MAC0438 – EP2

Evandro Fernandes Giovanini – nUSP 5173890

1. Introdução

O programa calcula uma aproximação do número de Euler (*e*) de forma concorrente e com alta precisão.

2. Detalhes da Implementação

A fórmula usada para a aproximação é derivada da série de Taylor descrita no enunciado.

O programa foi escrito em C e testado com os compiladores GCC e LLVM. Foram usadas as bibliotecas GMP para tratamento de números de alta precisão e pthreads para concorrência.

Para sincronização de threads foram usadas duas barreiras do tipo pthread_barrier_t. As threads chegam na primeira barreira, fazem os seus cálculos e depois esperam na segunda barreira até que todas as threads terminem. Isso é importante para garantir que nenhuma thread passe para a próxima iteração se o valor de parada já foi atingido; nesse caso a thread que passou ficaria esperando outras chegarem na barreira, mas essas já teriam saído pela condição de parada e teríamos um deadlock.

3. Instruções

Para compilar o programa rode o comando `make`. Isso irá criar o executável `ep2`, que tem seus argumentos válidos descritos no enunciado do EP.

4. Testes

Os testes abaixo foram executados em um notebook com processador de dois núcleos com hyper threading Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz e 4GB de memória DDR3. O sistema operacional é o Fedora 20 de 64 bits, com as seguintes versões de software: kernel-3.14.4-200.fc20.x86_64 , glibc-2.18-12.fc20.x86_64 e gcc-4.8.2-7.fc20.x86_64. Os tempos foram medidos com o programa `GNU time`.

Note que o sistema operacional enxerga o processador de dois núcleos como sendo de quatro porque ele tem a tecnologia hyper threading da Intel, que permite a execução de mais de uma thread simultaneamente no pipeline do processador. Os valores abaixo correspondem a execução de quatro threads em um processador de apenas dois núcleos, não quatro.

Todos os testes foram criados a partir dos scripts no diretório `testes/`. Eles foram executados 5 vezes cada, medindo com o comando time. Apresentamos a média e o desvio padrão

nos gráficos abaixo. Os valores completos de todas as execuções de teste estão na planilha `testes/resultados.ods`.

Teste 1: opção f, valor 10^{-2000}

Teste 1	Sequencial	Uma thread	Duas threads	Quatro threads
Tempo médio (s)	3,829	3,999	4,290	3,536
Desvio padrão	0,007	0,011	0,390	0,071

Teste 2: opção m, valor 10^{-2000}

Teste 2	Sequencial	Uma thread	Duas threads	Quatro threads
Tempo médio (s)	3,711	3,808	3,118	2,675
Desvio padrão	0,005	0,040	0,424	0,063

Também forçamos o algoritmo a calcular duas mil iterações ao invés de usar as condições m e f de parada. Note que esta versão alterada não é precisa pois estoura a precisão (seria necessário muita memória RAM para cálculos tão precisos), mas ainda assim é útil comparar o tempo de execução do algoritmo.

Teste 3: 2000 iterações

Teste 3	Sequencial	Uma thread	Duas threads	Quatro threads
Tempo médio (s)	15,350	15,473	10,180	8,857
Desvio padrão	0,170	0,020	0,784	0,061

5. Análise dos Resultados

O tempo de execução nos mostrou que não vale a pena usar concorrência quando vamos calcular e na ordem 10^{-2000}. O algoritmo roda tão rapidamente que os ganhos devido a paralelização não são suficientes para justificar o overhead causado, e o programa acaba ficando mais lento.

Usando a modificação do programa que força o cálculo de um número grande de termos, independentes da precisão correta, nos leva a resultados mais otimistas. Nesse caso o algoritmo concorrente mostrou uma boa vantagem sobre o programa sequencial.

Algo curioso de se notar nos testes 1 e 2 é que o desvio padrão é bem maior nos programas concorrentes. Isso também mostra que boa parte do tempo se perde no overhead causado pela criação de threads, tarefa que depende do estado do sistema operacional e portanto varia mais entre execuções diferentes.

Nas considerações finais fazemos observações sobre otimizações que poderíamos fazer para justificar ainda mais o uso de concorrência, tornando o algoritmo mais rápido.

6. Considerações Finais

Este programa foi minha primeira experiência com a biblioteca GMP, e eu gostei bastante. A biblioteca é impressionante e cumpre bem o seu propósito, além de ser fácil de usar.

O programa calcula milhares de dígitos do número *e* em segundos, mas ainda poderíamos fazer mais otimizações no código. No algoritmo atual cada thread mantém seu próprio valor do fatorial, fazendo suas multiplicações, o que passa a ser custoso com valores muito grandes; poderíamos implementar alguma colaboração entre as threads. Outra otimização possível seria incrementar a precisão necessária em cada passo usando a função mpf_set_prec_raw da biblioteca GMP, já que quanto maior a precisão pedida mais lentos são os cálculos, e não é necessário usarmos uma precisão grande para calcular os termos iniciais da sequência; poderíamos aumentá-la gradativamente conforme necessário.