustivamente simulações, afim de calcular probabilidades heuristicamente, tal com se, de fato, se registrassem os resultado reais de jogos em cassino, da onde derivou-se o nome.

Apesar de ter se despertado o interesse por esse método durante a Segunda Guerra Mundial, na construção de uma bomba atômica, quando foi batizado com o nome. A técnica de Monte Carlo já era utilizada em uma discussão das equações de Boltzmann, onde é calculado a função de distribuição de partículas em estados diferente.

1.2 Black-Scholes

A fórmula de Black-Scholes for apresentada inicialmente em [1]. A base para sua pesquisa utilizou o trabalho desenvolvido por pesquisadores como Jack L. Treynor, Paul Samuelson, A. James Boness, Sheen T. Kassouf, e Edward O. Thorp. O conceito fundamental de *Black-Scholes* é que se uma ação é negociada, então a opção é precificada.

Conforme mostrado em [2] Robert C. Merton foi o primeiro a mostrar uma expansão da compreensão matemática do modelo de precificação de opções, e cunhou o termo modelo de precificação de opções de *Black-Scholes*.

2 Método Monte Carlo para cálculo de π

Uma simplificação do algoritmo de Metropolis utilizamos o Monte Carlo de Erro-Unilateral para calcular a aproximação de π, que consiste em uma distribuição amostral onde o evento não pertence é garantida a resposta - garantimos que o ponto que não pertencem ao círculo, com certeza se encontram no quadrado; porém quando o amostra pertence não é garantida que a resposta esta correta - temos de verificar se cada conjunto de coordenadas se encontra no interior do círculo.

[1] https://code.google.com/p/pc2014-grupo09-turmab/

2.1 Algoritmo

Neste algoritmo, para utilizar o método de Monte Carlo nós consideramos uma circunferência inscrita em um quadrado. A relação entre as áreas é $\frac{\pi}{4}$ e por isso nós conseguimos chegar ao valor de π utilizando o método de Monte Carlo.

Utilizando uma função que gera números randômicos, conseguimos escolher pontos aleatórios dentro desse quadrado. No fim teremos o número de pontos que estão dentro da circunferência e o número total de pontos gerados. Figura 1

A razão entre os dois números é uma aproximação da razão das duas áreas que é $\frac{\pi}{4}$, portanto, devemos multiplicar essa razão por 4 e assim teremos uma aproximação do número π.

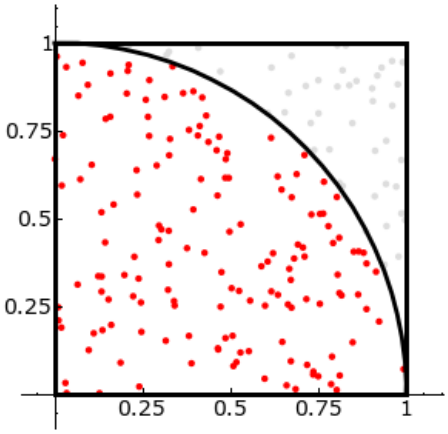

Figura 1: Método de Monte Carlo aplicado para encontrar o valor aproximado de π

Fizemos duas versões desse algoritmo, uma totalmente sequencial e outra paralela utilizando *threads*. Na versão com *threads* tivemos que observar o que poderia ser paralelizado. A maneira que encontramos foi paralelizar o laço que gera os pontos aleatórios.

2.2 Como rodar o programa

Nesta seção é apresentado como é feita a compilação e execução e os parâmetros de execução e, também, os de entrada do programa.

2.2.1 Sequencial

Estando dentro do diretório *montecarlo/sequencial* onde contém os arquivos *makefile* e *montecarlo.c*, que pode ser visto em Código 3, devemos executar os seguintes comandos:

usuer@pc-name:dir $ make

O comando *make* irá compilar o Código 3 passando todos os parâmetros de lincagem das bibliotecas extras.

user@pc-name:dir $/usr/bin/time -f "%e"./montecarlo >> path/file.out 2>&1

O comando irá executar o programa para o cálculo de π, redirecionando o tempo e o valor de π para *file.out*.

2.2.2 Paralelo

Estando dentro do diretório *montecarlos/paralelo* onde contém os arquivos *makefile* e *montecarlo.c*, que pode ser visto em Código 4, devemos executar os seguintes comandos:

usuer@pc-name:dir $ make

O comando *make* irá compilar o Código 4 passando todos os parâmetros de lincagem das bibliotecas extras.

user@pc-name:dir $/usr/bin/time -f "%e"./montecarlo [number of threads] >> path/file.out 2>&1

O comando irá executar o programa para o cálculo do valor de π, recebendo como parâmetro o *number of threads* e redirecionando o tempo e o valor de π para *file.out*.

2.3 Dificuldades e Soluções

Para o problema sequencial tivemos uma dificuldade para entender o funcionamento da função *double erand48(unsigned short xsubi[3])* pois essa recebe um parâmetro que serve como seed para os valores randômicos que serão gerados posteriormente dentro da função, ou seja, os valores do vetor *xsubi* eram, inicialmente, inicializados com um valor constante e a cada execução do programa a saída dos valores randômicos eram sempre as mesmas. Para solucionar esse problema foi utilizado a função *time(NULL)* nas posições 1 e 2 do vetor *xsubi*. Posteriormente, a função foi mudada para a *double drand48(void)* pois esta é *thread-safe* e para inicializar a semente da função randômica utilizamos a função *void srand48(long int seedval)*.

Para o problema paralelo tivemos problema na modelagem da solução, ou seja, como iriamos resolver o problema de forma paralelizada. A solução foi discutida entre colegas na sala e conseguimos extrair um modelo que, inclusive, foi posteriormente discutido em sala. A solução envolvia um vetor global com o tamanho do número de *threads* em que o programa vai ser separado, ou seja, em um programa que tem 16 *threads* teríamos um vetor com tamanho 16. E em cada posição do vetor a sua *thread* correspondente irá atualizar o número de acertos que ocorreu, entenda acertos como pontos (x, y) que estão dentro da circunferência de raio igual a 1.

2.4 Resultados Obtidos

Por se tratar de um problema em que o número de interações é muito grande, que em nosso caso é definido como um parâmetro 10^9, o tempo de execução sempre é razoalmente grande, conforme pode ser visto dentro das saídas geradas dentro do repositório do *Google Code* 1.

Para os programas sequencial e paralelo os dados obtidos são apresentados na Tabela 1 e Tabela 2, respectivamente.

Conforme pode ser visto, tanto nas Tabelas 1 e 2 quanto na Figura 2, houve uma diminuição considerável no tempo de execução entre o programa sequêncial e paralelo, apenas para um número de threads maior que 8, exclusive. O *speedup* para 4 e 8 *threads* foi < 1 o que representa um aumento no tempo de execução do programa paralelo em relação ao sequencial. Isso, possivelmente, se deve

	Média	Variância
π	3.141578	0.000000
Tempo	39.735000	0.048050

Tabela 1: Média e Variância para os dados do *Monte Carlo* sequencial.

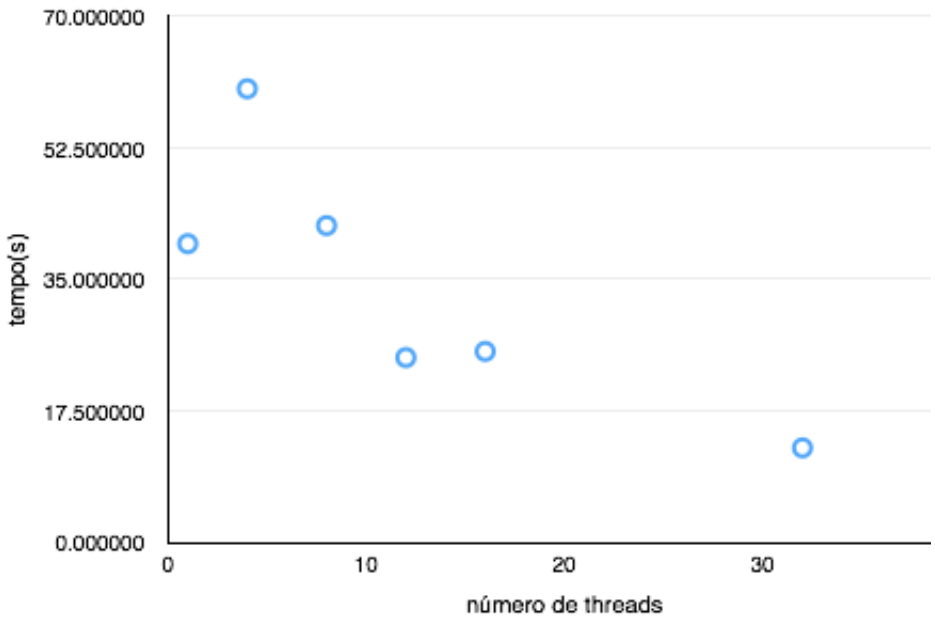

Figura 2: Gráfico *threads* x tempo(s) para o algoritmo de cálculo de π.

ao fato de o algoritmo paralelo adicionar complexidade de memória, ou seja, dentro do Código 4 podemos ver que foi adicionado muito mais alocação de memório em relação ao Código 3, o que fez ocorrer o aumento de operação de memória e por consequência o tempo de execução do programa.

Para 12 e 16 *threads*, o *speedup* calculado ficou em torno de 1.61, ou seja, o programa paralelo foi cerca de 1.61 vezes mais rápido que o sequencial, o que representa um aumento razoavelmente bom para o tempo de execução. Já para 32 *threads* o *speedup* calculado foi $3,15$, o que representa um aumento muito considerável, para entender o cálculo do *speedup* consulte a Seção 6.

3 Modelo de Black-Scholes

O modelo de Black-Scholes do mercado para um ativo faz as seguintes suposições explícitas:

1. É possível emprestar e tomar emprestado a uma taxa de juros livre de risco constante e conhecida;
2. O preço segue um movimento Browniano geométrico com tendência (drift) e volatilidade constantes;
3. Não há custos de transação;
4. A ação não paga dividendos;
5. Não há restrições para a venda a descoberto.

4 *Threads*		
	Média	Variância
π	3.141586	0.000000
Tempo	60.325427	1.015872
8 *Threads*		
	Média	Variância
π	3.141592	0.000000
Tempo	42.139608	4.870509
12 *Threads*		
	Média	Variância
π	3.141586	0.000000
Tempo	24.598560	0.660768
16 *Threads*		
	Média	Variância
π	3.141588	0.000000
Tempo	25.417140	0.785086
32 *Threads*		
	Média	Variância
π	3.141588	0.000000
Tempo	12.595500	0.146668

Tabela 2: Média e Variância para os dados do *Monte Carlo* paralelo.

3.1 Algoritmo

Para o modelo de Black-Scholes nós seguimos o algoritmo apresentado na Figura 3. Este algoritmo utiliza o método de Monte Carlo para calcular o preço das opções européias. As variáveis de entrada desse algoritmo são:

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 e M : número de iterações.

Pseudo-código do algoritmo de Black Scholes com Monte Carlo

1: **for** $i = 0$ to $M - 1$ **do**
2: $t := S \cdot \exp\left((r - \frac{1}{2}\sigma^2) \cdot T + \sigma\sqrt{T} \cdot \text{randomNumber}()\right)$ ▷ t é uma variável temporária
3: trials [i] := $\exp(-r \cdot T) \cdot \max\{t - E, 0\}$
4: **end for**
5: mean := *mean*(trials)
6: stddev := *stddev*(trials , mean)
7: confwidth := $1.96 \cdot \text{stddev}/\sqrt{M}$ ▷ Cálculo do intervalo de confiança
8: confmin := mean − confwidth
9: confmax := mean + confwidth

Adaptado de http://www.cs.berkeley.edu/~yelick/cs194f07

Figura 3: Algoritmo do modelo de Black Scholes

3.2 Como rodar o programa

Nesta seção é apresentado como é feita a compilação e execução e os parâmetros de execução e, também, os de entrada do programa.

3.2.1 Sequencial

Estando dentro do diretório *blackscholes/sequencial* onde contem os arquivos *makefile* e *blackscholes.c*, conforme pode ser visto em Código 5, devemos executar os seguintes comandos:

usuer@pc-name:dir $ make
O comando *make* irá compilar o Código 5 passando todos os parâmetros de lincagem das bibliotecas extras.

user@pc-name:dir $*/usr/bin/time -f "%e"./blackshcoles < file.in >> path/file.out 2>&1*

O comando irá executar o programa para cálculo de aproximação de valor confiável para aplicação de investimentos, recebendo a entra a de *file.in*, redirecionando o tempo e as saídas do programa para *file.out*.

3.2.2 Paralelo

Estando dentro do diretório *blachsholes/paralelo* onde contem os arquivos *makefile* e *blackshcoles.c*, conforme pode ser visto em Código 6, devemos executar os seguintes comandos:

usuer@pc-name:dir $ make

O comando *make* irá compilar o Código 6 passando todos os parâmetros de lincagem das bibliotecas extras.

user@pc-name:dir $*/usr/bin/time -f "%e"./blackshcoles [numeber of threads] < file.in >> path/file.out 2>&1*
O comando irá executar o programa para cálculo de aproximação de valor confiável para aplicação de investimentos, recebendo como parâmetro o *number of threads* e na entrada padrão o arquivo *file.in*, redirecionando o tempo e as saídas do programa para *file.out*.

3.3 Dificuldades e Soluções

O maior problema encontrado dentro do modelo de Black-Scholes foi a distribuição dos números randômicos. Os números randômicos desse modelo devem seguir uma distribuição normal e não uma distribuição uniforme conforme exigido no cálculo de π. Com isso muitos problemas surgiram na hora de verificar se a solução aplicada estava correta.

Tanto no desenvolvimento do programa sequencial quanto do programa paralelo não houve nenhum problema pois já estávamos habituados com a forma de pensamento para solucionar o problema.

3.4 Resultados Obtidos

Por se tratar de um problema em que o número de interações não é muito grande, que em nosso caso é recebido como um parâmetro M, girando em torno, na grande maioria das vezes, de 1000 a

5000 iterações, o tempo de execução sempre é pequeno, conforme pode ser visto dentro das saídas geradas dentro do repositório do *Google Code* 1.

Para os programas sequencial e paralelo os dados obtidos são apresentados na Tabela 3 e Tabela 4, respectivamente.

	Média	Variância
Média Calculada	99.904995	0.000000
Intervalo de confiança	0.793321	0.000000
Tempo	0.020920	0.000015

Tabela 3: Média e Variância para os dados do *Black-Scholes* sequencial.

4 *Threads*		
	Média	Variância
Média Calculada	104.134842	0.030029
Intervalo de confiança	0.184846	0.000000
Tempo	0.010010	0.000000
8 *Threads*		
	Média	Variância
Média Calculada	104.167049	0.052991
Intervalo de confiança	0.184623	0.000000
Tempo	0.009520	0.000005
12 *Threads*		
	Média	Variância
Média Calculada	104.113276	0.127136
Intervalo de confiança	0.185021	0.000001
Tempo	0.009040	0.000009
16 *Threads*		
	Média	Variância
Média Calculada	104.288167	0.077040
Intervalo de confiança	0.185034	0.000001
Tempo	0.009330	0.000006
32 *Threads*		
	Média	Variância
Média Calculada	104.076457	0.225480
Intervalo de confiança	0.185103	0.000003
Tempo	0.010000	0.000000

Tabela 4: Média e Variância para os dados do *Black-Scholes* paralelo.

Conforme pode ser visto, tanto nas Tabelas 3 e 4 quanto na Figura 4, houve uma diminuição considerável no tempo de execução entre o programa sequêncial e paralelo. O *speedup* calculado ficou em torno de 2, ou seja, o programa paralelo foi cerca de 2 vezes mais rápido que o sequencial, o que representa um aumento considerável para o tempo de execução, para entender o cálculo do speedup consulte a Seção 6.

Podemos observar também pela Figura 4 que um aumento no número de *threads* não significa um aumento de eficiência visto que está limitado ao número de núcleos que o hardware possui, pois

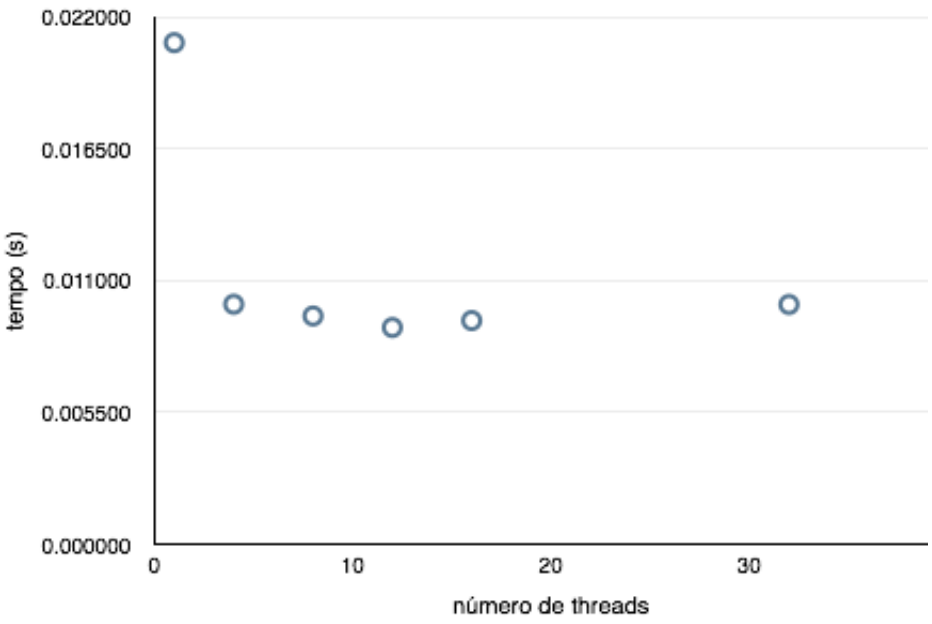

Figura 4: Gráfico *threads* x tempo(s) para o algoritmo de *Black-Scholes*.

quanto mais *threads*, maior é o número de chaveamentos que ele deve fazer, portanto existe um ponto máximo em que se tem um ganho real, depois disso o tempo de chaveamento é grande e a eficiência do programa é afetada.

4 Metodologia de execução dos experimentos

Para a execução dos programas feitos, foi criado um *script*, mostrado em Código 1, que é capaz de rodar todos os programas definidos (Monte Carlo - sequencial e paralelo, Black-Scholes - sequencial e paralelo) e para cada programa paralelo o *script* altera o número de *threads* que este vai executar.

Esse *script* faz a compilação e execução dos programas e após a execução esse redireciona todos os resultados para um arquivo. Esse arquivo, que foi formatado de forma simples, é utilizado para o cálculo da média e variância, conforme mostrado na Seção 5, dos dados que foram gerados pelos programas, assim podemos analisar melhor os resultados gerados.

4.1 Dificuldades e Soluções

Durante a execução do *script* tivemos alguns problemas na conexão com o cluster gerando *broken pipes* com este. Por isso, tivemos que fazer algumas alterações na chamada dos métodos criados dentro do *script*, por exemplo, quando estavamos executando o Monte Carlo paralelo ocorreu um problema de conexão e tivemos que comentar a chamada de função do Monte Carlo sequencial pois este já havia sido executado.

Outro problema gerado pela falha na conexão foi que o arquivo já possuía dados gerados na execução anterior e por isso o tipo de *pipe*, que antes era um *pipe* de redirecionamento simples, foi alterado para um *pipe* de concatenação para que os dados gerados na execução anterior fossem mantidos e não fosse necessário a execução do *script* desde do início.

Outra problema encontrado no desenvolvimento do *script* foi o redirecionamento da saída do programa */usr/bin/time* pois este redireciona sua saída para a *stderr* ao invés da *stdout*. A solução encontrada foi acrescentar uma *flag* ao redirecionamento que faz com que o *stderr* também seja redirecionado ao arquivo passado, isso foi obtido com o comando "2>&1".

5 Cálculo da média e da variância dos dados

Como a execução de cada programa foi feita múltiplas vezes, no nosso caso foi executado 1000 vezes, seria inviável calcular a média e o desvio padrão dos dados gerados de forma manual. Por isso, foi criado um programa, conforme mostrado em Código 2, que dado o tamanho de um vetor e o número de variáveis a ser analisado calculará a média e variância, levando em consideração que os dados estão distribuídos da seguinte maneira:

quantidade dos dados número de variáveis
dado 1 da variável a
dado 1 da variável b
dado 2 da variável a
dado 2 da variável b
e assim por diante.

Os dados gerados por esse programa serão apresentados dentro das seções respectivas aos seus dados.

6 Como calcular o *Speedup*

Como o próprio termo já diz, o *speedup* é o ganho na velocidade quando se compara duas situações. No nosso caso usamos o *speedup* para calcular o ganho na velocidade de execução entre um programa sequencial e um programa paralelo, por isso para saber qual a vantagem entre os dois devemos resolver a seguinte razão:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

onde S_p é o *speedup*,
T_1 é o tempo médio do algoritmo sequencial e
T_p é o tempo médio do algoritmo paralelo.

Espera-se que esse número seja sempre > 1 pois isso mostra que o houve um ganho no algoritmo paralelo em relação ao sequencial. Para ter um bom *speedup* o valor S_p esperado deve ser > 2, ou seja, houve uma diminuição de cerca de 50% no tempo de execução do programa sequencial em relação ao paralelo.

7 Hardware utilizado

Para todos os programas o mesmo hardware foi utilizado, o do *cluster*. Para obter a descrição detalhada do hardware utilizamos o seguinte programa, com um parâmetro:

phoronix-test-suite detailed-system-info

A saída obtida foi a seguinte (a grande maioria dos dados são auto-explicativos e por isso não será discutido):

Phoronix Test Suite v4.8.2
System Information

Hardware:
Processor: Intel Core 2 Quad Q9400 @ 2.67GHz (4 Cores),
Motherboard: Gigabyte G41MT-S2P,
Chipset: Intel 4 DRAM + ICH7,
Memory: 8192MB,
Disk: 500GB Western Digital WD5000AACS-0,
Graphics: Intel 4 IGP,
Audio: VIA VT2020,
Network: Realtek RTL8111/8168/8411
Software:
OS: Ubuntu 12.04, Kernel: 3.2.0-54-generic (x86_64), Compiler: GCC 4.6,
File-System: ext4, Screen Resolution: 640x480

Processor:
Core Count: 4
Thread Count: 4
Cache Size: 3072 KB
Instruction Set Extensions: SSE 4.1
AES Encryption: NO
Energy Performance Bias: NO
Virtualization: VT-x
Compiler Configuration: –build=x86_64-linux-gnu –disable-werror –enable-checking=release –enable-clocale=gnu –enable-gnu-unique-object –enable-languages=c,c++,fortran,objc,obj-c++ –enable-libstdcxx-debug –enable-libstdcxx-time=yes –enable-nls –enable-objc-gc –enable-plugin –enable six –host=x86_64-linux-gnu –target=x86_64-linux-gnu –with-arch-32=i686 –with-tune=generic -v
Disk Scheduler: CFQ
Disk Mount Options: barrier=1,data=ordered,relatime,rw,user_xattr
Cpu Scaling Governor: acpi-cpufreq ondemand

8 Conclusão

A partir dos resultados obtidos através de nossos algoritmos fica clara a vantagem de se paralelizar processos para se obter um ganho considerável de eficiência. Em nossos testes também observamos que existe uma relação entre o número de núcleos que o hardware possui e o número máximo de threads que ele pode suportar, depois desse número máximo a eficiência é diminuída. Isso ocorre por causa da quantidade de chaveamentos que precisam ocorrer para que as *threads* executem, aumentando, assim, o tempo de execução.

Appendices

Script para execução dos experimentos

```
#!/bin/bash

size="1000"
file="out.out"
threads="4 8 12 16 32"

method="Montecarlo Sequencial"
directory="montecarlo/sequencial"

montecarlo_seq(){
  path="montecarlo/sequencial"

  make --silent --directory=$path clean
  make --silent --directory=$path

  echo "Montecarlo Sequencial\n"
  echo "$size 2" > $path/$file
  for i in $(seq $size); do
    /usr/bin/time -f "%e" ./$path/montecarlo >> $path/$file 2>&1
  done

  make --silent --directory=$path clean
}

blackscholes_seq(){
  path="blackscholes/sequencial"

  make --silent --directory=$path clean
  make --silent --directory=$path

  echo "Blackscholes Sequencial\n"
  echo "$size 3" > $path/$file
  for i in $(seq $size); do
    /usr/bin/time -f "%e" ./$path/blackscholes < $path/entrada_blackscholes.txt >>
        $path/$file 2>&1
  done

  make --silent --directory=$path clean
}

montecarlo_par(){
  path="montecarlo/paralelo"

  make --silent --directory=$path clean
  make --silent --directory=$path

  echo "Montecarlo Paralelo\n"

  for j in $threads; do
    echo "$size 2" > $path/$j$file
    for i in $(seq $size); do
      /usr/bin/time -f "%e" ./$path/montecarlo $j >> $path/$j$file 2>&1
    done
  done
```

```
55    make --silent --directory=$path clean
}
57
blackscholes_par(){
59    path="blackscholes/paralelo"

61    make --silent --directory=$path clean
    make --silent --directory=$path
63
    echo "Blackscholes Paralelo\n"
65
    for j in $threads; do
67      echo "$size 3" > $path/$j$file
      for i in $(seq $size); do
69        /usr/bin/time -f "%e" ./$path/blackscholes $j < $path/entrada_blackscholes.txt
              >> $path/$j$file 2>&1
      done
71    done

73    make --silent --directory=$path clean
}
75
mean_variance_seq(){
77    path="mean_var"
    mean_file="mean_var.out"
79
    echo "Calculando media e variancia para $method\n"
81    ./$path/mean_var < $directory/$file > $directory/$mean_file
}
83
mean_variance_par(){
85    path="mean_var"
    mean_file="mean_var.out"
87
    echo "Calculando media e variancia para $method\n"
89    for i in $threads; do
      ./$path/mean_var < $directory/$i$file > $directory/$i$mean_file
91    done
}
93
main(){
95    clear
    path="mean_var"
97
    make --silent --directory=$path clean
99    make --silent --directory=$path

101   montecarlo_seq
    mean_variance_seq
103
    montecarlo_par
105   method="Montecarlo Paralelo"
    directory="montecarlo/paralelo"
107   mean_variance_par

109   blackscholes_seq
    method="Blackscholes Sequencial"
111   directory="blackscholes/sequencial"
```

```
  mean_variance_seq

  blackscholes_par
  method="Blackscholes Paralelo"
  directory="blackscholes/paralelo"
  mean_variance_par

  make --silent --directory=$path clean
}

main
```

Código 1: Script para execução dos experimentos

Código para cálculo da média e variância dos dados

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double mean(double *values, unsigned int size){
  if(size != 0){
    double mediaCalc = 0.0;
    unsigned int i = 0;
    for(;i<size;i++)
      mediaCalc += values[i];
    return mediaCalc/size;
  }
  return 0.0;
}

double variance(double *values, double mean, unsigned int size){
  if(size != 1){
    double variancia1Calc = 0.0f;
    unsigned int i = 0;
    for(;i<size;i++){
      variancia1Calc += pow(values[i]-mean,2.0f);
    }
    return variancia1Calc/(double)(size-1);
  }
  return -1;
}

int main(void){
  unsigned long int n = 0, n_var = 0;
  unsigned int i = 0, j =0;
  double **data;
/*  float mediaCalculada;*/

  scanf("%lu %lu",&n, &n_var);

  data = (double **)malloc(n_var*sizeof(double*));

  if(NULL == data)
    return EXIT_FAILURE;

  for(i = 0; i < n_var; i++){
```

```
42      data[i] = (double*)malloc(n*(sizeof(double)));
        if(NULL == data[i])
44        return EXIT_FAILURE;
    }
46
    for(i = 0; i < n; i++){
48    for( j = 0; j < n_var; j++){
        scanf("%lf", &data[j][i]);
50    }
    }
52
    for(i = 0; i < n_var; i++){
54    double dmean = mean(data[i],n);
      double var = variance(data[i], dmean, n);
56    printf("%lf\t%lf\n", dmean, var);
    }
58
    for(i = 0; i < n_var; i++)
60    free(data[i]);

62  free(data);

64  return EXIT_SUCCESS;
}
```

Código 2: Cálculo da média e variância dos dados.

Código para cálculo de PI, usando método de Monte Carlo, algoritmo sequencial

```
1  /* ------------------------------------------------------------------- */
   /*  Copyright (c) 2014 Fernando Noveletto Candiani, Marcius Leandro  */
3  /*  Junior, Rafael Hiroki de Figueiroa Minami                        */
   /*                                                                   */
5  /*  This program is free software; you can redistribute it and/or    */
   /*  modify it under the terms of the GNU General Public License as   */
7  /*  published by the Free Software Foundation; either version 3 of   */
   /*  the License, or (at your option) any later version.  See the     */
9  /*  file.                                                            */
   /*  LICENSE included with this distribution for more information.    */
11 /*  email: fncandiani, marcius, rafahiroki @usp.br                   */
   /* ------------------------------------------------------------------- */
13
#include <stdio.h>
15 #include <stdlib.h>
#include <time.h>
17
#define MAX_INTERACTION 1E+09
19
void montecarlo(unsigned long int num_interaction){
21    unsigned long int i = 0, hits = 0;;
  long double x = 0, y = 0;
23
  for(; i < num_interaction; i++){
25    x = drand48();        /*gera um valor aleatorio para x*/
      y = drand48();        /*gera um valor aleatorio para y*/
27
```

```
    if((x*x)+(y*y) < 1) /*verifica se o ponto aleatorio gerado esta dentro da
        circunferencia*/
29      hits++;

31  }

33  printf("%.61f\n", 4 * (hits/MAX_INTERACTION) );
}
35
int main(void){
37    srand48(time(NULL));  //semente para gerar os numeros aleatorios

39  montecarlo(MAX_INTERACTION); /*funcao que encontra o valor de pi utilizando o
        metodo de monte carlo*/

41  return EXIT_SUCCESS;
}
```

Código 3: Método de Monte Carlo sequencial.

Código para cálculo de PI, usando método de Monte Carlo, algoritmo usando paradigma de paralelismo

```
/* ------------------------------------------------------------------------ */
2 /*  Copyright (c) 2014 Fernando Noveletto Candiani, Marcius Leandro  */
  /*  Junior, Rafael Hiroki de Figueiroa Minami                        */
4 /*                                                                   */
  /*  This program is free software; you can redistribute it and/or    */
6 /*  modify it under the terms of the GNU General Public License as   */
  /*  published by the Free Software Foundation; either version 3 of   */
8 /*  the License, or (at your option) any later version.  See the     */
  /*  file.                                                            */
10 /*  LICENSE included with this distribution for more information.    */
  /*  email: fncandiani, marcius, rafahiroki @usp.br                   */
12 /* ------------------------------------------------------------------------ */

14 #include <pthread.h>
  #include <stdio.h>
16 #include <stdlib.h>
  #include <time.h>
18 #include <math.h>

20 #define MAX_INTERACTION (1E+09)

22 const int A = 1103515245, C = 12345, m = (1<<30);
  const long long T = (long long) m*m;
24 pthread_t *callThd;
  unsigned long long int *hits, num_interaction;
26
  int next(int& x) /*funcao que gera numeros aleatorios*/
28 {
    return x = (x*A+C)%m;
30 }

32 void *calculate_pi(void *arg) {

34   unsigned long long int i = 0;
```

```
  long offset;
  int x = time(NULL);

  offset = (long)arg;

  for (; i < num_interaction; i++) { /*Gera um ponto (x, y) aleatorio */
    long double a = next(x);
    long double b = next(x);

    if ((a * a)+(b * b) <= T) {  /*verifica se esse ponto esta dentro da
        circunferencia*/
      hits[offset]++;
    }
  }

  pthread_exit((void*) EXIT_SUCCESS);
}

int main(int argc, char** argv) {

  unsigned int num_threads;

  if(argc < 2)
    num_threads = 4;
  else
    num_threads = atoi(argv[1]);

  num_interaction = MAX_INTERACTION/num_threads;

  long i;
  long long int sum_hits=0;
  hits = (unsigned long long int *)malloc(sizeof(unsigned long long int)*num_threads
      );
  callThd = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);

  for (i = 0; i < num_threads; i++) {
    hits[i] = 0;
    pthread_create(&callThd[i], NULL, calculate_pi, (void *) i); /*cria as threads
        que executarao a funcao calculate_pi*/
  }

  for (i = 0; i < num_threads; i++) {
    pthread_join(callThd[i], NULL);   /*as threads terminam juntas*/
    sum_hits += hits[i];
  }

  printf("%.6lf\n", sum_hits * 4/MAX_INTERACTION);  /* pi = 4*soma dos pontos dentro
      da circunferencia/quantidade de pontos */

  free(hits);
  free(callThd);
  return (EXIT_SUCCESS);
}
```

Código 4: Método de Monte Carlo paralelo.

Código para cálculo de aproximação de valor confiável para aplicação de investimentos, usando método de Black-Scholes, algoritmo sequencial

```
/* ------------------------------------------------------------------- */
/*  Copyright (c) 2014 Fernando Noveletto Candiani, Marcius Leandro    */
/*  Junior, Rafael Hiroki de Figueiroa Minami                          */
/*                                                                     */
/*  This program is free software; you can redistribute it and/or      */
/*  modify it under the terms of the GNU General Public License as     */
/*  published by the Free Software Foundation; either version 3 of     */
/*  the License, or (at your option) any later version.  See the       */
/*  file.                                                              */
/*  LICENSE included with this distribution for more information.      */
/*  email: fncandiani, marcius, rafahiroki @usp.br                     */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define max(a,b) ({ \  /* retorna o maior entre a e b */
  typeof(a) _a_temp_; \
  typeof(b) _b_temp_; \
  _a_temp_ = (a); \
  _b_temp_ = (b); \
  _a_temp_ = _a_temp_ < _b_temp_ ? _b_temp_ : _a_temp_; \
})

double gaussrand()  /* gera numero randomico com distribuicao normal */
{
  static double V1, V2, S;
  static int phase = 0;
  double X;

  if(phase == 0) {
    do {
      double U1 = (double)rand() / RAND_MAX;
      double U2 = (double)rand() / RAND_MAX;

      V1 = 2 * U1 - 1;
      V2 = 2 * U2 - 1;
      S = V1 * V1 + V2 * V2;
      } while(S >= 1  S == 0);

    X = V1 * sqrt(-2 * log(S) / S);
  } else
    X = V2 * sqrt(-2 * log(S) / S);

  phase = 1 - phase;

  return X;
}

long double standard_deviation(long double *data, long int size, long double mean){
    /* desvio padrao */

  if(0 == size)
    return 0.0;

```

```
  long double mean_difference = 0.0, squared_deviation = 0.0;
57  long int i = 0;

59  for(; i < size; i++){
      mean_difference = data[i] - mean;
61    squared_deviation += mean_difference*mean_difference;
    }
63
/*  printf("squared_deviation: %Lf\n", squared_deviation);*/
65
    return sqrt(squared_deviation/size);
67}

69int main(void){

71  long double S = 0.0, E = 0.0, r = 0.0, sigma = 0.0, T = 0.0, *trials;
    long double sum = 0.0;
73  long int M = 0, i = 0;

75  scanf("%Lf", &S);        /*valor da acao  */
    scanf("%Lf", &E);        /* preco de exercicio da opcao */
77  scanf("%Lf", &r);        /* taxa de juros livre de risco */
    scanf("%Lf", &sigma);    /* volatilidade da acao */
79  scanf("%Lf", &T);        /* tempo de validade da opcao */
    scanf("%ld", &M);        /* numero de iteracoes */
81
    trials = (long double*) malloc(sizeof(long double)*M);
83
/*  printf("%Lf, %Lf, %Lf, %Lf, %Lf, %ld\n", S, E, r, sigma, T, M);*/
85        /* utiliza o metodo de monte carlo para calcular black scholes */
    for( ;i < M; i++){
87    long double rand = gaussrand();
      long double t = S * exp( ( (r - ((sigma*sigma)/2)) * T ) + (sigma*sqrt(T)*rand)
          );
89    trials[i] = exp( (-1*r)*T) * max(t-E, 0);
      sum += trials[i];
91  }
        /* calculo do intervalo de confianca */
93/*  printf("Sum: %Lf\n", sum);*/
    long double mean = sum / M;
95  long double confidence_interval = 1.96*(standard_deviation(trials, M, mean)/sqrt(M
      ));

97/*  printf("mean %Lf, ci %Lf\n", mean, confidence_interval);*/
/*  printf("The confidence interval calculated is [%Lf,%Lf]\n", mean -
    confidence_interval, mean + confidence_interval);*/
99
    printf("%.6Lf\n%.6Lf\n", mean, confidence_interval);
101
    free(trials);
103
    return EXIT_SUCCESS;
105}
```

Código 5: Método de Black-Scholes sequencial.

Código para cálculo de aproximação de valor confiável para aplicação de investimentos, usando método de Black-Scholes, algoritmo usando paradigma de paralelismo

```
/* ------------------------------------------------------------------- */
/*  Copyright (c) 2014 Fernando Noveletto Candiani, Marcius Leandro    */
/*  Junior, Rafael Hiroki de Figueiroa Minami                          */
/*                                                                     */
/*  This program is free software; you can redistribute it and/or      */
/*  modify it under the terms of the GNU General Public License as     */
/*  published by the Free Software Foundation; either version 3 of     */
/*  the License, or (at your option) any later version.  See the       */
/*  file.                                                              */
/*  LICENSE included with this distribution for more information.      */
/*  email: fncandiani, marcius, rafahiroki @usp.br                     */
/* ------------------------------------------------------------------- */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <pthread.h>

#define max(a,b) ({ \   /* retorna o maior entre a e b */
  typeof(a) _a_temp_; \
  typeof(b) _b_temp_; \
  _a_temp_ = (a); \
  _b_temp_ = (b); \
  _a_temp_ = _a_temp_ < _b_temp_ ? _b_temp_ : _a_temp_; \
})

const int A = 1103515245, C = 12345, m = (1<<30);
pthread_t *callThd;
unsigned long int num_threads, M = 0;
long double S = 0.0, E = 0.0, r = 0.0, sigma = 0.0, T = 0.0, mean = 0.0, *trials, *
    sum, *squared_deviation;

int next(int& x){   /* gera um numero aleatorio */
  x = (x*A+C)%m;
  if (x < 0)
    x = (x+m)%m;
  return x;
}

void* blackscholes(void *arg){    /* Modelo de Black Scholes atraves da simulacao de
     Monte Carlo */

  long long int offset, size = M/num_threads;
  int x = time(NULL);

  offset = (unsigned long int) arg;

  unsigned long long int i = 0;
  unsigned long long int begin = offset*size;
  unsigned long long int end = (offset+1)*size;

  for(i = begin; i < end; i++){
    double t = S * exp((r - 0.5*sigma*sigma) * T + sigma * sqrt(T) * next(x) / m);
        trials[i] = exp(-r * T) * max(t - E, 0);
    sum[offset] += trials[i];
```

```
55    }
57    pthread_exit((void*) EXIT_SUCCESS);
}
59
void *standard_deviation(void* arg){    /* calculo do desvio padrao */
61
    long long int offset, size = M/num_threads;
63
    if(0 == size)
65      pthread_exit((void*) EXIT_SUCCESS);
67    offset = (long long int )arg;
69    long double mean_difference = 0.0;
    unsigned long long int i = 0;
71    unsigned long long int begin = offset*size;
    unsigned long long int end = (offset+1)*size;
73
    for(i = begin; i < end; i++){
75      mean_difference = trials[i] - mean;
      squared_deviation[offset] += mean_difference*mean_difference;
77    }
79    pthread_exit((void*) EXIT_SUCCESS);
}
81
int main(int argc, char** argv){
83
85    if(argc < 2)
      num_threads = 4;
87    else
      num_threads = atoi(argv[1]);
89
    scanf("%Lf", &S);         /*valor da acao  */
91    scanf("%Lf", &E);         /* preco de exercicio da opcao */
    scanf("%Lf", &r);         /* taxa de juros livre de risco */
93    scanf("%Lf", &sigma);     /* volatilidade da acao */
    scanf("%Lf", &T);         /* tempo de validade da opcao */
95    scanf("%ld", &M);         /* numero de iteracoes */
97    unsigned long int i;
    long double sum_hits = 0.0;
99    long double std_deviation = 0.0;
101   trials = (long double*) malloc(sizeof(long double)*M);
    sum = (long double*) malloc(sizeof(long double)*num_threads);
103   squared_deviation = (long double*)malloc(sizeof(long double)*num_threads);
    callThd = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);
105
/*  printf("%Lf, %Lf, %Lf, %Lf, %Lf, %ld\n", S, E, r, sigma, T, M);*/
107
    for (i = 0; i < num_threads; i++) {      /* cria as threads para a funcao
        blackscholes */
109     pthread_create(&callThd[i], NULL, blackscholes, (void *) i);
    }
111
    for (i = 0; i < num_threads; i++) {
```

```
113                 pthread_join(callThd[i], NULL);      /* as threads devem terminar
                        juntas */
      sum_hits += sum[i];
115   }

117   mean = sum_hits / M;

119   for (i = 0; i < num_threads; i++) {      /* cria as threads para a funcao
          standard_deviation */
      pthread_create(&callThd[i], NULL, standard_deviation, (void *) i);
121   }

123   for (i = 0; i < num_threads; i++) {
      pthread_join(callThd[i], NULL);      /* as threads devem terminar juntas */
125   std_deviation += squared_deviation[i];
      }
127
      long double confidence_interval = 1.96*(sqrt(std_deviation/M)/sqrt(M));
129
/*    printf("The confidence interval calculated is [%Lf,%Lf]\n", mean -
      confidence_interval, mean + confidence_interval);*/
131
      printf("%.6Lf\n%.6Lf\n", mean, confidence_interval);
133
      free(trials);
135   free(sum);
      free(squared_deviation);
137   free(callThd);

139   return EXIT_SUCCESS;
}
```

Código 6: Método de Black-Scholes paralelo.

Referências

[1] Black, Fischer; Myron Scholes. (1973). *The Pricing of Options and Corporate Liabilities*. Journal of Political Economy **81** (Black and Scholes' original paper.)

[2] Merton, Robert C.. (1973). *Theory of Rational Option Pricing*. Bell Journal of Economics and Management Science **4**