Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 6 - Escopo de variáveis e regiões críticas
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

Esta tarefa tem como objetivo entender paralelismo com OpenMP, o impacto das cláusulas de escopo e como a ausência delas pode causar conflitos nas variáveis devido a condição de corrida. Para isso implementaremos um algoritmo estocástico para estimativa do número π usando o método de Monte Carlo.

Inicialmente, a tarefa propõe a paralelização do algoritmo utilizando a diretiva `#pragma omp parallel for`. No entanto, essa abordagem incorreta pode resultar em comportamentos inesperados devido ao compartilhamento inadequado de variáveis entre as threads, ocasionando condições de corrida (race conditions). O relatório analisa os motivos que levam a esse resultado incorreto e propõe uma reestruturação do código utilizando as diretivas `#pragma omp parallel` em conjunto com `#pragma omp for`, além da aplicação das cláusulas private, firstprivate, lastprivate, shared e default(none).

2 Metodologia

O método estocástico de Monte Carlo para estimar o valor de π é uma técnica probabilística que usa números aleatórios para resolver um problema matemático. Para estimar π utilizaremos um circulo de raio 1 circunscrito em um quadrado de lado 2 como mostrado na figura 1:

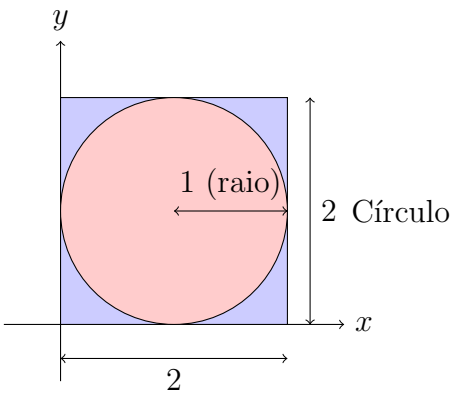

Figura 1: Circulo de raio 1 circunscrito em um quadrado de lado 2

Em seguida geraremos muitos pares (x,y) com valores aleatórios entre 0 e 2, ou seja, dentro da área do quadrado, a proporção de pontos encontrados dentro do círculo em relação ao total gerado se aproxima da razão entre as áreas:

$$\frac{\text{Pontos no Círculo}}{\text{Total de Pontos}} \approx \frac{\pi}{4}$$

Então para estimarmos o valor de π pelo método de Monte Carlo temos:

$$\pi \approx 4 \times \frac{\text{Pontos no Círculo}}{\text{Total de Pontos}}$$

Assim, a base de nosso código será composto por um laço de repetição que gerará n pares aleatórios (x,y) e testará se os pontos estão dentro ou não do circulo. Teremos uma variável chamada `hit` que servirá de contador dos pontos encontrados dentro da circunferência e os valores para x e y serão gerados aleatoriamente usando a função `rand` para geração e `srand` para inicialização da semente.

O código terá duas versões, uma com a paralelização incorreta devido ao mal compartilhamento das variáveis e um segundo com os ajustes necessários. Para a paralelização utilizaremos a biblioteca `OpenOMP` a qual na versão incorreta foi utilizado apenas a diretiva `#pragma omp parallel for`; já na segunda foram utilizadas as diretivas `#pragma omp parallel` e `#pragma omp for` em conjunto com as clausulas para as variáveis. Em ambas as versões o numero de pontos (n) gerados foi de 1.000.000.000.

Por fim, temos nosso código incorreto:

```
int main() {
   int hit = 0;

   srand(time(NULL));

   #pragma omp parallel for
   for (int i = 0; i < NUM_DOTS; i++) {
      double x = (double)rand() / RAND_MAX;
      double y = (double)rand() / RAND_MAX;

      if (x*x + y*y <= 1.0) {
         hit++;
      }
   }

   double pi = 4.0 * hit / NUM_DOTS;
   printf("Pi estimado (incorreto): %f\n", pi);

   return 0;
}
```

E o código com os ajustes:

```
int main() {

int hit = 0;
   unsigned int seeds[omp_get_max_threads()];

   for (int i = 0; i < omp_get_max_threads(); i++) {
      seeds[i] = time(NULL) + i;
   }

   #pragma omp parallel default(none) shared(hit) private(seeds)
   {
```

```
        int tid = omp_get_thread_num();
        unsigned int seed = seeds[tid];
        int hit_priv = 0;

        #pragma omp for
        for (int i = 0; i < NUM_PONTOS; i++) {
           double x = 2.0 * rand_r(&seed) / RAND_MAX - 1.0; // x entre -1 e 1
           double y = 2.0 * rand_r(&seed) / RAND_MAX - 1.0; // y entre -1 e 1

           if (x*x + y*y <= 1.0) {
              hit_priv++; // Contagem local (sem condição de corrida)
           }
        }

        #pragma omp critical
        {
           hit += hit_priv;
        }
    }
    double pi = 4.0 * (double)hit / NUM_PONTOS;
    printf("Estimativa de Pi (com variável local + critical): %f\n", pi);

    return 0;
}
```

3 Resultados

Com relação aos resultados obtidos, observamos que a primeira versão do código apresenta um valor incorreto para π, estimado em aproximadamente 3,088775. Esse resultado se deve ao fato de que, ao paralelizar diretamente o laço com `#pragma omp parallel for`, a variável global `hit` passa a ser acessada e modificada simultaneamente por múltiplas threads. Isso gera uma condição de corrida (*race condition*), uma vez que operações como incremento (`hit++`) não são atômicas, ou seja, podem ser interrompidas no meio da execução, resultando em valores errôneos quando acessadas por múltiplas threads simultaneamente..

Para ilustrar, considere o trecho de código abaixo:

```
if (x*x + y*y <= 1.0) {
        hit++;
}
```

Suponha que, em um dado instante da execução, o valor armazenado em `hit` seja 6. Se duas threads acessarem essa variável ao mesmo tempo, ambas podem ler o valor 6. A primeira thread soma 1 e armazena 7, mas a segunda, que também havia lido 6, sobrescreve o valor com outro 7. Assim, perdemos uma contagem válida, e o valor final será incorreto (deveria ser 8, mas será 7).

Esse tipo de paralelização, sem o devido controle de acesso às variáveis compartilhadas, compromete a precisão do resultado. No caso da estimativa de π, a inconsistência resultou em um valor subestimado (3,088775), evidenciando a necessidade de mecanismos como `reduction`, seções críticas ou o uso de clausulas para garantir a correção.

A segunda versão do código implementa as correções necessárias para que a estimativa de π funcione corretamente em ambiente paralelo. A primeira modificação relevante foi o tratamento

das sementes do gerador de números pseudoaleatórios: foi criada uma semente distinta para cada *thread*. Isso garante que cada *thread* utilize sequências diferentes de números aleatórios, evitando sobreposição e assegurando maior aleatoriedade na amostragem.

Em seguida, foi aplicada a diretiva `#pragma omp parallel default(none) shared(hit) private(seed` que define o início da região paralela. A cláusula `default(none)` exige que todas as variáveis utilizadas dentro do bloco tenham seus escopos explicitamente definidos, o que aumenta a clareza e evita erros. A variável `hit` é marcada como `shared`, pois será utilizada por todas as *threads* para computar o total de acertos. Já `seeds` é declarada como `private`, garantindo que cada *thread* tenha sua própria cópia da variável.

Dentro da região paralela, utiliza-se a diretiva `#pragma omp for` para dividir o laço `for` entre as *threads*, distribuindo a carga de trabalho. Além disso, emprega-se a diretiva `#pragma omp critical` para proteger o trecho onde cada *thread* soma sua contagem local (`hit_priv`) à variável compartilhada `hit`. Essa seção crítica evita condições de corrida, garantindo a consistência do resultado final. Com isso, a estimativa de π obtida foi significativamente mais precisa, chegando ao valor aproximado de $\pi \approx 3{,}141572$, com quatro casas decimais corretas.

Com relação às cláusulas, já discutimos sobre `default(none)`, `shared` e `private`, que foram efetivamente utilizadas no código. Sabemos que `default(none)` obriga o programador a especificar explicitamente o escopo de cada variável, promovendo um controle mais rigoroso e seguro do paralelismo. A cláusula `shared` indica que uma variável é compartilhada entre todas as threads, como no caso da variável `hit`, cuja soma dos acertos deve ser consolidada ao final da execução. Por outro lado, `private` define que cada thread terá sua própria cópia da variável, isolando modificações — útil, por exemplo, no vetor `seeds`, onde cada thread possui uma semente independente para geração de números aleatórios.

Além dessas, é importante compreender o papel das cláusulas `firstprivate` e `lastprivate`. A cláusula `firstprivate` atribui a cada thread uma cópia privada de uma variável, inicializada com o valor presente antes da região paralela. Ela seria aplicável, por exemplo, à variável `seed`, caso desejássemos que todas as threads partissem de uma mesma semente inicial (o que, neste contexto, não seria apropriado, pois geraria sequências idênticas de números pseudoaleatórios, comprometendo a aleatoriedade da simulação). Já a cláusula `lastprivate` garante que, ao final da região paralela, a variável correspondente receba o valor da última iteração lógica do laço — como se o loop tivesse sido executado sequencialmente. Embora essa cláusula não tenha sido usada no código em questão, ela poderia ser útil caso quiséssemos, por exemplo, registrar o último ponto gerado na simulação para análise ou depuração. No entanto, como a estimativa de π depende da contagem acumulada e não de valores específicos do final do loop, o uso de `lastprivate` não se faz necessário neste caso.

4 Conclusão

A realização desta tarefa permitiu compreender, na prática, os impactos do escopo de variáveis e das condições de corrida na programação paralela utilizando OpenMP. Foi possível observar que a simples paralelização de um laço, sem o devido controle sobre as variáveis compartilhadas, pode levar a resultados incorretos e inconsistentes, como no caso da estimativa errada de π.

A reestruturação do código utilizando diretivas adequadas, como `#pragma omp parallel`, `#pragma omp for` e `#pragma omp critical`, juntamente com o uso apropriado de cláusulas de escopo (`private`, `shared`, `default(none)`), foi fundamental para garantir a corretude da execução paralela. Além disso, o uso de sementes distintas para a geração de números aleatórios assegurou maior diversidade na amostragem e contribuiu para a precisão do resultado.

Concluímos, portanto, que compreender e aplicar corretamente os conceitos de escopo de variáveis e controle de acesso a regiões críticas é essencial para o desenvolvimento de algoritmos paralelos eficientes e corretos. A experiência prática adquirida nesta tarefa será valiosa para futuras implementações que envolvam concorrência e paralelismo.