Insper

03 - Introdução a OpenMP

SuperComputação - 2018/2

Igor Montagner, Luciano Soares

Parte 1 - OpenMP

Nesta parte do roteiro usaremos 3 chamdas do OpenMP para recriar o primeiro exemplo da aula passada.

1. `#pragma omp parallel` cria um conjunto de threads. Deve ser aplicado acima de um bloco de código limitado por `{ }`
2. `int omp_get_num_threads();` retorna o número de threads criadas (dentro de uma região paralela)
3. `int omp_get_thread_num();` retorna o id da thread atual (entre 0 e o valor acima, dentro de uma região paralela)

O código abaixo (*exemplo1.c*) ilustra como utilizar OpenMP para fazer o exercicio 1 do roteiro anterior (criar 4 threads e imprimir um id de 0 a 3).

```
#include <iostream>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        std::cout << "ID:" << omp_get_thread_num() << "/" <<
                            omp_get_num_threads() << "\n";
    }
    std::cout << "Join implicito no fim do bloco!" << "\n";
    return 0;
}
```

A compilação de código usando OpenMP é feita com as seguintes flags:

```
$ g++ (flags de compilação) arquivo.cpp -o nome_executavel
-fopenmp
```

O OpenMP permite alterar o número máximo de threads criados usando a variável de ambiente `OMP_NUM_THREADS`. Compile o código acima e o execute usando a seguinte linha de comando.

```
OMP_NUM_THREADS=2 ./exemplo1
```

Os resultados foram os esperados? Rode agora sem a variável de ambiente. Qual é o valor padrão assumido pelo OpenMP? É uma boa ideia usar mais threads que o valor padrão?

A utilização desta variável de ambiente ajuda a realizar testes de modo a compreender os ganhos de desempenho de um programa conforme mais threads são utilizadas.

Parte 2 - Cálculo do PI

O arquivo *calculo_pi.cpp* contém uma implementação serial do

Exercício 1: Refatore o arquivo *calculo_pi.cpp* para que o cálculo serial seja feito um uma função `double pi_seq(long steps);`.

Exercício 2: Crie uma função `double pi_par1(long steps)` que faça o cálculo do pi de modo paralelo. Faça seu trabalho nos moldes do exerício 2 do roteiro passado:

1. As iterações do `for` são divididas por igual entre as threads;
2. Cada thread acumula seus resultados parciais em um vetor `double sum[]`;
   - Por efeitos de exercício, garanta que você faz uma construção do tipo `sum[id] += ...;`
3. No fim os resultados parciais são usados para o cálculo final.

Teste seu programa com 1 a 4 threads. Os ganhos de desempenho

Exercício 3: Apesar do código acima ter um desempenho bom, é recomendado escrever o mínimo possível em variáveis globais. Troque a acumulação diretamente no vetor de somas parciais pela acumulação em uma variável local e escreva o valor final no vetor global somente após ele ser calculado.

Além das construções acima o OpenMP suporta duas construções de sincronização de alto nível: `critical` e `atomic`. A diretiva `atomic` executa uma atribuição ou uma operação aritmética *inplace* (`+=, -=, *=, /=`).

Exercício 4: Como você usaria `atomic` para simplificar o código do seu programa? O uso correto de `atomic` não resulta em perda de velocidade.

A diretiva `critical` é aplicada a um bloco e faz com que ele esteja em execução em no máximo 1 das threads. Este nome vem do conceito de *seção crítica*, que representa uma seção de uma tarefa que não pode ser paralelizada de jeito algum e obrigatoriamente deve ser executada de modo sequencial. O uso de `critical` é muito perigoso, pois ao forçar a execução sequencial de um bloco de

código podemos estar efetivamente matando o paralelismo do nosso programa. A construção `atomic` é uma seção crítica de apenas uma linha.

Exercício 5: Faça uma versão do seu programa usando `critical`. Novamente, sua utilização não incorre em perda de desempenho e os resultados são praticamente equivalentes a `atomic`.

Neste momento você deve ter obtido um programa com desempenho ao menos cerca de 50% mais rápido que o programa original. Mais importante, seu programa agora é muito mais simples de ler (e escrever) do que usando diretamente `std::thread`. Na próxima aula veremos como simplificar ainda mais estes códigos usando construções de alto nível do OpenMP.

Exercício extra: Faça uma comparação de desempenho cuidadosa entre suas 4 implementações levando em conta

1. Número de threads usadas
2. Número de iterações

Exercício extra: Adicione ao exercício anterior uma versão SIMD do calculo_pi sequencial.