UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

SSC0143
Programação Concorrente - Turma B

Algoritmos para realizar o cálculo aproximado do número π

Grupo:
Bruno Junqueira Adami
Lucas Junqueira Adami
Lucas Lobosque

Professor Dr.:
Julio Cezar Estrella

24 de março de 2012

1 Introdução

1.1 Algoritmo de Gauss-Legendre

O algoritmo de Gauss-Legendre, baseado no trabalho individual de Carl Friedrich Gauss e Adrien-Marie Legendre, é um algoritmo para calcular os dígitos de π. Sua característica principal é sua convergência rápida, de segunda ordem, onde 24 iterações produzem 10 milhões de dígitos corretos de π. Ele segue os seguintes passos:

$$a_0 = 1$$

$$b_0 = 1/\sqrt{2}$$

$$t_0 = 1/4$$

$$p_0 = 1$$

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

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

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

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

1.2 Algoritmo de Borwein

O algoritmo de Borwein, criado pelos matemáticos Jonathan e Peter Borwein, é usado para calcular o valor de π, com a precisão aumentando a cada iteração. Existem várias versões do algoritmo. A versão utilizada no trabalho quadruplica o número de dígitos corretos a cada iteração, gerando 10 milhões de dígitos em apenas 12 iterações. Ela segue os seguintes passos:

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

$$y_0 = \sqrt{2} - 1$$

$$y_{k+1} = \frac{1 - (1 - y_k^4)^{\frac{1}{4}}}{1 + (1 - y_k^4)^{\frac{1}{4}}}$$

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

1.3 Algoritmo de Monte Carlo

O algoritmo de Monte Carlo é um método estatístico para calcular funções complexas de um modo aproximado. Este método tipicamente envolve a geração de observações de alguma distribuição de probabilidades e o uso da amostra obtida para aproximar a função de interesse. As aplicações mais comuns são em computação numérica para avaliar integrais. A ideia do método é escrever a integral que se deseja calcular como um valor esperado. Foi utilizado este método para calcular a área de 1/4 do círculo. A partir deste valor, é possível estimar o valor do π. O número de iterações do algoritmo determina a precisão do cálculo, cada casa decimal a mais multiplica um fator de 10x na quantidade de iterações necessárias para tal. Mesmo tendo várias casas decimais, isso não implica que as casas estão corretas. Além disso, o raio pode ser um valor arbitrário. Foi utilizado como raio o maior número aleatório possível.

$$Area = \frac{\pi R^2}{4}$$

$$\pi = 4\frac{Area}{R^2}$$

$$Area = \frac{PontosDentrodoCirculo}{PontosTotais}$$

$$\pi = 4\frac{PontosDentrodoCirculo}{PontosTotais}$$

```
1  pontosDentro = 0
2  pontosTotais = 0
3  M = 1000000000 // 10^9: 9 casas no resultado
4
5  pontosTotais = M
6
7  fazer M iteracoes
8          a = rand()
9          b = rand()
10         se sqrt(a*a + b*b) <= r
11                 ++pontosDentro
12
13 retornar 4*pontosDentro/pontosTotais
```

Código 1: Algoritmo de Monte Carlo

2 Soluções

Para os algoritmos de Gauss-Legendre e Borwein, foi utilizada a biblioteca GNU MP, uma biblioteca open-source para as linguagens C e C++ que é capaz de criar números com infinitas casas decimais. Ela foi usada porque as variáveis do tipo double não alcançam essa precisão. Para o algoritmo do Monte Carlo, não foi utilizada nenhuma biblioteca de números grandes, pois iria ser necessário mais do que 10^{19} iterações para justificar o uso.

Números aleatórios no algoritmo de Monte Carlo: No C é possível usar a função rand() da stdlib para gerar os números aleatórios. Porém, segundo o padrão do C, a função não é thread-safe, ou seja, não há garantias quanto ao uso desta função em várias threads. Investigando mais a fundo, a implementação da função rand() pela libc usa locks de contexto, isto é, apenas uma thread por vez pode chamar a função rand(). Isso iria aumentar drasticamente o tempo do algoritmo, funcionando como algo serial. Então, foi implementado um gerador congruente linear, usando parâmetros parecidos com o mesmo usado pelo gcc, de tal modo que o gerador seja uniforme.

```
1 X(n+1) = (A*X(n)+C) % M
2 X(0) = time(NULL) // numero inicial (seed) de acordo com o tempo atual..
3 A = 1103515245
4 C = 12345
5 M = 2^31
```

Código 2: Algoritmo do gerador congruente linear

O maior número aleatório gerado é $2^{31} - 1$ (cabe em um inteiro de 32 bits com sinal, int em C no gcc). Este valor foi usado para a conta logo abaixo caber num inteiro de 64 bits com sinal (long long em C no gcc), Para não precisar usar o módulo no gerador, os bits são apenas truncados nas contas, deixando mais rápido os cáculos, por isso o módulo é 2^{31}. Para verificar se o ponto está dentro do círculo não é preciso usar pontos flutuantes ou raiz quadrada, deixando os cálculos mais rápidos e precisos:

$$R = \sqrt{a^2 + b^2}$$

$$R^2 = aa + bb$$

2.1 Método sequencial

O algoritmo de Monte Carlo foi implementado seguindo os passos descritos anteriormente. O número que é impresso na tela é o π tirando a vírgula (ex: 3,1415 = 31415). Nos algoritmos de Gauss-Legendre e Borwein, que são iterativos, variáveis auxiliares (com sufixo _) foram utilizadas para guardar os resultados da iteração anterior. O valor de π é mostrado em cada iteração.

2.2 Método paralelo

No método paralelo, foi utilizada a biblioteca pthreads (POSIX Threads) para a criação das threads do programa. No algoritmo de Monte Carlo, foram criadas N threads. Cada thread é responsável por gerar M' pontos. Se multiplicarmos M' por N teremos o M utilizado no método sequencial, ou seja, cada thread gera um pouco dos M pontos aleatórios. No fim tudo é somado obtendo o número esperado. Observe que as threads são independentes, não precisando de mutex ou sincronizações. No algoritmo de Borwein, foram utilizadas duas threads, uma para calcular y_{k+1} e outra para calcular a_{k+1}. No algoritmo de Gauss-Lagrange, foram criadas três threads, uma para calcular a_{n+1} e t_{n+1}, outra para calcular b_{n+1} e a última para calcular p_{n+1}.

3 Resultados e conclusões

Nos experimentos, o comando time do Linux, que conta o tempo de execução de um processo, foi utilizado. Além disso, os comandos de impressão dos dígitos de π foram removidos para que apenas o tempo de processamento fosse computado. Para o algoritmo de Monte Carlo, foram testados várias quantidades de threads, com a otimização O3 e sem. Para os outros dois algoritmos, as otimizações O3, O2, O1 e sem otimização foram utilizadas. Além disso, para os algoritmos que utilizaram a GNU MP, a precisão passada à biblioteca foi 3.32 maior que 10 milhões, pois este número não é necessariamente a precisão final. Portanto, o número de iterações nesses casos foi um pouco maior. Esse valor foi calculado empiricamente.

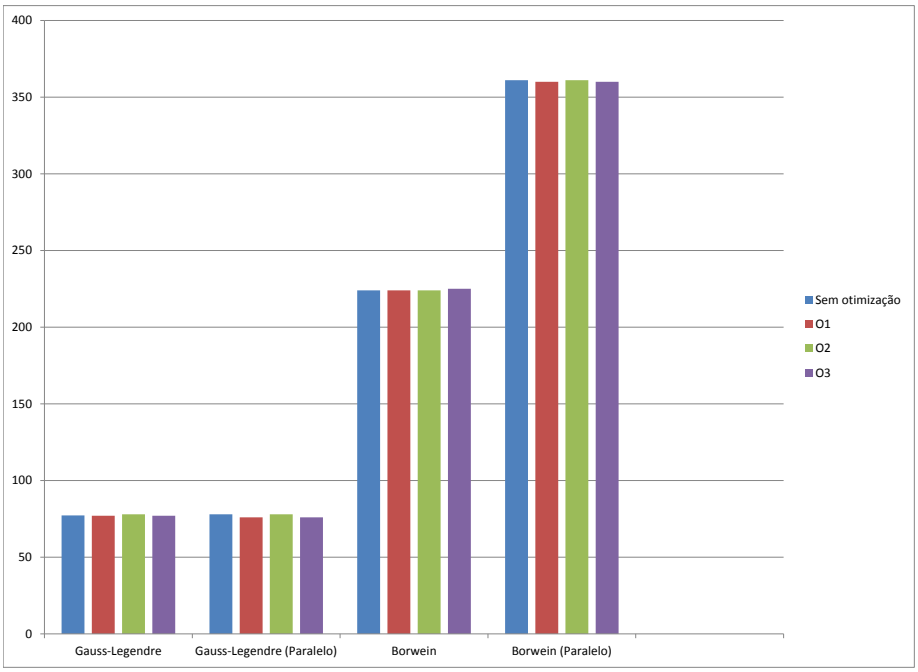

Figura 1: Resultados dos algoritmos de Gauss-Legendre e Borwein

Observando os resultados nas figuras de 1 a 3, para o método de Monte Carlo, é possível concluir que o número ideal de threads depende da quantidade de núcleos/unidades de processamento do computador utilizado. Se este número é pequeno, não será utilizado toda a capacidade de processamento, se o número é grande, o overhead do escalonamento do sistema operacional para alocar o processamento de cada thread (no caso número de threads » unidades de processamento) será muito grande, deixando lento o método. Para os outros méto-

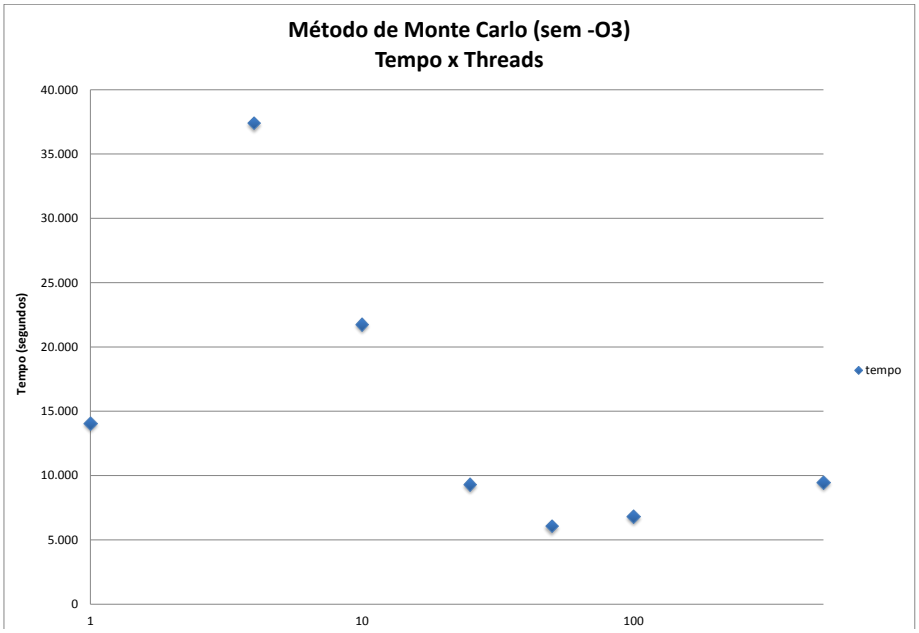

Figura 2: Resultados do algoritmo de Monte Carlo sem otimização

dos, esse fato também pode ser considerado. Os algoritmos de Gauss-Legendre e Borwein, como são iterativos, tiveram apenas um pequeno ganho de tempo. Isto é, cada iteração depende da anterior, fazendo com que a abordagem paralela não seja muito chamativa. Em especial, no algoritmo de Borwein, a implementação não é eficiente, já que para calcular a_{k+1}, é necessário esperar o resultado de y_{k+1}. No algoritmo de Gauss-Legendre, há essa dependência apenas para a_{n+1} e t_{n+1}. Este pequeno ganho nos métodos também pode ser duvidável, visto que outros processos no computador estavam sendo executados, podendo influenciar no cálculo do tempo de execução.

4 README

Cada algoritmo está implementado em um arquivo único. As implementações do método serial estão separadas do método paralelo, usando o sufixo *-threaded* no nome do arquivo. Um arquivo makefile foi criado para facilitar a compilação dos códigos. Para compilá-los, basta executar *make release*. As bibliotecas GNU MP e pthreads devem estar instaladas para o sucesso da operação. Os binários serão gerados na mesma pasta.

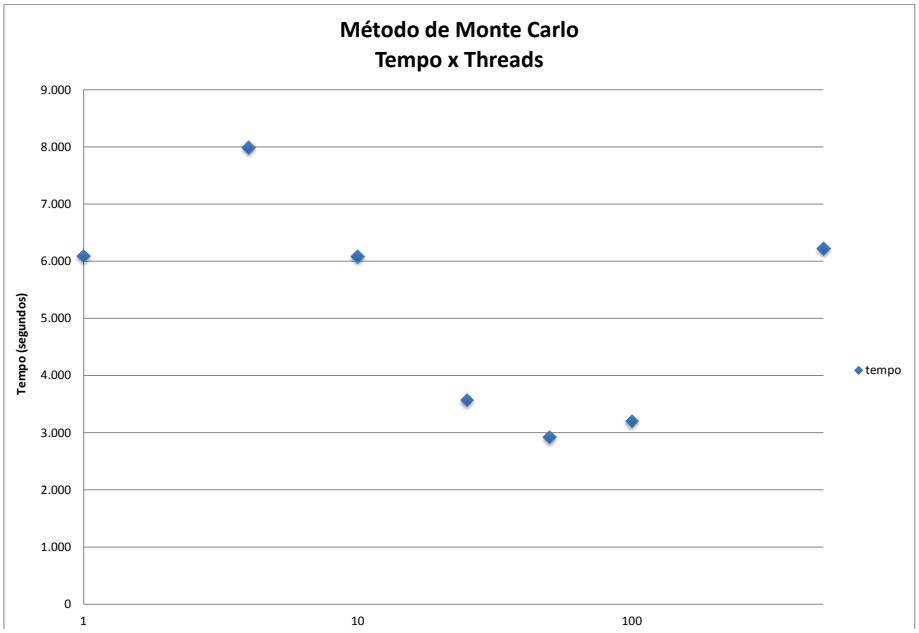

Figura 3: Resultados do algoritmo de Monte Carlo com otimização

5 Referências

- http://pubs.opengroup.org/onlinepubs/009695399/functions/rand.html
- http://www.gnu.org/software/libc/download.html
- http://cer.freeshell.org/renma/LibraryRandomNumber/
- http://en.wikipedia.org/wiki/Linear_congruential_generator
- http://mathworld.wolfram.com/MonteCarloMethod.html
- http://www.chem.unl.edu/zeng/joy/mclab/mcintro.html
- http://en.wikipedia.org/wiki/Native_POSIX_Thread_Library
- http://www.icir.org/gregor/tools/pthread-scheduling.html
- http://en.wikipedia.org/wiki/Gauss-Legendre_algorithm
- http://en.wikipedia.org/wiki/Borwein's_algorithm