Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

SSC0143 – Programação Concorrente

# Trabalho 1 – Cálculo do $\pi$ e Black-Scholes

Elias Italiano Rodrigues – 7987251
Rodolfo Megiato de Lima – 7987286
Vinicius Katata Biondo – 6783972

São Carlos, 5 de setembro de 2014

Repositório: https://code.google.com/p/pc2014-02-grupo2-turmab/

# Sumário

# 1 Introdução

## 1.1 Cálculo do $\pi$? Black-Scholes?

Este trabalho implementa algoritmos conhecidos para o cálculo de muitas casas decimais do número $\pi$. O objetivo com isso não é obter uma precisão gigantesca e inútil do número $\pi$, mas sim aplicar conhecimentos da disciplina de Programação Concorrente nos algoritmos implementados. Para isso é feita a implementação sequencial e paralela (*multi-thread*) de cada algoritmo, e são eles: Gauss-Legendre [1], Borwein (1984) [2] e Método de Monte Carlo [3].

Além disso, este trabalho também implementa as versões sequencial e paralela da Simulação Monte Carlo para o algoritmo de Black-Scholes [4] que descreve um fenômeno financeiro que é a precificação de derivativos. O algoritmo implementado é o descrito na especificação do trabalho.

Este relatório faz parte de um conjunto de arquivos referentes ao trabalho. A documentação do código-fonte encontra-se nos próprios arquivos `.c` referentes aos algoritmos dentro do diretório `./src` a partir do diretório raiz.

## 1.2 Os Algoritmos

### 1.2.1 Gauss-Legendre

O algoritmo de Guass-Legendre é um algoritmo para computar os dígitos do $\pi$ e é notável por sua convergência quadrática.

1. Define-se os valores iniciais para as variáveis:

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

2. Repete-se os seguintes passos até que a diferença entre $a_n$ e $b_n$ esteja dentro da precisão desejada:

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

$$b_{n+1} = \sqrt{a_n \cdot b_n} \tag{2}$$

$$t_{n+1} = t_n - p_n \cdot (a_n - a_{n+1})^2 \tag{3}$$

$$p_{n+1} = 2 \cdot p_n \tag{4}$$

3. Obtém-se $\pi$ aproximando por:

$$\pi \approx \frac{(a_{n+1} + b_{n+1})^2}{4 \cdot t_{n+1}} \tag{5}$$

**Algoritmo 1:** Gauss-Legendre

### 1.2.2 Borwein (1984)

Este algoritmo de Borwein é um dos usados para se calcular dígitos do $\pi$ e também possui convergência quadrática.

1. Define-se os valores iniciais para as variáveis:

$$a_0 = \sqrt{2} \quad , \quad b_0 = 0 \quad , \quad p_0 = 2 + \sqrt{2}$$

2. Repete-se os seguintes passos até que $p_n$ aproxime $\pi$ na quantidade de dígitos desejada:

$$a_{n+1} = \frac{a_n + 1}{2 \cdot \sqrt{a_n}} \tag{6}$$

$$b_{n+1} = \frac{(1 + b_n) \cdot \sqrt{a_n}}{a_n + b_n} \tag{7}$$

$$p_{n+1} = \frac{(1 + a_{n+1}) \cdot p_n \cdot b_{n+1}}{1 + b_{n+1}} \tag{8}$$

3. Obtém-se $\pi$ aproximando por:

$$\pi \approx p_n \tag{9}$$

**Algoritmo 2:** Borwein (1984)

### 1.2.3 Método de Monte Carlo

O Método de Monte Carlo para o cálculo $\pi$ consiste em:

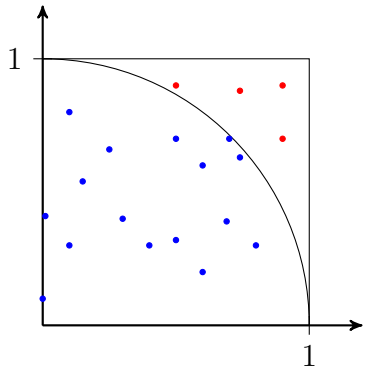


Figura 1: Ilustração do algoritmo para cálculo do $\pi$ com Método Monte Carlo.

1. Toma-se um quadrado de lado $l = 1$ e dentro dele um setor de 90° de raio $r = 1$.
2. Gera-se aleatoriamente $N$ pontos $(x, y) \in [0, 1]$
3. Então, para cada um dos $N$ pontos, confere se:

$$x^2 + y^2 \leq 1$$

4. Conta-se em $C$ quantos pontos satisfazem a condição acima
5. Obtém-se $\pi$ aproximando por:

$$\pi \approx \frac{4 \cdot C}{N}$$

**Algoritmo 3:** Método Monte Carlo para $\pi$

### 1.2.4 Black-Scholes

Foi elaborado por dois cientistas chamados Fisher Black e Myron Scholes, que adaptaram uma fórmula física para descrever um fenômeno financeiro que é a precificação de derivativos.

$S \leftarrow$ valor da ação
$E \leftarrow$ preço de exercício da opção
$r \leftarrow$ taxa de juros livre de risco (SELIC)
$\sigma \leftarrow$ volatilidade da ação
$T \leftarrow$ tempo de validade da opção
$M \leftarrow$ número de iterações
**for** $i \leftarrow 0$ **to** $M - 1$ **do**
  $t \leftarrow S \cdot \texttt{exp}((r - 0.5 \cdot \sigma^2) \cdot T + \sigma\sqrt{T} \cdot \texttt{randomNumber}())$
  $trials[i] \leftarrow \texttt{exp}(-r \cdot T) \cdot \texttt{max}(t - E, 0)$
**end**
$mean \leftarrow \texttt{mean}(trials)$
$stddev \leftarrow \texttt{stddev}(trials, mean)$
$confwidth \leftarrow 1.96 \cdot stddev/\sqrt{M}$
$confmin \leftarrow mean - confwidth$
$confmax \leftarrow mean + confwidth$

**Algoritmo 4:** Black-Scholes

# 2 Desenvolvimento

Os programas foram desenvolvidos em linguagem `C` em sistema operacional `GNU/Linux`. Além das bibliotecas convencionais `stdlib.h` e `stdio.h`, foi utilizada também a `gmp.h` [5] para usar tipos de dados numéricos com precisão arbitrária. A implementação de *threads* foi feita com a biblioteca `pthread.h` seguindo o modelo visto em sala de aula.

O tipo de dado usado para representar os números reais foi `mpf_t` que é próprio da biblioteca `gmp.h`, assim como as respectivas funções para manipulá-lo. De modo geral, optou-se por economizar na quantidade de variáveis e na quantidade de operações dentro dos laços.

## 2.1 Gauss-Legendre – Sequencial

A implementação sequencial de Gauss-Legendre foi feita de modo direito, "traduzindo-se"o algoritmo da linguagem matemática para a linguagem `C` como pode ser conferido nas Listagens 1 e 2 referentes às Equações 1, 2, 3, 4 e 5.

Listagem 1: Gauss-Legendre – Sequencial: trecho com variáveis.

```
mpf_t a0; // a_{n}
mpf_t a1; // a_{n+1}
mpf_t b;  // b_{n} e b_{n+1}
mpf_t t;  // t_{n} e t_{n+1}
mpf_t p;  // p_{n} e p_{n+1}
mpf_t tmp; // auxiliar
```

Listagem 2: Gauss-Legendre – Sequencial: trecho com equações.

```
// Iteracao do algoritmo
for (i = N; i > 0; i--) {
    // Primeira equacao
    mpf_add(a1, a0, b);
    mpf_div_ui(a1, a1, 2);
    // Segunda equacao
    mpf_mul(b, a0, b);
    mpf_sqrt(b, b);
    // Terceira equacao
    mpf_sub(tmp, a0, a1);
    mpf_pow_ui(tmp, tmp, 2);
    mpf_mul(tmp, tmp, p);
    mpf_sub(t, t, tmp);
    // Quarta equacao
    mpf_mul_ui(p, p, 2);

    mpf_set(a0, a1);
}
// Resultado (quinta equacao)
mpf_add(tmp, a1, b);
mpf_pow_ui(tmp, tmp, 2);
mpf_div_ui(tmp, tmp, 4);
mpf_div(tmp, tmp, t);
```

## 2.2 Gauss-Legendre – Paralelo

A implementação paralela do algoritmo consiste em identificar operações artiméticas que possam ser realizadas independentes, alocá-las em tarefas $T_i$ e então atribuí-las em processos $P_i$ que representam *threads*. Para isso, as equações referentes ao algoritmo foram analisadas e criou-se o seguinte **grafo de dependêcia de tarefas** da Figura 2. Importante observar que foram abertos os quadrados para se obter maior quantidade de termos independentes, por exemplo: $(a_n - a_{n+1})^2 = a_n^2 - 2a_n a_{n+1} + a_{n+1}^2$.

O código-fonte foi escrito segundo esse grafo, obtendo-se **grau de concorrência 5**. Confira a listagem completa do código do programa no arquivo `./src/gauss-legendre_pthread.c`.

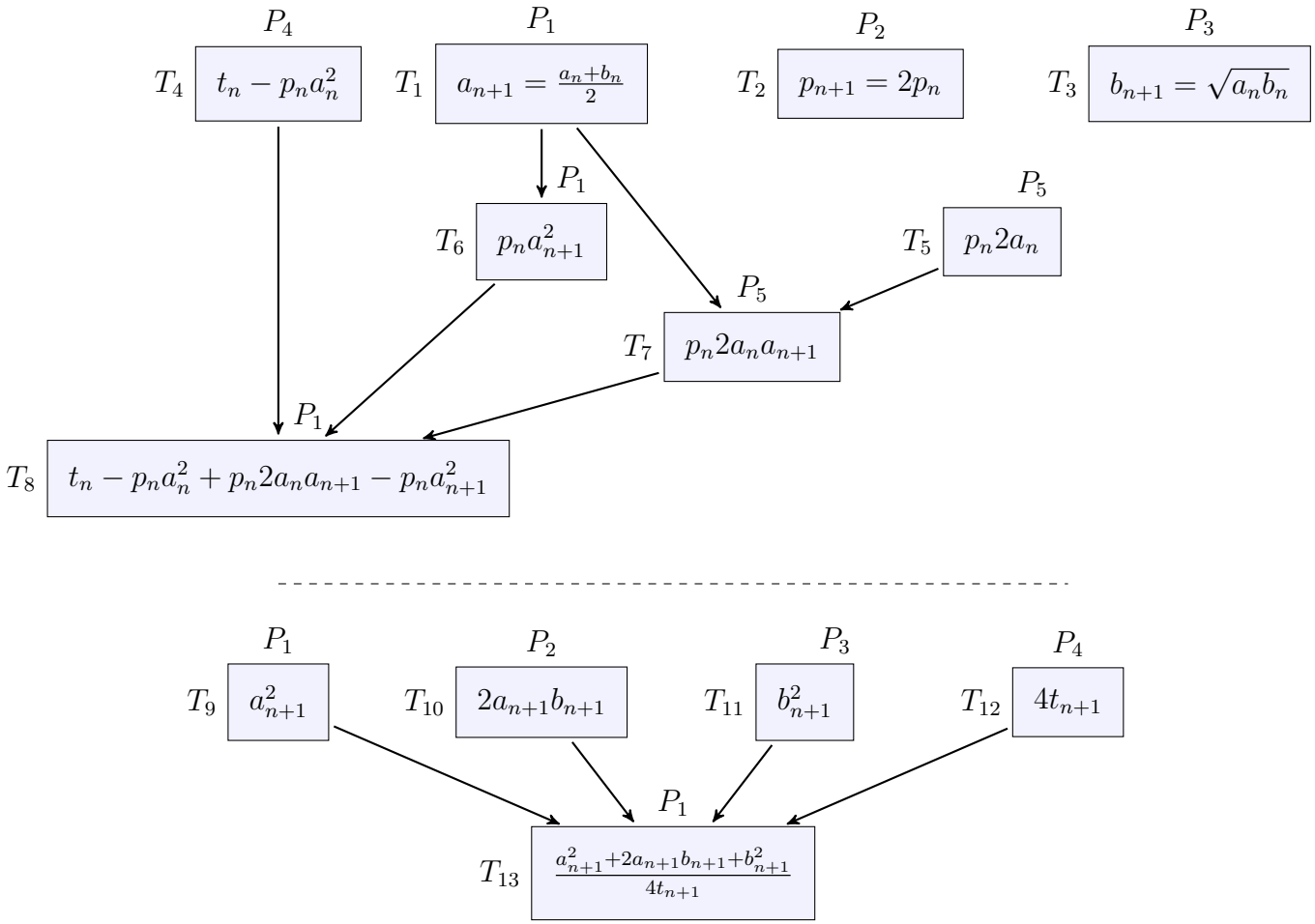


Figura 2: Grafo de dependência de tarefas para o algoritmo de Gauss-Legendre.

## 2.3 Borwein (1984) – Sequencial

A implementação sequencial de Borwein (1984) também foi feita de modo direito, "traduzindo-se"as Equações 6, 7, 8 e 9 da linguagem matemática para a linguagem `C`. As variáveis foram definidas de modo análogo à implementação de Guass-Legendre e também foi possível traduzir diretamente as equações para o programa.

## 2.4 Borwein (1984) – Paralelo

Assim como a implementação paralela do algoritmo de Guass-Legendre, para este algoritmo de Borwein (1984) também foram identificadas as operações artiméticas que podem ser realizadas independentemente. Pode-se conferir o **grafo de dependêcia de tarefas** da Figura 3.

O código-fonte foi escrito segundo esse grafo, obtendo-se **grau de concorrência 4**. Confira a listagem completa do código do programa no arquivo `./src/borwein_pthread.c`.

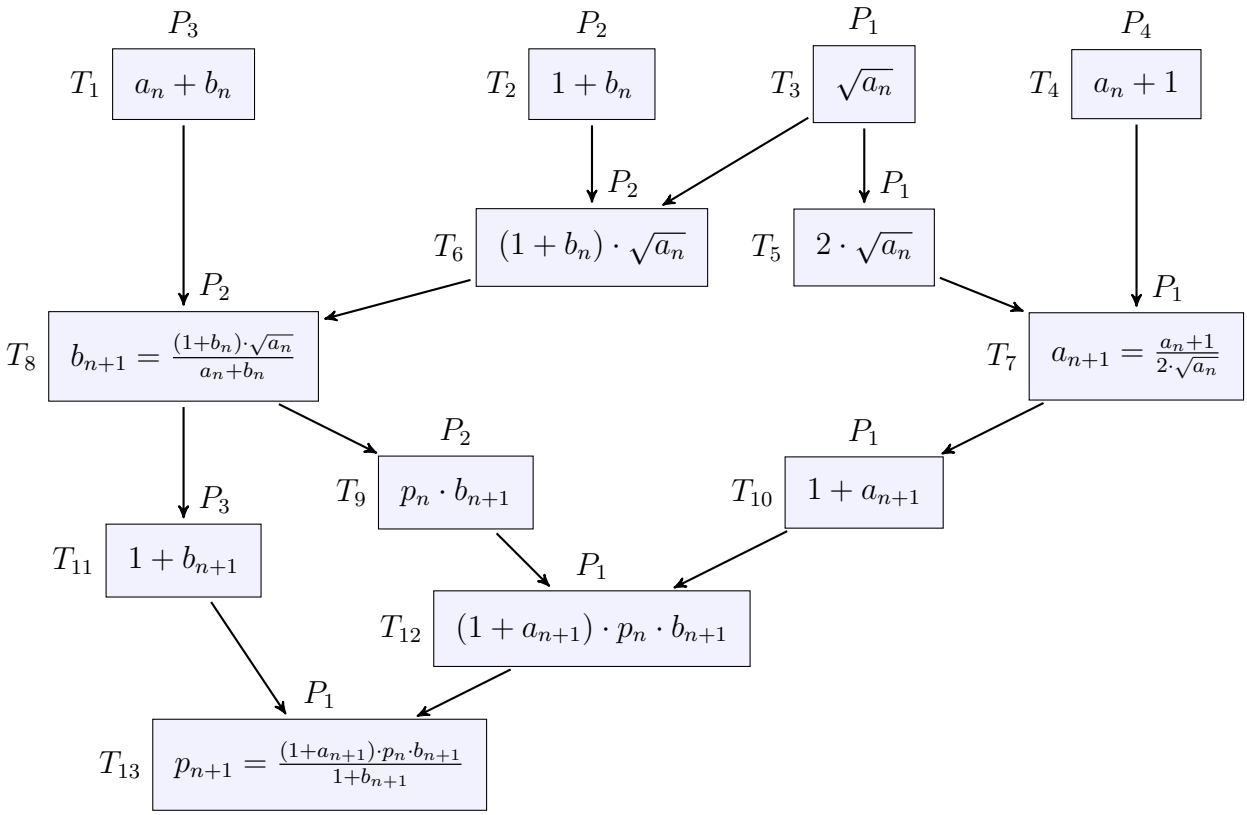


Figura 3: Grafo de dependência de tarefas para o algoritmo de Borwein (1984).

## 2.5 Método de Monte Carlo – Sequencial

A implementação deste método utilizou o código para geração de números aleatórios dado na especificação do trabalho. Não houve necessidade de usar *big numbers* uma vez que o algoritmo dificilmente converge para uma precisão $d = 6$ do número $\pi$ mesmo com $N = 10^9$ iterações. A forma sequencial é feita de maneira direita em uma iteração, da seguinte forma:

Listagem 3: Método Monte Carlo para $\pi$ – Sequencial: trecho de código

```
// Iteracao do algoritmo
for (i = N; i > 0; i--) {
    randomx = boxMullerRandom(&random);
    randomy = boxMullerRandom(&random);
    if ((randomx*randomx + randomy*randomx) <= 1.0) {
        circleArea++;
    }
}
// Resultado
printf("%.8lf\n", 4.0 * ((double)circleArea / (double)N));
```

## 2.6 Método de Monte Carlo – Paralelo

Como na iteração do algoritmo não há dependência entre os dados, podemos paralelizar essa iteração atribuindo um quantidade $n = N/nthreads$, onde $nthreads$ é quantidade de *threads* desejada. Tendo feito os ajustes para a alocação das *threads*, o programa para esta versão paralela fica da seguinte forma:

Listagem 4: Método Monte Carlo para $\pi$ – Paralelo: trecho de código

```
    // Calcula quantidade de iteracao por thread
    n = N / nthreads;
...
    // Iteracao do algoritmo em todas as threads
    for (i = 0; i < nthreads; i++) {
        indices[i] = i;
        sums[i] = 0;
        pthread_create(&threads[i], NULL, func, &indices[i]);
    }

    // Espera todas threads terminarem e soma os valores
    for (i = 0; i < nthreads; i++) {
        pthread_join(threads[i], NULL);
        circleArea += sums[i];
    }
...
    // Resultado
    printf("%.8lf\n", 4.0 * ((double)circleArea / (double)N));
```

## 2.7 Black-Scholes – Sequencial

Assim como o Método Monte Carlo para o cálculo do $\pi$, para este algoritmo de Black-Scholes também foi utilizado o código para geração de números aleatórios dado na especificação do trabalho. O algoritmo consiste na leitura de entradas, um laço e o cálculo dos valores necessários para retornar o intervalo de confiança. Tal implementação para linguagem `C` "traduz"diretamente do Algoritmo 4.

## 2.8 Black-Scholes – Paralelo

Como também não há dependência entre os dados nesse algoritmo, podemos paralelizar essa iteração atribuindo um quantidade $m = M/nthreads$, onde $nthreads$ é quantidade de *threads* desejada. Tendo feito os ajustes para a alocação das *threads*, a ideia do programa para a versão paralela fica análogo ao programa paralelo de Monte Carlo para $\pi$.

## 2.9 Compilação e Execução dos Programas

Estando no diretório raiz dos arquivos deste trabalho, é possível compilar e executar os programas seguindo as instruções:

- Para compilar os programas, execute o comando:
  `make`
- Para rodar um dos programas calculando o tempo de execução e redirecionando a entrada e a saída para os arquivos `entrada_pi.txt` e `saida_pi.txt`, execute o comando:
  `make run EXE=nome-do-programa`

- Para rodar Black-Scholes com entrada e saída de `entrada_blackscholes.txt` e `saida_blackscholes.txt`, execute o comando:
  ```
  make run-bs  # para versao sequencial
  make run-bsp # para versao paralela
  ```

- Para limpar todos os arquivos compilados, execute o comando:
  ```
  make clean
  ```

Caso não consiga compilar e rodar os programas, confira em seu sistema operacional por dependências das bibliotecas `gmp.h`, `math.h` e `pthread.h` assim como dos programas usados `make`, `gcc` e `valgrind`.

# 3 Resultados

Os programas foram compilados e executados de modo automatizado por *shell scripts* no computador de um dos integrantes do grupo do trabalho. Seguem as especificações desse computador:

```
Phoronix Test Suite v5.2.1
System Information

Hardware:
Processor: Intel Core i7-3612QM @ 3.10GHz (8 Cores),
Motherboard: Dell 0DNMM8, Chipset: Intel 3rd Gen Core DRAM, Memory: 8192MB,
Graphics: Intel HD 4000 2048MB (1100MHz)

Software:
OS: Fedora 20, Kernel: 3.15.10-201.fc20.x86_64 (x86_64),
Compiler: GCC 4.8.3 20140624, File-System: ext4
```

## 3.1 Gauss-Legendre *vs* Borwein (1984)

A especificação do trabalho pede para que os programas retornem o número $\pi$ com precisão $d = 6$ casas corretas considerando um quantidade de iterações $N = 10^9$. Porém, para os algoritmos de Gauss-Legendre e Borwein (1984), essa precisão do $\pi$ é alcançada rapidamente com a apenas $N = 2$ iterações. Diante disso, com o objetivo de "estressar"as implementações desses algoritmos, foram criados casos de testes para alcançar precisão $d = 10^i, i = 4, 5, 6, 7$.

Para conferir a corretude dos dígitos calculados, eles foram comparados com os dígitos gerados pelo programa `pi` da CLN[7] da seguinte maneira: executou-se `pi` para as precisões $d = 10^i, i = 4, 5, 6, 7$ redirecionando a saída para arquivos nomeados de acordo com $d$, e então gerou-se uma lista `md5sum` desses arquivos à qual as saídas deste trabalho foram comparadas.

Como podemos observar a partir de uma análise do *speedup* de cada algoritmo, a implementação paralela torna-se mais rápida conforme a precisão requerida do $\pi$ aumenta. Porém, tal ganho do programa paralelo mostra-se ser pouco significativo se considerarmos a quantidade de *threads* usadas – 5 para Gauss-Legendre e 4 para Borwein (1984).

| Precisão | GL | GLP | *Speedup* GL | B | BP | *Speedup* B |
|---|---|---|---|---|---|---|
| $10^4$ | 0.05 | 0.03 | 1.66 | 0.09 | 0.09 | 1.00 |
| $10^5$ | 0.89 | 0.82 | 1.09 | 2.26 | 2.38 | 0.95 |
| $10^6$ | 21.53 | 20.79 | 1.04 | 54.36 | 48.58 | 1.12 |
| $10^7$ | 365.77 | 330.42 | 1.1 | 938.84 | 765.62 | 1.23 |

Tabela 1: Tempos (s) de execução e *speed-up* dos algoritmos Gauss-Legendre e Borwein (1984)

GL: Gauss-Legendre Sequencial / GLP: Gauss-Legendre Paralelo
B: Borwein (1984) Sequencial / BP: Borwein (1984) Paralelo

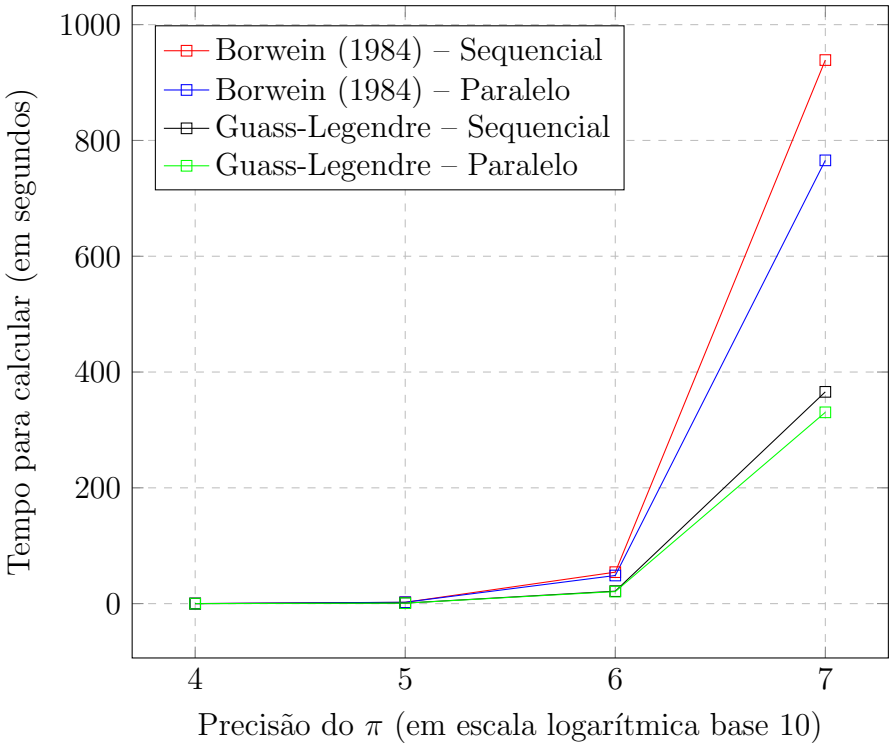


Figura 4: Comparativo dos dados do cálculo do $\pi$ com Gauss-Legendre e Borwein (1984).

## 3.2 Método de Monte Carlo

Os testes do Monte Carlo para cálculo do $\pi$ foram feitos com uma quantidade fixa de $N = 10^9$ iterações, mas com uma quantidade váriavel de *threads* o que nos possibilitou conferir o tempo de execução do programa de acordo com a quantidade de *threads* utilizada. Pelos resultados da Tabela 2 pode-se perceber que o tempo de execução diminuiu conforme o aumento da quantidade de *threads*, porém não de maneira proporcional.

## 3.3 Black-Scholes

Análogo aos testes do Método de Monte Carlo para $\pi$, os testes para Black-Scholes também foram feitos com uma quantidade variável de *threads* e com uma entrada fixa do arquivo `entrada_blackscholes.txt` que informa uma quantidade $M = 10^9$ iterações. Pelos resultados da Tabela 3 pode-se perceber que o tempo de execução diminuiu conforme o aumento da quantidade de *threads*, porém também não de maneira proporcional.

| nthreads | MC | MCP | *Speedup* | $\approx \pi$ |
|---|---|---|---|---|
| – | 29.68 | - | - | 3.16098686 |
| 2 | - | 22.31 | 1.33 | 3.14160214 |
| 4 | - | 20.61 | 1.44 | 3.14156532 |
| 8 | - | 15.51 | 1.92 | 3.14165089 |

Tabela 2: Tempos (s) de execução e *speed-up* do Método de Monte Carlo para $\pi$

MC: Monte Carlo Sequencial
MCP: Monte Carlo Paralelo

| nthreads | BS | BSP | *Speedup* | intconf |
|---|---|---|---|---|
| – | 115.95 | - | - | [99.986946, 99.986947] |
| 2 | - | 74.67 | 1.55 | [99.989184, 99.989185] |
| 4 | - | 59.66 | 1.94 | [99.983979, 99.983980] |
| 8 | - | 47.62 | 2.43 | [100.000951, 100.000951] |

Tabela 3: Tempos (s) de execução e *speed-up* do Black-Scholes

BS: Black-Scholes Sequencial
BSP: Black-Scholes Paralelo
intconf: intervalo de confiança

# 4 Conclusão

A paralelização nem sempre pode levar a resultados melhores, pois existem vários outros detalhes a serem considerados, como por exemplo a quantidade de núcleos da CPU, quantos e quais processos estão em execução no sistema operacional etc.

Neste trabalho, vimos com Gauss-Legendre e Borwein (1984) que o ganho final foi pouco significativo apesar desses dois algoritmos terem sidos paralelizados com grau de concorrência 4 e 5, respectivamente. Por outro lado, o ganho com a paralelização do Método de Monte Carlo para $\pi$ e do Black-Scholes mostrou-se significativo, porém não ao ponto de ser proporcional ao número de *threads* utilizadas.

A produção desde trabalho nos introduziu às bibliotecas `gmp.h` e `pthread.h`, a pensar em como "quebrar"um algoritmo de modo a codificá-lo em *threads* e à técnica de decomposição usando grafo de dependência de tarefas segundo o livro-texto [8] da disciplina. Isso é um retorno positivo para nós.

# Referências


[1] http://en.wikipedia.org/wiki/Gauss-Legendre_algorithm

[2] http://en.wikipedia.org/wiki/Borwein's_algorithm

[3] http://en.wikipedia.org/wiki/Monte_Carlo_method

[4] http://pt.wikipedia.org/wiki/Black-Scholes

[5] GNU Multiple Precision Arithmetic Library
https://gmplib.org/

[6] LaSDPC – Laboratório de Sistemas Distribuídos e Programação Concorrente
http://lasdpc.icmc.usp.br/
Cluster do LaSDPC
http://cluster.lasdpc.icmc.usp.br/

[7] CLN – Class Library for Numbers
http://www.ginac.de/CLN/
Debian `pi` package
https://packages.debian.org/stable/math/pi

[8] Introduction to parallel computing / Ananth Grama [et al] – 2nd ed.