UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS E
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO

Disciplina SCC0640: Sistemas Operacionais I
Relatório de Trabalho Prático

Trabalho Prático: Sistemas Operacionais I

Carlos Henrique Lima Melara - 9805380
Prof. Júlio C. Estrella

São Carlos
Junho de 2019

1. Objetivos

Realizar a comparação de *performance* entre programas que utilizam um *thread* único de execução e programas que utilizam mais um *thread* para execução do código. Para isso, serão implementados quatro programas, três deles para o cálculo da constante Pi e um para o cálculo do valor de opções. Cada um desses programas será implementado em um único *thread* e posteriormente utilizando *multithreads*. Esses programas serão executados e a *performance* deles será analisada comparativamente.

2. Métodos Numéricos

2.1. Gauss-Legendre

O método de Gauss-Legendre para o cálculo do número Pi utiliza o método de Gauss para aproximação da média aritmética-geométrica entre dois números. Iniciando-se com dois valores específicos, descobriu-se que é possível calcular o número Pi com grande precisão. Utilizaremos as seguintes fórmulas iterativas para o cálculo no algoritmo:

$$\begin{aligned} a_{n+1} &= \frac{a_n + b_n}{2}, \\ b_{n+1} &= \sqrt{a_n b_n}, \\ t_{n+1} &= t_n - p_n(a_n - a_{n+1})^2, \\ p_{n+1} &= 2p_n. \end{aligned}$$

De forma que o número Pi é calculado iterativamente da maneira mostrada a seguir.

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

Assim, necessitamos dos seguintes valores iniciais para calcular o valor de Pi.

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

O algoritmo segue calculando os valores de a_n, b_n, t_n, p_n e pi_n iterativamente até o número de iterações *N_ITER* definido no código fonte.

A **paralelização** do código foi feita a partir do cálculo dos parâmetros, isto é, a tarefa de calcular a_ne b_n foi delegada a um *thread*, o cálculo de t_n foi delegado a outro, e o cálculo de p_n e pi_n foi delegado para um terceiro *thread*. Cada um desses *threads* parte do valor inicial dos parâmetros e os calcula até o valor da 16.384ª iteração. Isto é feito para se utilizar ao

máximo da eficiência dos *threads* sem que haja um overhead muito grande criando e destruindo-se *threads*.

Para o código funcionar corretamente dessa maneira, é necessário implementar dois conceitos de programação concorrente, Semáforos e Variáveis de condição. O semáforos são utilizados para controlar o acesso à variável de controle de iteração de cada *thread*. As variáveis de condição são utilizadas para bloquear um *thread* enquanto ele não pode avançar sua execução (i.e. ele depende da ação de outro *thread* para prosseguir). Dessa forma, o cálculo dos parâmetros acaba sendo feito em sequência mesmo paralelizando-se a execução. Isto é uma consequência intrínseca ao método numérico em questão, atingido também o método de Borwein.

2.2. Borwein

O método de Borwein segue o mesmo padrão algorítmico do método de Gauss-Legendre, parte-se de valores iniciais pré-definidos para os parâmetros da fórmula iterativa, e então calcula-se iterativamente os valores até se atingir um número pré-determinado de iterações.

As fórmulas para o cálculo dos parâmetros são indicadas a seguir.

$$a_{n+1} = \frac{\sqrt{a_n} + 1/\sqrt{a_n}}{2}$$
$$b_{n+1} = \frac{(1+b_n)\sqrt{a_n}}{a_n + b_n}$$
$$p_{n+1} = \frac{(1+a_{n+1})\, p_n b_{n+1}}{1+b_{n+1}}$$

E os valores iniciais são os seguintes.

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

Vale salientar que as fórmulas iterativas e os valores iniciais apresentados são específicos para o método de convergência quadrática de Borwein. Para maiores graus de convergência, há mudanças tanto nas fórmulas, quanto nos valores iniciais.

A **paralelização** do método de Borwein seguiu a mesma abordagem do método de Gauss-Legendre. Um *thread* ficou responsável por calcular o parâmetro *a*, outro *thread* ficou responsável pelo cálculo do parâmetro *b*, e um terceiro, pelo parâmetro *c*. As mesmas formas de sincronização foram empregadas (semáforos e variáveis de condição) para controlar em

qual iteração cada *thread* está. E, da mesma forma, perde-se o paralelismo proporcionado pelos *threads* para manter os valores dos parâmetros coerentes a cada iteração.

2.3. Monte Carlo

O método de Monte Carlo baseia-se na análise estatística para determinar o valor de Pi. É sabido ser possível estimar a Esperança (***Ω***) de uma variável aleatória *X* a partir de *n* observações dos valores apreciados de uma variável aleatória X_i, onde *i = 1, ..., n*, independente e identicamente distribuída em relação à *X*. Se a média for dada por:

$$\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

Pode-se utilizá-la como um estimador para ***Ω*** através da Lei dos Grandes Números. Seguindo uma abordagem para valores *n* muito grandes, pelo teorema do Limite, essa média terá uma distribuição normal aproximada, com média próxima à ***Ω*** e desvio padrão baixo. Assim, quanto maior for o valor de *n*, melhor será a aproximação de ***Ω***.

O cálculo da constante Pi é feito a partir desta propriedade. Partindo de um vetor aleatório (***X***, ***Y***) que representa um ponto no espaço contido no quadrado de aresta 2 centrado na origem. A probabilidade deste ponto aleatório estar contido dentro de um círculo de raio 1 também centrado na origem é dada por

$$\begin{aligned} P\{(X,Y) \quad \textit{está no círculo}\} &= P\{X^2+Y^2 \leq 1\} \\ &= \frac{\textit{Área do círculo}}{\textit{Área do quadrado}} = \frac{\pi}{4}. \end{aligned}$$

Portanto, utilizando a propriedade apresentada inicialmente e o resultado acima, temos que, para um número grande de pontos gerados (por distribuições ***X*** e ***Y*** independentes e igualmente distribuídas em (-1, 1)) a esperança da probabilidade de estar contido no círculo é um quarto de Pi. Matematicamente isto é feito definindo uma variável aleatória *I* da seguinte forma:

$$I = \begin{cases} 1, & se \quad X^2+Y^2 \leq 1, \\ 0, & caso\ contrário\ . \end{cases}$$

Segue-se então que

$$E(I) = P\{X^2+Y^2 \leq 1\} = \frac{\pi}{4}.$$

Desta forma, pode-se montar um algoritmo para calcular o valor de Pi gerando vários pontos aleatórios dentro de um quadrado de área 4 e contabilizando quantos estão dentro do círculo de raio 1.

Para a **paralelização** deste algoritmo, optou-se por criar vários *threads*, cada um deles é responsável pelo cálculo e verificação de *n / N_THREADS* pontos. Após o cálculo e verificação de todos os pontos incubidos ao *thread*, ele precisa atualizar uma variável global com o número de pontos que estavam contidos no círculo. O acesso à essa variável global precisa ser controlada através de um semáforo mutex para garantir que somente um *thread* a acessará em um determinado instante de tempo. Após a finalização de todos os *threads* criados, o thread principal, o qual aguardava a finalização dos *threads* auxiliares, é responsável por calcular o valor de Pi utilizando o valor da variável global.

3. Metodologia

A análise de performance e tempo de execução é bastante crítica dado o número de variáveis que impactam a execução de programa em um sistema operacional moderno. Para tentar minimizar esse problema, os códigos serão executados em um ambiente controlado, somente esse processo do usuário estará ativo no momento, além dele podemos ter outros processos essenciais ao sistema ativos eventualmente. Será feita também uma média do tempo de execução de cinco instâncias do código fonte para tentar minimizar o impacto de variáveis desconhecidas.

A comparação entre os algoritmos será feita levando em consideração o número de iterações imposto aos algoritmos. Isto é, a performance do algoritmo de Borwein com *16.384* iterações será comparada com os outros algoritmos com o mesmo número de iterações. Além disso, será observado também o tempo de execução no espaço do usuário e no espaço do *kernel* para cada algoritmo. Também será feita a análise do algoritmo de Monte Carlo utilizando diferentes números de *threads* de execução, essa análise será feita seguindo os mesmo moldes propostos anteriormente.

4. Resultados

A *tabela 1*, a seguir, mostra os resultados de tempo de execução no espaço do usuário, espaço do núcleo e tempo de execução real (tempo transcorrido entre início e fim do processo) para os algoritmos de Gauss-Legendre, Borwein e Monte Carlo com número de iterações igual a *16.384*.

Tabela 1: *Exibe tempo de execução médio em segundos para algoritmos com N_ITER = 16.384.*

	User time	Kernel time	Real Time
Gauss Sequencial	0,004	-	0,004
Gauss Paralelo	0,06	0,19	0,196
Borwein Sequencial	0,005	-	0,005
Borwein Paralelo	0,076	0,16	0,19
Monte Carlo Sequencial	0,004	-	0,004
Monte Carlo Paralelo	0,01	0,014	0,009

Vale salientar que os valores obtidos para as 5 execuções de cada algoritmo foram muito próximos, ou seja, a variância foi baixa. Esses valores serão mostrados na *tabela 2*.

Tabela 2: *Exibe o desvio padrão do tempo de execução médio em segundos para algoritmos com N_ITER = 16.384.*

	User time	Kernel time	Real Time
Gauss Sequencial	0	-	0
Gauss Paralelo	0,01	0,01	0,003
Borwein Sequencial	0	-	0
Borwein Paralelo	0,008	0,01	0,01
Monte Carlo Sequencial	0,001	-	0,001
Monte Carlo Paralelo	0,006	0,008	0,001

Seguindo com os resultados, parte-se para a análise com o número de iterações igual a 10^9. Para esse valor, foram testados somente os algoritmos de Borwein e Monte Carlo já que o algoritmo de Gauss-Legendre esbarra em um problema de *overflow* descrito no *Apêndice A*. Na *tabela 3*, estão dispostos também os valores obtidos variando-se o número de *threads* de execução do algoritmo de Monte Carlo, os valores utilizados foram *2*, *4* e *8* - visto que a máquina de teste possui apenas *2 cores* e *4 threads*.

Tabela 3: *Exibe tempo de execução médio em segundos para algoritmos com N_ITER = 10^9.*

	User time	Kernel time	Real Time
MC sequencial	40,3	0,003	40,3
MC paralelo 2 Thd	900	1182	1042
MC paralelo 4 Thd	490	756	334
MC paralelo 8 Thd	497	834	338
Borwein sequencial	46,38	0,001	46,39
Borwein paralelo	> 580	> 1355	> 1532

Na *tabela 4*, são exibidos os valores do desvio padrão calculado para as médias. Novamente observa-se que não há grandes variações em relação à média obtida.

Tabela 4: *Exibe o desvio padrão do tempo de execução médio em segundos para algoritmos com* $N\_ITER = 10^9$.

	User time	Kernel time	Real Time
MC sequencial	0,9	0,002	0,9
MC paralelo 2 Thd	10	11	7
MC paralelo 4 Thd	2	3	1
MC paralelo 8 Thd	10	5	4
Borwein sequencial	0,02	0,002	0,02
Borwein paralelo	-	-	-

5. Conclusão

Analisando os dados obtidos, observa-se uma grande piora na *performance*, isto é tempo de execução real, quando os algoritmos iterativos e intrinsecamente sequenciais (Gauss, Borwein) são paralelizados usando a biblioteca *pthread*. Isto é, até certo ponto, intuitivo pois torna-se necessário implementar/utilizar diversas outras estruturas (e.g. semáforos, variáveis de condição, sinais) para viabilizar a condição sequencial do algoritmo. Esses mecanismos causam um grande *overhead* quando são executados[1], e eles são executados pelo menos *N_ITER* vezes, incidindo principalmente no espaço do *kernel*. Isto é mostrado quando analisa-se comparativamente a diferença entre o tempo médio no espaço do *kernel* no algoritmo sequencial e no algoritmo paralelo com *N_ITER* igual a *16.384*, e a diferença entre os tempos médios no espaço do núcleo com *N_ITER* igual a 10^9.

Com relação aos resultados obtidos sobre a paralelização do algoritmo de Monte Carlo, o qual, a princípio, seria beneficiado com a execução em *multithread*, mostrou também uma *performance* inferior quando implementado com a biblioteca *pthread*. Algumas hipóteses podem ser geradas com relação à quebra da expectativa inicial. Primeiramente, quando há utilização de mais de um núcleo, especialmente em um processador dual core, há maior disputa de tempo de CPU com processos essenciais ao sistema, e eles geralmente têm maior prioridade, o que causa um maior número de preempções e mudanças de contexto. Além disso, é necessário manter a tabela de *threads* no núcleo atualizada e consistente com a execução em tempo real. Também há o *overhead* de criação dos *threads*, do semáforo, da tabela de atributos de *thread*, e da requisição de acesso ao semáforo, da liberação do mesmo e da desalocação de todas estas estruturas. Todos esses procedimentos e hipóteses impactam principalmente no tempo de execução no espaço do núcleo, mas também atingem o tempo gasto no espaço do usuário como podemos observar na *tabela 3* comparando os valores para o algoritmo sequencial e o paralelo.

Fica claro, tendo observado todos estes dados, que para aplicações limitadas à CPU (*CPU-bound*) não há tantos benefícios ao se utilizar a paralelização, principalmente pelo *overhead* de acesso ao núcleo do sistema. Porém é bastante provável que aplicações orientadas à entrada e saída (I/O-bound) sejam bastante beneficiadas pelas vantagens de se utilizar *multithreading* visível ao *kernel* do sistema operacional.

6. Apêndice

6.1. README

Os códigos foram escritos utilizando a biblioteca *math.h*, por isso todos os códigos fontes devem ser compilados utilizando a opção ***-lm***. Além disso, os códigos feitos para utilizar mais de um *thread* de execução devem ser compilados com a opção ***-pthread*** porque utilizam a biblioteca *pthread.h*.

O método de Gauss-Legendre possui um de seus parâmetros com crescimento exponencial da forma 2^n, de modo que, para não exceder o limite superior da variável *long double*, deve-se calcular o algoritmo até a iteração número *16.384*. Dessa forma, todas as análises comparando seu tempo de execução com o de outros algoritmos será feito com os outros algoritmos também limitados à iteração *16.384*.

O cálculo do preço de opções no mercado europeu foi implementado sequencialmente, mas como os resultados não foram coerentes com os propostos pelo Professor e por outros algoritmos implementados *online*, mesmo após extensas verificações da implementação, esse algoritmo não foi testado, nem implementado paralelamente. Contudo, sua elaboração utilizando mais de um *thread* de execução deve seguir a mesma implementação empregada no algoritmo de Monte Carlo *multithreaded*.