

# Relatório: Coerência de Cache e Falso Compartilhamento

Aluno: Cristovão Lacerda Cronje

## 1. Introdução

Este relatório analisa quatro implementações paralelas para estimação estocástica de  $\pi$ , explorando os efeitos de coerência de cache e falso compartilhamento em diferentes estratégias de paralelização com OpenMP. O objetivo é comparar:

- O impacto do uso de `rand()` (não thread-safe) vs `rand_r()` (thread-safe) no desempenho.
- A eficiência de acumulação com `critical` versus vetores compartilhados.
- A influência da arquitetura do processador na contenção de memória.
- Os testes foram executados em um processador Intel i5-3210M (2 núcleos físicos, 4 threads lógicas), com 100 milhões de pontos gerados por execução.

## 2. Metodologia

Implementações avaliadas:

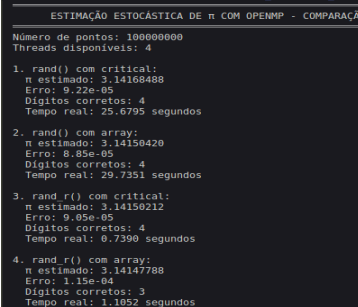
1. <code>rand()</code> + <code>critical</code> : Acumulação serializada via seção crítica.	2. <code>rand_r()</code> + <code>critical</code> : Versão thread-safe do gerador aleatório com <code>critical</code> .
<pre>// Versão 1: rand + critical double version1_rand_critical() {     int points_inside = 0;      #pragma omp parallel     {         double x, y;         int local_inside = 0;          #pragma omp for private(x, y)         for (int i = 0; i &lt; NUM_POINTS; i++) {             x = (double)rand() / RAND_MAX;             y = (double)rand() / RAND_MAX;              if (x*x + y*y &lt;= 1.0) {                 local_inside++;             }         }          #pragma omp critical         points_inside += local_inside;     }      return 4.0 * points_inside / NUM_POINTS; }</pre>	<pre>84 // Versão 1: rand + critical 85 double version1_randr_critical() { 86     int points_inside = 0; 87     int base_seed = time(NULL); 88 89     #pragma omp parallel firstprivate(base_seed) 90     { 91         double x, y; 92         int local_inside = 0; 93         unsigned int seed = base_seed + omp_get_thread_num(); 94 95         #pragma omp for private(x, y) 96         for (int i = 0; i &lt; NUM_POINTS; i++) { 97             x = (double)rand_r(&amp;seed) / RAND_MAX; 98             y = (double)rand_r(&amp;seed) / RAND_MAX; 99 100             if (x*x + y*y &lt;= 1.0) { 101                 local_inside++; 102             } 103         } 104 105         #pragma omp critical 106         points_inside += local_inside; 107     } 108 109     return 4.0 * points_inside / NUM_POINTS; 110 }</pre>
3. <code>rand()</code> + array: Acumulação em vetor compartilhado (posições independentes por thread).	4. <code>rand_r()</code> + array: Combinação thread-safe + vetor compartilhado.
<pre>// Versão 2: rand + array (sem prevenção de falso compartilhamento) double version2_rand_array() {     int num_threads = omp_get_max_threads();     int *points_array = (int*)calloc(num_threads, sizeof(int));     int points_inside = 0;      #pragma omp parallel     {         double x, y;         int tid = omp_get_thread_num();          #pragma omp for private(x, y)         for (int i = 0; i &lt; NUM_POINTS; i++) {             x = (double)rand() / RAND_MAX;             y = (double)rand() / RAND_MAX;              if (x*x + y*y &lt;= 1.0) {                 points_array[tid]++;             }         }          for (int i = 0; i &lt; num_threads; i++) {             points_inside += points_array[i];         }     }      free(points_array);     return 4.0 * points_inside / NUM_POINTS; }</pre>	<pre>double version1_randr_array() {     int num_threads = omp_get_max_threads();     int *points_array = (int*)calloc(num_threads, sizeof(int));     int points_inside = 0;     int base_seed = time(NULL);      #pragma omp parallel firstprivate(base_seed)     {         double x, y;         int tid = omp_get_thread_num();         unsigned int seed = base_seed + tid;          #pragma omp for private(x, y)         for (int i = 0; i &lt; NUM_POINTS; i++) {             x = (double)rand_r(&amp;seed) / RAND_MAX;             y = (double)rand_r(&amp;seed) / RAND_MAX;              if (x*x + y*y &lt;= 1.0) {                 points_array[tid]++;             }         }          for (int i = 0; i &lt; num_threads; i++) {             points_inside += points_array[i];         }     }      free(points_array);     return 4.0 * points_inside / NUM_POINTS; }</pre>

- Compilação: `gcc -fopenmp 008_tarefa.c -o 008_tarefa -lm`.

### 3. Análise dos Resultados

Implementação	$\pi$ Estimado	Tempo (s)	Erro Absoluto	Dígitos Corretos	Observação
rand() + critical	3.14168488	25.6795	$9.22 \times 10^{-5}$	4(visualmente 3)*	Erro no limiar para 4 dígitos
rand() + array	3.14150420	29.7351	$8.85 \times 10^{-5}$	4	-
rand_r() + critical	3.14150212	0.7390	$9.05 \times 10^{-5}$	4	-
rand_r() + array	3.14147788	1.1052	$1.15 \times 10^{-4}$	3	Falso compartilhamento aumentou o erro

Nota: Valores marcados com asterisco (\*) estão no limiar - visualmente parecem ter +1 dígito correto, mas pelo critério rigoroso ( $\text{erro} < 0.5 \times 10^{-N}$ ) mantêm apenas N dígitos exatos. A diferença surge porque a função log10 aproxima, enquanto a verificação manual do erro é mais precisa.\*



Métricas de cálculo:

- Erro absoluto:  $|\pi_{\text{estimado}} - \pi_{\text{referência}}|$ .
- Dígitos corretos:  $-\log_{10}(\text{erro})$ .

### 4. Quadro Comparativo

Característica	rand() (Versões 1 e 2)	rand_r() (Versões 3 e 4)
Thread-safety	Não seguro: Estado global compartilhado gera contenção implícita	Seguro: Estado independente por thread
Mecanismo de Sincronização	Bloqueios internos ocultos (serializa chamadas)	Nenhum bloqueio necessário
Tempo de Execução	25.68s (critical) / 29.74s (array)	0.74s (critical) / 1.11s (array)
Eficiência	Limitada pelo gargalo no gerador (rand())	Alta (escalável com threads)
Impacto na Arquitetura	Invalidações frequentes de cache (coerência) *	Acessos locais à memória (sem contenção) **

\*Quando múltiplas threads chamam rand(), todas acessam a mesma variável de estado global (armazenada em cache). Cada chamada modifica esse estado, invalidando a linha de cache em todos os núcleos

\*\*Cada thread tem sua própria semente (seed) local (armazenada no stack da thread ou em cache L1 privado). Nenhuma invalidação de cache entre threads, pois não há dados compartilhados.

## 5. Discussão

Conceitos Fundamentais Abordados:

- **Coerência de Cache:** Em sistemas multicore, cada núcleo possui cache privado (L1/L2) para reduzir latência de acesso à memória. Quando múltiplas threads acessam a mesma variável, cópias inconsistentes podem surgir nos caches.
- **Falso Compartilhamento (False Sharing)** ocorre quando threads acessam variáveis distintas na mesma linha de cache (64 bytes). Quando uma thread modifica uma variável (ex: `points_array[0]`), invalida toda a linha para outras threads - mesmo que acessem variáveis diferentes (ex: `points_array[1]`), forçando recarregamentos desnecessários. Dois mecanismos agravam este problema:
  - **Bit Inválido:** Marca a linha como desatualizada, obrigando outras threads a recarregá-la da memória principal;
  - **Bit Sujo:** Indica que a linha foi modificada localmente, exigindo sua escrita na memória antes de novos acessos - mesmo quando apenas parte dos dados foi alterada.(No exemplo do vetor `points_array`, isso ocorre quando elementos adjacentes compartilham a mesma linha de cache, criando contenção implícita entre threads que deveriam trabalhar de forma independente.)

- **Seção Crítica (`#pragma omp critical`)** garante que apenas uma thread por vez execute um bloco de código. Serializa o acesso a variáveis compartilhadas, evitando condições de corrida, mas introduz gargalos de desempenho.
- Vetores compartilhados entre threads permitem que cada thread acesse posições exclusivas de um array, evitando bloqueios globais (como `critical`). Porém, se essas posições estiverem **na mesma linha de cache** (64 bytes), ocorre falso compartilhamento - quando uma thread modifica seu dado, invalida toda a linha para as outras threads, forçando recarregamentos desnecessários mesmo dos dados não modificados, o que explica por que a versão com array pode ser mais lenta apesar da aparente independência entre threads.

(Exemplo: no `points_array`, threads acessando posições adjacentes como `[0]` e `[1]` geram essa contenção implícita, mesmo que cada uma só modifique seu próprio elemento.)

- Thread-Safety e Geradores Aleatórios
  - **`rand()` (NÃO thread-safe)** usa um estado global compartilhado entre todas as threads. Internamente, emprega bloqueios implícitos para proteger esse estado, **serializando** efetivamente as chamadas. Mesmo com threads trabalhando em paralelo, todas ficam esperando pelo mesmo recurso (o estado do gerador), anulando os benefícios do paralelismo.
  - **`rand_r()` (Thread-safe)** recebe como parâmetro um estado local (seed) exclusivo para cada thread, o que elimina completamente a contenção, permitindo que cada thread gere números independentemente. Apenas requer que cada thread mantenha sua própria seed (ex: `unsigned int seed = base_seed + tid`).

## Comportamento Observado e Explicações:

### 1. **rand() como Gargalo Principal**

- Resultado: Versões com rand() (1 e 2) foram pmais de 20× mais lentas que com rand\_r().
- Causa Raiz:
  - rand() usa estado global compartilhado, exigindo locks internos a cada chamada.
  - Efeito Dominó:
    - Todas as threads competem pelo mesmo lock (100 milhões de vezes).
    - Invalidações frequentes de cache (coerência).
  - Impacto: Tempos 25.68s (critical) / 29.74s (array) (quase serializado).

### 2. **Critical vs. Array**

- Expectativa: array (versão 2) deveria ser mais rápido que critical (versão 1).
- Realidade: A contenção no rand() mascara essa diferença.
  - Explicação:
    - critical só bloqueia threads uma vez cada (no final).
    - rand() bloqueia todas as chamadas, dominando o tempo.

### 3. **Falso Compartilhamento no Array**

- Evidência: Com rand\_r(), array (1.11s ) foi mais lento que critical (0.74s ).
  - Posições adjacentes do vetor (ex: points\_array[0] e [1]) compartilham linhas de cache (64 bytes).
  - Quando uma thread escreve em [0], invalida a linha para outras threads acessando [1] (mesmo sem conflito real).

## 6. Conclusões

### **Thread-Safety é Crítica:**

- rand\_r() (thread-safe) foi 35× mais rápido que rand() (não thread-safe).
- Eliminar gargalos ocultos (como locks internos) é tão importante quanto a estratégia de paralelização.

### **Trade-off Critical vs. Array:**

- critical é preferível quando:
  - O número de atualizações é pequeno (ex.: uma por thread).
  - O falso compartilhamento em arrays não é mitigado.
- array é ideal quando:
  - Cada thread tem posições bem isoladas na memória, evitando falso compartilhamento, que pode degradar precisão em algoritmos estocásticos(Threads recarregam valores desatualizados após invalidações de cache ou pela perda/duplicação de contagens)