Universidade Federal do Rio Grande do Norte
Departamento de Engenharia da Computação e Automação
DCA3703 - Programação Paralela
Tarefa 10 - Mecanismos de sincronização: atomic e reduction
Aluno: Daniel Bruno Trindade da Silva

1 Introdução

O valor de π (pi) é uma constante matemática fundamental, frequentemente utilizada em cálculos científicos e de engenharia. Uma forma interessante e didática de estimá-lo é por meio do método de Monte Carlo, que utiliza experimentos estocásticos baseados em geração de números aleatórios para simular eventos e calcular probabilidades. Neste experimento, implementamos três versões paralelas de um estimador de π utilizando a API OpenMP, explorando diferentes mecanismos de sincronização para controlar o acesso concorrente às variáveis compartilhadas entre threads.

A primeira versão emprega a diretiva `#pragma omp critical`, que garante exclusão mútua, mas pode introduzir gargalos de desempenho. A segunda substitui essa abordagem por `#pragma omp atomic`, adequada para operações simples como incrementos. Por fim, a terceira versão utiliza a cláusula `reduction`, que permite ao compilador otimizar a operação de soma paralela com contadores locais, sendo geralmente mais eficiente. Com isso, buscamos comparar o impacto dessas técnicas tanto em termos de desempenho quanto de produtividade na programação paralela.

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, ambos posicionados no centro do eixo de coordenadas como mostrado na figura 1:

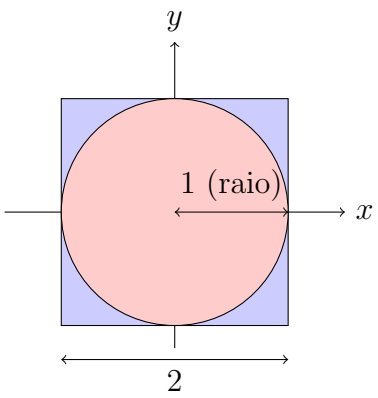

Figure 1: Circulo de raio 1 circunscrito em um quadrado de lado 2

Em seguida geraremos muitos pares (x,y) com valores aleatórios entre -1 e 1, 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. Ao final utilizaremos a proporção de acertos pelo número de pares (x, y) para estimarmos o valor de π.

Nesta tarefa, foram desenvolvidas três versões distintas do código, cada uma utilizando um mecanismo diferente de sincronização:

- A **primeira versão** emprega a diretiva `#pragma omp critical`, que protege a região crítica onde a variável global é atualizada, garantindo que apenas uma thread por vez execute essa operação.
- A **segunda versão** utiliza a diretiva `#pragma omp atomic`, que oferece uma forma mais leve de sincronização para operações simples, como a adição à variável global.
- A **terceira versão** adota a cláusula `reduction`, permitindo que cada thread mantenha um contador local, cuja soma final é realizada automaticamente pelo OpenMP, eliminando a necessidade de sincronização explícita.

Sendo assim, a região paralela de nosso código ficou da seguinte forma:

Versão com `#pragma omp critical`

```
// Região paralela
#pragma omp parallel default(none) shared(hit, 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_DOTS; i++){
        double x = 2.0 * rand_r(&seed) / RAND_MAX - 1.0;
        double y = 2.0 * rand_r(&seed) / RAND_MAX - 1.0;

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

    #pragma omp critical
    {
        hit += hit_priv;
    }
}
```

Versão com `#pragma omp atomic`

```
// Região paralela
#pragma omp parallel default(none) shared(hit, 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_DOTS; i++){
  double x = 2.0 * rand_r(&seed) / RAND_MAX - 1.0;
  double y = 2.0 * rand_r(&seed) / RAND_MAX - 1.0;

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

 #pragma omp atomic
  hit += hit_priv;
}
```

Versão com `reduction`

```
#pragma omp parallel default(none) shared(seeds) reduction(+:hit)
{
 int tid = omp_get_thread_num();
 unsigned int seed = seeds[tid];

 #pragma omp for
 for(int i = 0; i < NUM_DOTS; 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++;
  }
 }
}
```

3 Resultados

Os programas foram executados em um mesmo ambiente computacional, utilizando 1.000.000.000 (um bilhão) de pontos aleatórios para a estimativa de π. A Tabela 1 apresenta os tempos de execução observados para cada versão do código, com diferentes mecanismos de sincronização.

Table 1: Tempo de execução para cada versão do código

Versão	Tempo de Execução (s)
`omp atomic`	2,0597
`omp critical`	2,0739
`omp reduction`	2,1183

Embora a expectativa teórica fosse de que a versão com `reduction` apresentasse o melhor desempenho — devido ao seu alto nível de otimização interna e eliminação de diretivas de sincronização explícitas —, os resultados demonstraram que a versão com `atomic` foi ligeiramente mais rápida, seguida de perto pela versão com `critical`, e por último a versão com `reduction`.

Essas pequenas variações podem ser atribuídas a diversos fatores, como:

- A sobrecarga de criação e gerenciamento dos contadores locais na cláusula `reduction`, que pode não compensar os ganhos esperados em determinados contextos.
- O uso eficiente da operação atômica (`atomic`) em hardware moderno, que pode apresentar desempenho comparável ou até superior a outras técnicas, especialmente quando a operação protegida é simples.

Apesar das diferenças de tempo serem mínimas (todas próximas de 2 segundos), os resultados mostram que nenhuma técnica é absolutamente superior em todos os contextos, reforçando a importância de avaliar o desempenho prático de cada abordagem dentro do cenário de aplicação específico.

4 Conclusão

A realização deste experimento permitiu explorar, na prática, diferentes estratégias de sincronização em programação paralela com OpenMP. Através da implementação de três versões do estimador de π com os mecanismos `critical`, `atomic` e `reduction`, foi possível comparar o impacto de cada abordagem no desempenho da aplicação.

Os resultados demonstraram que, embora a cláusula `reduction` seja frequentemente considerada a mais eficiente devido à sua capacidade de otimização interna, a versão utilizando `atomic` apresentou o menor tempo de execução no contexto testado. Isso evidencia que o desempenho real pode variar de acordo com fatores como o tipo de operação sincronizada, a carga de trabalho, o número de threads e as características do hardware utilizado.

Do ponto de vista de produtividade, a cláusula `reduction` oferece uma sintaxe mais limpa e intuitiva, sendo ideal para operações de agregação simples. Já as diretivas `critical` e `atomic` oferecem maior flexibilidade, mas requerem atenção redobrada ao risco de condições de corrida e ao impacto no desempenho.

Em resumo, a escolha do mecanismo de sincronização mais adequado deve considerar tanto a natureza da operação concorrente quanto o equilíbrio entre legibilidade do código e eficiência da execução. Experimentos como este reforçam a importância da análise empírica para embasar decisões de projeto em sistemas paralelos.