Universidade de São Paulo
Instituto de Matemática e Estatística
Departamento de Ciência da Computação

MAC0438 - Programação Concorrente
EP2 - Aproximação da função cosseno por série de Taylor

Professor: Daniel Macedo Batista

Autoras:
Bárbara de Castro Fernandes - 7577351
Taís Aparecida Pereira Pinheiro - 7580421

São Paulo - SP, 25 de maio de 2015

Sumário

1 Introdução

A **série de Taylor** de uma função é a representação desta função como uma série de infinitos termos calculados a partir de suas derivadas. Quanto mais termos desta série forem calculados, maior será a precisão de seu resultado.

Neste exercício-programa utilizamos a série de Taylor correspondente à função trigonométrica **cosseno**. Sua fórmula é a seguinte:

$$cos(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

Para aumentar a eficiência do cálculo, o programa é normalmente feito de forma paralelizada, onde cada *thread* calcula um termo de forma independente dos outros.

A condição de parada pode ser atingida de duas formas. Dependendo dos valores dados como entrada, o cálculo do cosseno pode acabar quando o módulo da diferença entre o valor do cosseno calculado por duas rodadas consecutivas for menor do que um dos argumentos ou quando o programa calcular um termo menor, em módulo, do que este argumento.

2 Desenvolvimento

2.1 Bibliotecas

Das bibliotecas da linguagem C utilizadas neste EP, destacam-se graças às suas funcionalidades as seguintes:

- **gmp.h**: Biblioteca para cálculos de precisão arbitrária com números inteiros, racionais ou de ponto flutuante.
- **pthread.h**: Responsável pela criação e gerenciamento das *threads* que calculam os termos da série.
- **semaphore.h**: Define um tipo de variável **semáforo**, e é utilizada para realizar as operações com semáforos necessárias para o funcionamento correto das *threads*.

2.2 Estruturas principais

As estruturas principais utilizadas na elaboração deste exercício-programa são estruturas já prontas localizadas nas três bibliotecas citadas anteriormente.

- **pthread_t**: Variável do tipo ***thread*** localizada na biblioteca **pthread.h**.
- **pthread_barrier_t**: Variável do tipo **barreira** localizada na biblioteca **pthread.h**.
- **sem_t**: Variável do tipo **semáforo** localizada na biblioteca **semaphore.h**.
- **mpf_t**: Variável do tipo **ponto flutuante** localizada na biblioteca **gmp.h**.

2.3 Inicialização

A princípio temos a captura dos argumentos passados pela linha de comando. Em seguida, inicializamos e globalizamos certas variáveis com ajuda das seguintes funções:

- **void globalizaParametros(int n, mpf_t x, char *modo, char *opcional)**: Globaliza o valor de algumas variáveis dadas como entrada que serão utilizadas por outras funções.
- **void inicializaControladores(int n)**: Inicializa os semáforos e a barreira do programa;

  Existem ainda outras funções menores que oferecem suporte a estas citadas.

2.4 Precisão

A função **mpf_set_default_prec(unsigned long int prec)** da biblioteca gmp.h tem como função definir a precisão padrão dos números de ponto flutuante do tipo mpf_t. Em sua chamada que fazemos na função main(), passamos como argumento $NUM\_DIGITOS \ / \ log_{10}^2$ onde $NUM\_DIGITOS = 100.000$, devido ao fato da função receber a precisão em bits. Portanto, temos que convertê-la para bits passando a precisão da base decimal para a base binária.

Na impressão dos resultados indicamos que a número de dígitos apresentados deve ser $NUM\_DIGITOS$ para que o valor seja apresentado em base decimal.

Para os resultados dos testes apresentados, utilizamos a fim de simplificar a execução dos testes uma precisão menor do que a pedida. Os resultados, porém, permanecem coerentes porque os tempos de execução para os diferentes modos de utilização mudam proporcionalmente de acordo com a precisão que colocamos.

2.5 Cálculo do cosseno

2.5.1 Cálculo do cosseno com processos paralelos

A maior parte dos modos de execução do programa realiza os cálculos através da utilização de processos paralelos. As funções responsáveis por essa parte do programa são as seguintes:

- **void cossenoConcorrente()**: Responsável pela criação, execução e destruição das *threads*;
- **void *calculaCossenoConc(void *param)**: Função encarregada do gerenciamento das *threads*;
- **void calculaTermoCossenoConc(int id, int n)**: Calcula o valor do termo de uma *thread*.
- **void condicaoDeParadaConc(mpf_t termo)**: Verifica se a condição de parada escolhida pelo usuário foi atingida.
- **void incrementaCosseno(mpf_t termo, char c)**: Incrementa o valor da variável cosseno ou da variável cosseno auxiliar com o valor do último termo calculado.

2.5.2 Cálculo do cosseno de forma sequencial

O cálculo do cosseno é feito de forma sequencial se a opção **'s'** for passada como argumento opcional. A função **void cossenoSequencial()** é a encarregada de cuidar desse caso.

2.6 Exibição dos resultados

A relação dos resultados é exibida através de um relatório. Para obtenção dos dados utilizamos a função **void imprimeResultado(int n)**. Esta função imprime os valores parciais e finais do cosseno, assim como a quantidade de termos calculados e o número de rodadas realizadas, de acordo com as opções dadas como entrada.

3 Dificuldades

Assim como no exercício-programa anterior, a principal dificuldade foi testar o programa a cada passo. Se ocorresse de o programa travar no meio da execução, achar o ponto em que esse erro ocorria era bastante problemático devido ao fato de não conseguirmos localizar, através da impressão de algum texto na saída-padrão, que linha estava sendo executada.

4 Modo de execução

Este projeto foi entregue em um arquivo comprimido de extensão **.tar.gz** contendo 5 arquivos:

- **ep2.c**: Arquivo com o código fonte deste exercício-programa.
- **ep2.h**: Arquivo header do código fonte ep1.c.
- **Makefile**: Makefile para compilação do programa.
- **LEIAME**: Arquivo com a descrição técnica no programa.
- **Relatório.pdf**: Este relatório.

4.1 Compilação

Para compilar, digite:

```
$ make
```

4.2 Entrada

Existem três maneiras de se executar o programa principal após a devida compilação:

- **./ep2 n f p x**: Calcula o valor de cos(**x**) através da criação de **n** *threads*. Se **n** for igual a zero, serão criadas uma quantidade de *threads* igual à quantidade de núcleos do computador. A condição de parada será atendida quando o módulo da diferença entre dois valores de cos(**x**) calculados por duas rodadas consecutivas for menor que $10^{-\mathbf{p}}$. Após o término da execução serão impressos o número de rodadas e o valor final de cos(**x**).

- **./ep2 n m p x**: Calcula o valor de cos(**x**) através da criação de **n** *threads*. Se **n** for igual a zero, serão criadas uma quantidade de *threads* igual à quantidade de núcleos do computador. A condição de parada será atendida quando alguma *thread* calcular um termo menor, em módulo, que **p**. Após o término da execução serão impressos o número de rodadas e o valor final de cos(**x**).

- **./ep2 n [f|m] p x [d|s]**: Se for passado o argumento **d**, serão impressas a cada rodada a ordem em que as *threads* chegam na barreira e o valor parcial de cos(**x**) e, após o término da execução, o número de rodadas e o valor final de cos(**x**). Se o argumento passado for **s**, será impressa a cada rodada o valor parcial de cos(x) e, após o término da execução, a quantidade de termos calculados e o valor final de cos(**x**).

5 Análise de desempenho

O programa foi desenvolvido conjuntamente em dois computadores com as seguintes configurações:

Computador 1:
Marca: ASUS
Modelo: K53SD
Processador: Intel ® Core™ i3-2350M CPU 2.30GHz x 4
Memória: 5,6 GB
Disco: 63,3 GB

Computador 2:
Marca: ASUS
Modelo: K45VM
Processador: Intel® Core™ i7 3610QM
Memória: 8 GB
Disco: 500 GB

Foram computadas para análise duas entradas iguais para as duas máquinas em que o programa foi desenvolvido. As entradas foram escolhidas como exposto abaixo e os resultados foram medidos em milissegundos:

Entrada 1:

Precisão = 100.000

x = 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706798214808651328230664709384460955058223172535940812848111745084102701938521105559644622948954930381964428810975665933446128475648233786783165271201909145648566923460348610454326648213393607260249141273

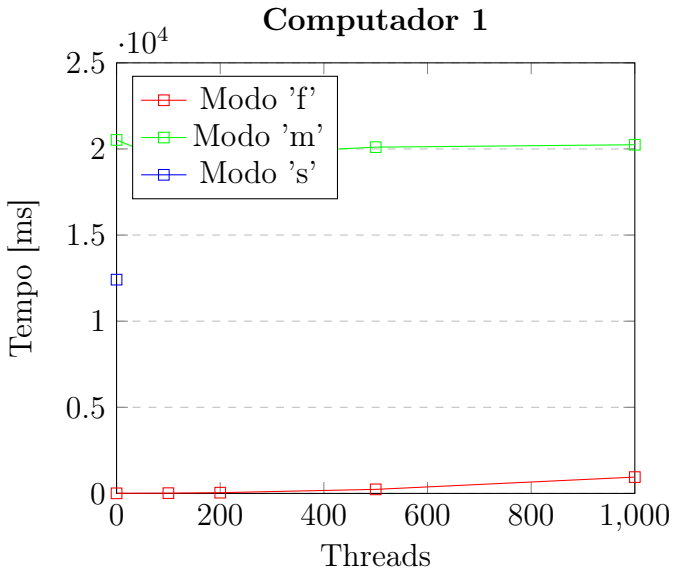
Entrada 2:

Precisão = 10.000

x = 6.28318530767982148086513282306647093844609550582231725359408128481117450284102701938521105559644622948954930381964455552525288894144789652314785252417728316532749685741235471196857432145879632587412547896541235874217895412365863 25

Tabela 1: Resultados medidos no Computador 1 para Entrada 1:

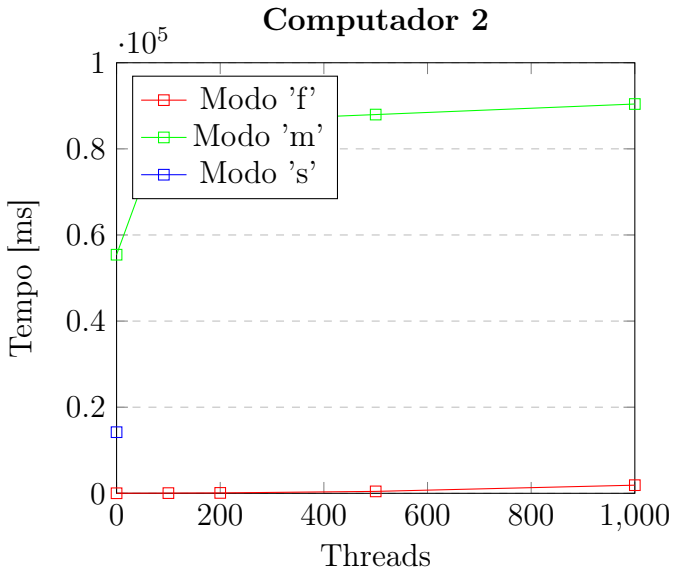
Número de *threads*	Modo *'f'*	Modo *'m'*
0	6	20525
100	15	19280
200	44	19688
500	235	20104
1000	948	20245
Modo sequencial	12412	



Média para o modo *'f'* = 249,6 ms
Média para o modo *'m'* = 19964,4 ms

Tabela 2: Resultados medidos no Computador 2 para Entrada 1:

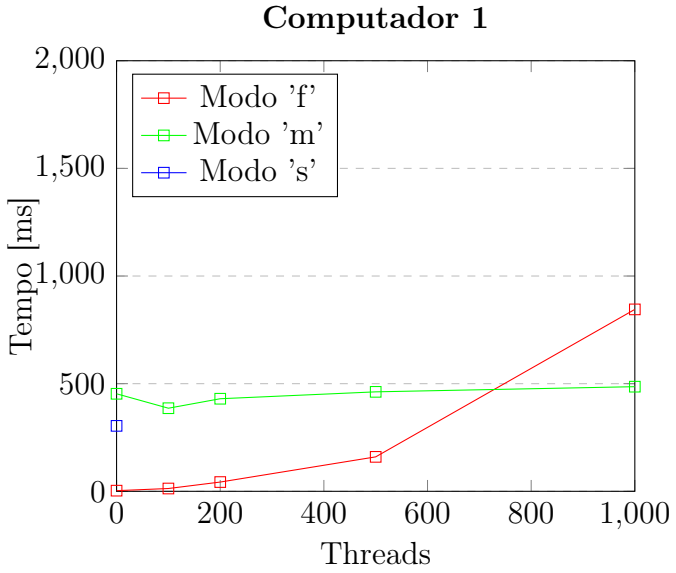
Número de *threads*	Modo *'f'*	Modo *'m'*
0	14	55416
100	48	88316
200	92	86491
500	448	87963
1000	1890	90413
Modo sequencial	14197	



Média para o modo *'f'* = 498,4 ms
Média para o modo *'m'* = 81719,8 ms

Tabela 3: Resultados medidos no Computador 1 para Entrada 2:

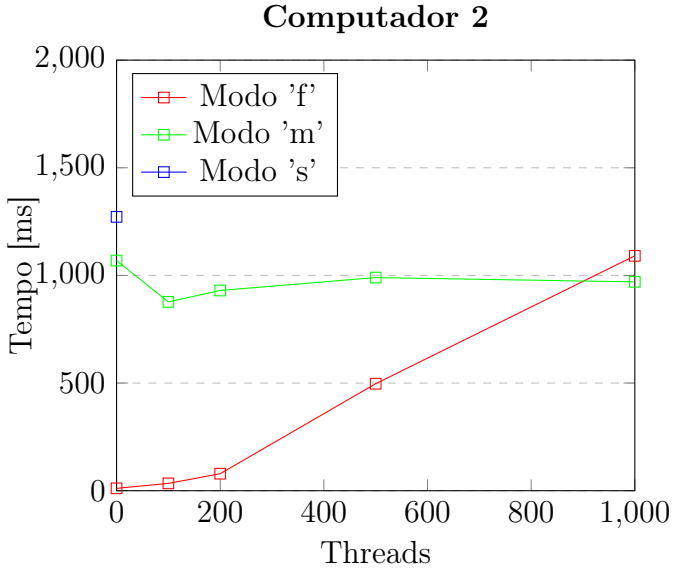
Número de *threads*	Modo *'f'*	Modo *'m'*
0	3	453
100	13	386
200	43	430
500	160	462
1000	845	486
Modo sequencial	304	



Média para o modo *'f'* = 212,8 ms
Média para o modo *'m'* = 443,4 ms

Tabela 4: Resultados medidos no Computador 2 para Entrada 2:

Número de *threads*	Modo *'f'*	Modo *'m'*
0	11	1069
100	34	877
200	79	930
500	497	990
1000	1091	970
Modo sequencial	1272	



Média para o modo *'f'* = 342,4 ms
Média para o modo *'m'* = 967,2 ms

Medição do tempo:

O tempo de execução do Programa para as entradas declaradas foi medido pela função *clock()* da biblioteca *time.h* nativa da linguagem C.

Considerações sobre os resultados:

Os resultados para o modo *'f'* se mostraram melhores em relação ao modo *'m'* como esperávamos para os dois computadores, uma vez que neste modo N termos são calculados pelas *threads* e só então é avaliada a condição de parada, contrário ao modo *'m'* que tem a condição de parada verificada a cada novo termo gerado.

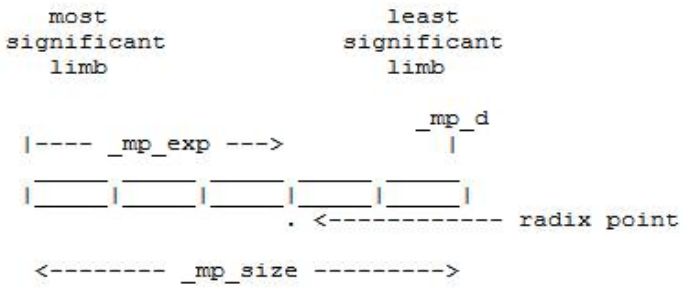
Esperávamos também que o modo *sequencial* levasse tempo superior ao modo *concorrente*, o que não aconteceu na prática para todos os casos.

6 Considerações sobre a biblioteca GMP

Observamos que a biblioteca GMP permite trabalhar sem limites práticos de precisão, no entanto, como os recursos de desenvolvimento do programa não são infinitos, consideramos que o principal limitante para uso da biblioteca é a memória da máquina.

6.1 Como é possível obter alta precisão dos números?

Neste programa, utilizamos predominantemente o tipo de variável GMP *mpf_t*, o armazenamento deste tipo de variável é dado da seguinte forma:



Os domínios são como se segue:

_mp_size: O número de membros em uso no momento corrente, ou o negativo do mesmo, quando representa um valor negativo. Zero é representado por _mp_size e _mp_exp, ambos definidos como zero.

_mp_prec: A precisão da mantissa, em números. Em qualquer cálculo o objetivo é produzir *_mp_prec* números de resultado (sendo o mais significativo diferente de zero).

_mp_d: Um ponteiro para o conjunto de números, que é o valor absoluto da mantissa. Estes são armazenados *"little endian"* de acordo com as mpn funções, de modo *_mp_d[0]* é o valor menos significativo e valor *_mp_d[ABS(_mp_size)-1]* o mais significativo.

O número mais importante é sempre diferente de zero, mas não existem outras restrições sobre o seu valor. *_mp_prec+1* são atribuídos aos valores *_mp_d*, o valor adicional é atribuído por conveniência. Não há reajustamentos durante um cálculo, apenas em uma mudança de precisão com *mpf_set_prec*.

_mp_exp: O expoente, nos números, determina a localização do ponto da raiz implícita. Zero significa que o ponto da raiz é um pouco acima do membro mais importante. Os valores positivos significam um deslocamento para os membros inferiores. Expoentes negativos significam um ponto da raiz ainda mais acima do mais alto membro. Naturalmente, o expoente pode ser qualquer valor. Outros que não os incluídos nos membros *_mp_d,_mp_size* dados são tratados como zero.

Os campos *_mp_size* e *_mp_prec* são inteiros , embora o tipo *mp_size_t* é geralmente um long, assim como o campo *_mp_exp*. Isto é feito para fazer tratar alguns campos de 32 bits em alguns sistemas de 64 bits, poupando assim alguns bytes de espaço de dados, mas ainda fornece uma abundância de precisão e uma gama de valores muito grande.

7 Considerações sobre a biblioteca *pthread*

7.1 Sobre a barreira de sincronização

A barreira de sincronização localizada na biblioteca *pthread* é implementada através de uma estrutura contendo as seguintes variáveis:

- Uma variável do tipo *mutex*;
- Um contador para a quantidade de *threads* que ainda necessitam passar pela barreira;
- Número necessário de *threads* para atravessar a barreira;
- Um contador de eventos.

Cada vez que a função **pthread_barrier_wait(barrier_t *barrier)** é executada, ela verifica se a *thread* que chegou à função é a última. Em caso negativo, desbloqueia-se a barreira, e espera-se até que o contador de eventos da barreira mude. Caso seja a última, incrementa-se a quantidade de eventos, acordam-se todos os processos e devolve-se o valor de retorno referente à última *thread*.

8 Referências

Referências

[1] Enunciado do exercício-programa

[2] Séries de Taylor

[3] Tutorial para a biblioteca pthread

[4] Manual da biblioteca GMP

[5] Análise de desempenho da biblioteca GMP

[6] Análise interna das Variáveis mpf_t