Insper

07 - Thread safety

Super Computação 2018/2

Igor Montagner, Luciano Soares

Vamos trabalhar hoje com os arquivos *pi_mc.c* e *random.c/h*. Serão testadas duas técnicas diferentes de paralelização neste roteiro e ambas possuem vantagens e desvantagens. O algoritmo que usamos para gerar números aleatórios é o *LCG* (Linear Congruent Generator). A aleatoriedade da sequência depende da escolha dos 3 parâmetros `mult, add` e `pmod` do algoritmo.

Nosso principal objetivo hoje é escrever código *sem efeitos colaterais*. Ou seja, toda a informação usada deve ser passada como argumento para nossas funções e toda modificação é feita via um valor retornado. No próximo módulo serão tratadas técnicas de sincronização entre threads.

Para transformar código "comum" em código sem efeitos colaterais tipicamente precisamos eliminar variáveis globais e evitar o uso de ponteiros. Não se esqueça de que variáveis `static` também são globais, porém seu escopo é limitado a um arquivo ou função específica. Outro ponto importante é podemos passar a mesma variável como ponteiro para duas funções rodando em threads diferentes. Logo, uma thread poderia modificar o andamento de outra, violando a ideia de efeito colateral.

Parte 0: analizando o código existente

Nesta parte do roteiro iremos analisar o código exemplo, testar uma paralelização ingênua e identificar seus possíveis problemas de paralelização.

Exercício 1: Considerando somente o arquivo *pi_mc.c*, existe código com efeitos colaterais? Faça uma paralelização ingênua deste código.

Exercício 2: Teste a paralelização ingênua do exercício anterior. Ela retorna os mesmos resultados em todas execuções? Se não, comente por que isto é um problema.

Exercício 3: Dado que não encontramos problemas no arquivo *pi_mc.c*, vamos olhar então os arquivos *random.c/h*.

1. Existe código com efeitos colaterais? Liste as funções encontradas.

2. Voltando para *pi_mc.c*, onde são chamadas as funções identificadas acima?

3. Agora que você está familiarizado com todo o código, explique por que os resultados são diferentes quando rodamos o código ingenuamente paralelo.

Só prossiga após validar as respostas do item anterior com o professor ou com um colega que já tenha finalizado esta parte.

Parte 1: paralelização das distâncias

Identificamos na parte anterior que a função `drandom` possui efeitos colaterais e estes efeitos colaterais estão atrapalhando a paralelização do código. Note, porém, que os cálculos de distância são independentes desde que os sorteios aleatórios já tenham sido feitos.

Exercício: Modifique o código de *pi_mc.c* para que os sorteios dos pontos seja feito antes da região paralela que computa `Ncirc`. Ou seja, primeiro eu sorteio todos os pontos e depois eu paralelizo a contagem.

Exercício: Houve ganho expressivo de desempenho?

Exercício (opcional): Tente modificar seu código para que o autovetorizador possa otimizar o loop do cálculo das distâncias. Você não deve usar OpenMP neste caso. Compare os valores obtidos com o código acima.

Esta estratégia é muito comum em casos em que a tarefa de interesse pode ser decomposta em uma parte inerentemente sequencial e uma que pode ser paralelizada mas que depende dos resultados da parte sequencial. Se a parte paralelizável for custosa essa estratégia pode trazer ganhos mesmo que o programa não seja inteiramente paralelizável.

Parte 2: paralelização

A segunda estratégia que usaremos é criar um gerador de números aleatórios para cada thread. Ou seja, cada thread precisará guardar:

1. Seus parâmetros `mult` e `pmod`. Para simplificar, vamos supor que `add=0`.
2. Os valores `hi` e `low`, que indicam o intervalo em que os números são gerados.
3. O último número de sua sequência.

Isto implica que precisamos criar uma lista de possíveis bons valores para `mult` e `pmod` e que esta lista deve estar acessível durante a compilação. Supondo que identificamos 8 bons valores, podemos setar o número máximo de threads do OpenMP usando a função `omp_set_num_threads`. Uma lista de bons valores para `mult` e `pmod` pode ser vista neste link(http://www.ams.org/journals/mcom/1999-68-225/S0025-5718-99-00996-5/S0025-5718-99-00996-5.pdf)

Exercício: seu trabalho nesta parte será reorganizar o código de *random.c/h* de modo que todas suas funções não possuam efeitos colaterais.

Exercício: esta estratégia paralelizaçao é capaz de produzir os mesmos resultados que um programa sequencial que possua somente um gerador de números aleatórios? Os resultados se modificam se mudamos o número de threads usadas?

Sugestão de organização do código:

1. Crie uma estrutura/classe com os atributos que antes eram globais.
2. Cria uma função/construtor que inicialize todos os atributos acima.
3. Crie uma função `get_next_random` que recebe a estrutura acima e devolve uma cópia contento o valor do último número da sequência modificado. Note que este é o único parâmetro que necessita de atualização, todos os outros continuam os mesmos.

Se você não entendeu alguma das instruções acima discuta-as com um colega. Depois, procure o professor para validar suas conclusões.

Parte final

Compare o desempenho das duas abordagens de paralelização. Elas chegam em bons resultados usando números similares de iterações? Qual é mais fácil de ser entendida?

Opcional

Se você chegou aqui tem duas opções:

1. Estudar e implementar o método *Leapfrog LCG*, que é uma modificação paralela do LCG e que possui resultados equivalentes a uma execução sequencial do LCG. Apesar de isto parecer desejável, existem problemas em sua utilização. Pesquise quais são eles e conclua se estes problemas afetariam o método que tratamos neste roteiro.
2. Trabalhar no projeto 1.