Introdução de OpenMP — Parte 1 - Visão Geral

Informações do Autor

- **Nome:** Roussian Gaioso
- **Data de atualização**: 29/09/2025

1) O que é OpenMP e por que usar

- OpenMP é um padrão para programar com threads em linguagens como C/C++ e Fortran. Ele começou em 1997 com foco em paralelizar loops de forma simples; hoje (5.2) abrange tarefas, vetorização, hierarquias de memória e até GPUs.
- Nota:
  - Ninguém usa “o padrão inteiro”: aprende-se um núcleo comum e consulta-se o resto quando preciso.

2) Revisão - Conceitos importantes: processos, threads e memória

Por que isso importa quando programamos com threads?

1) Quem enxerga o quê: shared vs. private

- Em um processo, todas as threads compartilham o mesmo espaço de endereçamento; o que muda é onde o dado vive:
  - **Stack** → pertence à thread que está executando aquela função; é privado por natureza.
  - **Heap** → pertence ao processo; é compartilhado por todas as threads. Em OpenMP isso vira uma regra prática: variáveis na stack tendem a ser privadas; variáveis no heap tendem a ser compartilhadas.
- Consequência: sempre que duas ou mais threads podem escrever no mesmo endereço sem sincronização (caso típico no heap/global), há possibilidade de *condição de corrida* → programa inválido.

```c
// Exemplo: CORRIDA DE DADOS (NÃO FAÇA)
#include <omp.h>
int total = 0;   // área de dados/heap -> compartilhada
int main(void){
  #pragma omp parallel num_threads(4)
  {
    total++;  // ++ concorrência sem ordem -> condição de corrida
  }
}
```

---

2) Tempo de vida do dado

- A stack dura só enquanto a função está ativa; ao sair, as variáveis “somem”.
- Passar ponteiros para memória de stack para fora do seu escopo leva a uso de memória inválida.
- Com várias threads, isso é ainda mais complicado porque várias stacks coexistem.

```c
// Padrão problemático (conceitual): ponteiro para a stack de uma
int *escape = NULL;
#pragma omp parallel
{
  // 'local' vive na stack de um thread
  int local = omp_get_thread_num();
  // guardar &local fora é bug (vida curta)
  escape = &local;
}  // aqui 'local' já não existe mais
```

---

3) Ordem e visibilidade

- Em CPUs reais, existem caches e reordenações: pode haver cópias temporárias do mesmo dado simultaneamente (registradores, L1/L2/DRAM).
- O sistema garante coerência ao longo do tempo, mas não garante que todas as threads veem imediatamente a mesma versão.
- Isso é o tal modelo de memória relaxado. Sem sincronização, o programa pode dar resultados diferentes entre execuções.
- É por isso que até o “Hello, parallel!” imprime linhas em ordens diferentes a cada execução: as threads são concorrentes e não ordenadas salvo pontos explícitos de sincronização.

```c
// Hello paralelo:
// ordem varia a cada execução (não há sincronização)
#include <stdio.h>
#include <omp.h>
int main(void){
  #pragma omp parallel
  {
    // declarado DENTRO da região -> stack -> privado
    int id = omp_get_thread_num();
    printf("hello %d ", id);
    printf("world %d\n", id);
  }
}
```

> A variável `id` foi declarada dentro da região paralela → está na stack da thread e, portanto, é privada (cada thread tem a sua).
>
> Já a saída na tela é compartilhada; sem ordem, as mensagens embaralham.

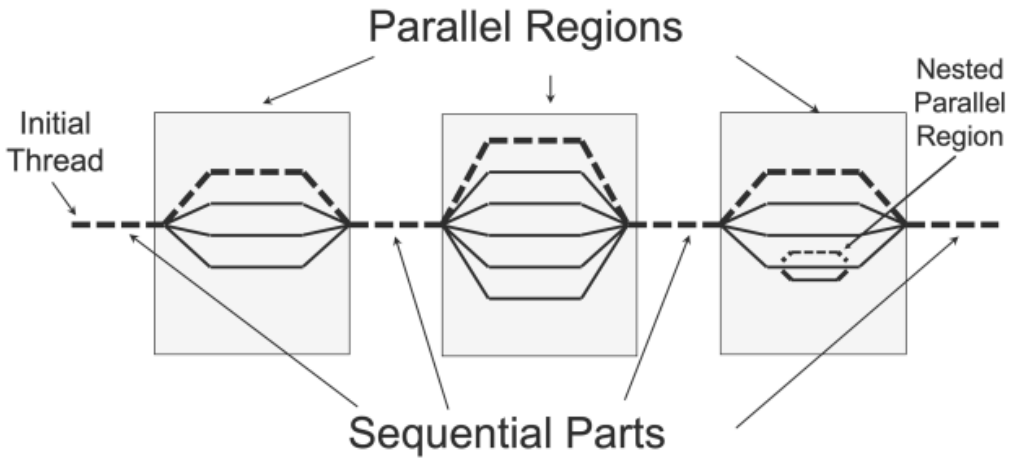
---

6) Desempenho e localidade

- Como caches escondem latências, manter cada thread trabalhando em dados que ela "acabou de usar" ajuda a performance.
    - Mas isso só funciona bem se entender onde os dados vivem (stack/heap) e como são compartilhados — do contrário, você pode forçar o hardware a sincronizar versões em caches o tempo todo.

Modelo Fork–join

- O programa começa sequencialmente com uma thread (a "thread master").
- Quando a thread master entrar numa região paralela (`#pragma omp parallel`), essa thread faz fork de um *time de threads* (2, 4, 8, ...).
- Todas as threads do time executam o mesmo bloco de código (cada uma com sua pilha/stack).
- Ao final do bloco, existe um join implícito: as threads terminam (ou são devolvidas ao *pool* do runtime) e a execução volta a ser sequencial.




```
sequencial ── fork ──▶ [ T0  T1  T2  T3 ] ── join ──▶ sequencial
```

> Pense na região paralela como “abrir uma sala” onde várias threads entram, trabalham, e todas saem juntas antes de seguir em frente.

---

1) Quem é quem nesse modelo

- **Thread master**: a única thread antes da primeira região paralela.
- **Time de threads**: conjunto criado pela diretiva `parallel`. Uma delas é a master (normalmente `omp_get_thread_num()==0`).
- **Master vs. outras threads**:
  - `#pragma omp master` → só a master executa aquele bloco (sem barreira no fim).
  - `#pragma omp single` → exatamente uma thread qualquer executa, e por padrão há barreira no fim (pode remover com `nowait`).

---

2) O que acontece com variáveis

- Stack é privado por thread.
- Heap é compartilhado pelo processo.
- Ao entrar na região paralela, cada thread tem sua própria pilha
  - Variáveis locais declaradas dentro do bloco paralelo são privadas.
- Dados globais/estáticos ou alocados no heap são visíveis por todas → se mais de uma thread escreve no mesmo endereço, é preciso sincronizar (atomic/critical/reduction).

---

3) Barreira implícita e ordem

O que é uma barreira implícita

> É um ponto de encontro automático que o OpenMP insere no fim de certos blocos de código.
>
> **Significa**: ninguém avança para o código seguinte até todas as threads chegarem a esse ponto.
>
> Não precisa "pedir": ela já existe por definição do construto.

Onde existe barreira implícita (por padrão)

1. **Fim de uma região paralela**
    - Construto: `#pragma omp parallel { ... }`
    - Ao sair das chaves, todas as threads terminaram o bloco.
    - A execução volta para o trecho sequencial só depois disso.
2. **Fim dos construtos de divisão de trabalho** (*worksharing*):
    - `#pragma omp for`
    - `#pragma omp sections`
    - `#pragma omp single` Em todos eles, **ao final do bloco**, há **barreira implícita** por padrão.

```
> Por enquanto, ignore qualquer exceção/variação. Foque nesta regr
```

O que essa barreira garante (noção de "ordem")

- **Entre blocos**: tudo que vem depois do bloco com barreira implícita enxerga o que foi feito dentro dele por todas as threads.
- **Dentro do bloco**: a ordem entre threads durante a execução não é garantida (elas correm em paralelo).
- A **ordem** só fica "estabelecida" **no final** (no ponto da barreira).

---

4) Exemplos

> **Linha de compilação**: // gcc -O3 -fopenmp exemplo.c -o a

1. Região paralela "abre/fecha"

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
  puts("Antes (sequencial)");

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf("Dentro: T%d\n", id);     // ordem livre entre threads
  } // <-- barreira implícita aqui (todas terminaram)

  puts("Depois (sequencial)");       // só executa após TODAS saír
}
```

O `puts("Depois")` sempre acontece após todas as linhas “Dentro: T...”.

---

2. Produção → Consumo (duas fases) com `for`

```c
#include <omp.h>
#include <stdio.h>
#define N 100000

int main(void) {
  static int A[N], B[N];

  #pragma omp parallel
  {
    #pragma omp for
    for (int i = 0; i < N; i++) {
      A[i] = i;                  // Fase 1: produzir A
    } // <-- barreira implícita: tudo de A pronto aqui

    #pragma omp for
    for (int i = 0; i < N; i++) {
      B[i] = 2 * A[i];   // Fase 2: consumir A com segurança
    } // <-- barreira implícita fecha o 2º for
  }
  printf("B[123]=%d\n", B[123]);
}
```

A segunda fase só começa “valendo” depois que a primeira acabou para todas as threads (isso porque tem uma barreira implícita no fim do `for`).

---

3. Inicialização única antes do trabalho

```c
#include <omp.h>
#include <stdio.h>

void inicializa_config(void) { /* ... */ }

int main(void) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      inicializa_config();      // apenas uma thread executa
    } // <-- barreira implícita: todas só seguem após a inicializa

    #pragma omp for
    for (int i = 0; i < 1000; i++) {
      // usa a configuração já pronta
    }
  }
}
```

Mesmo só uma thread executando a inicialização, todas esperam o término no fim do `single`.

---

4. Visualizando o Fork-Join

```c
#include <stdio.h>
#include <omp.h>

int main(void) {
  puts("Antes (sequencial)");

  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf("Dentro (parallel): thread %d\n", id);
  } // <- join implícito aqui

  puts("Depois (sequencial)");
  return 0;
}
```

Saída típica (ordem pode variar):

```
Antes (sequencial)
Dentro (parallel): thread 2
Dentro (parallel): thread 0
Dentro (parallel): thread 1
Dentro (parallel): thread 3
Depois (sequencial)
```

---

5) Custo de criar/fechar os times

- Criar um time de threads custa tempo (fork) e fechá-lo também (join).
- Se abre e fecha várias regiões paralelas em sequência, paga esse custo repetidamente.
- A prática melhor é abrir uma vez, fazer várias etapas dentro do mesmo `parallel` e fechar no final.

Exemplo - muitos forks/joins em sequência

```c
// Etapa 1
#pragma omp parallel
{
  etapa1_por_thread();// cada thread faz sua parte
}                     // join aqui

// Etapa 2
#pragma omp parallel
{
  etapa2_por_thread();    // novo fork + novo join
}
```

Exemplo - um fork, várias etapas:

```c
#pragma omp parallel
{
  etapa1_por_thread();    // trabalho paralelo (fase 1)

  // Se existir uma preparação curta
  // que deve ocorrer uma única vez
  // e só depois todos prosseguirem,
  // use 'single' (tem barreira implícita ao final)
  #pragma omp single
  {
    // executado por uma thread;
    // todas aguardam ao final
    passo_unico_de_preparacao();
  }

  etapa2_por_thread();  // trabalho paralelo (fase 2)
} // join único aqui (fim da região paralela)
```

Por que isso é melhor?

- Menos overhead: gasta um fork/join em vez de vários.
- Layout mais simples em fases:
  - Dentro do `parallel`, pode-se organizar o trabalho em etapas;
  - quando precisa de um "ponto de encontro", o `single` já traz barreira implícita no final, garantindo que todos avancem juntos para a próxima fase (sem introduzir outros mecanismos de sincronização neste momento).

---

6) Dividindo o trabalho dentro da região paralela

- `parallel` só cria o time; não divide o trabalho automaticamente entre as threads.
  - Quando se escreve `#pragma omp parallel { ... }`, o OpenMP cria um time de threads e cada thread executa o código entre as chaves.
  - Se dentro desse bloco você coloca um `for` comum (sem OpenMP), todas as threads vão tentar rodar o mesmo loop inteiro — duplica-se o trabalho e ainda pode criar condição de corrida.
- Dentro de um bloco `#pragma omp parallel { ... }`, usa-se **construtos de divisão de trabalho (*worksharing*)** para repartir as tarefas entre as threads do time.
- **Worksharing = diretivas que repartem o serviço entre as threads.**
  - "Construto" em OpenMP é *diretiva* + *bloco* estruturado.
  - O código executado forma uma região.

    - Os construtos de *worksharing* são justamente os que partem o trabalho do bloco entre as threads.
    - Eles não criam threads; usam as threads criadas pelo `parallel`.
- **Sem worksharing, cada thread faz "tudo" daquele bloco.**

> A diferença prática é essa: com worksharing você diz "quem faz qual pedaço"; sem ele, todas fazem o mesmo código.

```
#pragma omp parallel
{
  // TODAS as threads executam este loop inteiro
  for (long i = 0; i < N; i++) {
    A[i] = f(i);
  }
}
```

Diretiva `for`

Use quando se tem muitas iterações semelhantes (ex.: varrer um vetor/matriz).

- O compilador/runtime distribui as iterações entre as threads.
- Há barreira implícita no final do `for`: todas as iterações terminam antes de seguir.

Exemplo

- Preenchendo um vetor

```
#pragma omp parallel
{
  #pragma omp for
  for (long i = 0; i < N; i++) {
    A[i] = f(i);   // cada thread recebe um subconjunto de índices
  }                // <- barreira implícita: A inteiro está pronto
}
```

Quando usar: loops grandes e regulares (mesma "cara" em cada iteração).

Diretiva `sections`

- Dentro de um `parallel`, `sections` distribui blocos distintos de código entre as threads do time.
- Cada bloco marcado com `#pragma omp section` é executado exatamente uma vez por uma thread (não necessariamente sempre a mesma).

- Não cria threads: quem cria é o `parallel`. O `sections` só reparte os blocos.
- Barreira implícita no final: ao sair do `sections`, todas as seções terminaram.

> Pense em “tarefas grandes e diferentes” (A, B, C...). Se forem centenas de tarefas iguais, prefira `for`.

Quando usar

- Tem poucas tarefas diferentes (ex.: pré-processar dados, ler arquivo, calcular métricas independentes).
- Cada tarefa não depende do resultado das outras durante a execução do bloco
    - A dependência pode existir depois e a barreira implícita já garante que todas terminaram antes de prosseguir.

Observações

- Se houver menos seções que threads, algumas threads ficam ociosas durante esse trecho — ok para poucas seções.
- Se houver mais seções que threads, algumas threads executarão mais de uma seção (em sequência).
- As seções compartilham a memória do processo (heap/globais): não escreva no mesmo endereço em duas seções ao mesmo tempo.

Exemplo 1

```c
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      // executado por uma thread
      passoA();
    }

    #pragma omp section
    {
      // executado por outra thread
      // ou pela mesma, se houver poucas threads
      passoB();
    }
  }  // <- barreira implícita: A e B concluídos aqui
}

// Daqui em diante, é seguro usar os resultados de A e B.
consome_resultados_de_A_e_B();
```

- `passoA()` e `passoB()` não dependem um do outro enquanto rodam.
- A barreira implícita garante que, ao sair do `sections`, ambos terminaram.

Exemplo 2 — três seções com naturezas diferentes

```c
#include <stdio.h>
#include <omp.h>

void ler_logs(void)         { printf("ler_logs()\n"); }
void construir_indice(void) { printf("construir_indice()\n"); }
void enviar_telemetria(void){ printf("enviar_telemetria()\n"); }

int main(void) {
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      { ler_logs(); }

      #pragma omp section
      { construir_indice(); }

      #pragma omp section
      { enviar_telemetria(); }
    } // <- barreira implícita: todas as sections terminaram aqui
  }
  puts("pos-sections");
  return 0;
}
```

Observações:

- Se tivesse 4 threads e 3 seções, 1 thread poderia ficar ociosa neste trecho.
- Se tivesse 2 threads e 3 seções, uma das threads executaria duas seções (em sequência).

> Use `sections` quando você tem poucas tarefas distintas que podem rodar em paralelo;
>
> Cada `section` roda uma vez em uma thread;
>
> A barreira implícita no final garante que todas concluíram antes de você seguir.

Diretiva `single`

- Dentro de um `parallel`, o bloco marcado com `#pragma omp single` é executado exatamente uma vez por uma das threads do time (não necessariamente a 0).

- Ao final do `single` existe uma barreira implícita: todas as threads esperam até que essa execução única termine, e só então seguem adiante.

> Pense no `single` como “faça isso uma única vez e garanta que todos só prossigam depois”.

Quando usar

- Inicializações/Setup antes do trabalho paralelo
  - Por exemplo, carregar configurações, abrir um arquivo, alocar/organizar estruturas compartilhadas.
- Operações seriais inevitáveis
  - Por exemplo, imprimir um cabeçalho, criar diretórios, preparar logs.
- Cálculos únicos cujo resultado será usado por todas as threads nas próximas fases.

Por que é útil

- Evita duplicação de trabalho (sem `single`, cada thread poderia fazer a mesma inicialização).
- Garante ordem: o resto do time só começa quando a thread terminar o bloco (barreira implícita).
- Mantém o código explícito: você explicita que aquela tarefa é “uma vez só”.

Exemplo

```
#pragma omp parallel
{
  #pragma omp single
  {
    // executado por uma thread
    inicializar_recursos();
  }  // <- barreira implícita

  #pragma omp for
  for (int i = 0; i < N; i++) {
    processar(i);   // já pode usar os recursos inicializados
  } // <- barreira implícita do for
}
```

Observações

- Qual thread executa o `single`? Qualquer uma — não depende de ser a thread 0.

> Use `single` quando uma tarefa deve ocorrer uma única vez dentro da região paralela e se quer que todas as threads aguardem seu término antes de prosseguir — a barreira implícita do `single` já garante isso.

Exemplo - Combinando as diretiva

- Um fluxo comum em códigos iniciantes é: inicialização única → fase 1 (loop) → fase 2 (loop).
- As barreiras implícitas já dão a ordem correta: Fase 2 só começa quando a Fase 1 terminou para todas as threads.

```
#pragma omp parallel
{
  #pragma omp single
  { prepara_recursos(); }
  // barreira implícita ao fim do single

  #pragma omp for
  for (long i = 0; i < N; i++) etapa1(i);
  // barreira implícita ao fim do for: todos terminaram etapa1

  #pragma omp for
  for (long i = 0; i < N; i++) etapa2(i);
  // barreira implícita ao fim do for: todos terminaram etapa2
} // fim do parallel (join)
```

Quando usar qual?

- `for`: muitas unidades iguais (iterações de loop).
- `sections`: poucas unidades diferentes (A/B/C).
- `single`: algo que precisa acontecer uma única vez antes das demais threads prosseguirem.

`master` vs `single`

```c
#pragma omp parallel
{
  #pragma omp master
  { printf("Só a master executa, sem barreira ao fim\n"); }

  #pragma omp single
  { printf("Uma thread qualquer executa, com barreira por padrão\n

  // Se quiser que ninguém espere a single, use:
  #pragma omp single nowait
  { printf("Single sem barreira (nowait)\n"); }
}
```

- Use `master` quando precisar que seja a thread 0 (master) a executar e não queira barreira ao final.
- Use `single` quando tanto faz qual thread executa e você quer o *ponto de encontro* (barreira) logo depois.

---

7) Controlando o número de threads

O que estamos controlando, exatamente?

- Cada região `#pragma omp parallel` executa com um time de threads (team).
- Tamanho do time = quantas threads vão trabalhar naquela região.
- Mais threads ≠ sempre mais rápido (se passar do número de CPUs, pode até piorar). Para começar, pense em igualar ao número de núcleos lógicos da máquina.

Formas de definir o tamanho do time

1. **Na própria diretiva** (vale só para aquela região)

   ```c
   #pragma omp parallel num_threads(8)
   {
     /* ... */
   }
   ```

   Use quando se quer deixar explícito o tamanho naquele ponto do código.

2. **Por código** (afeta as próximas regiões)

```c
#include <omp.h>
int main(int argc, char** argv) {
  // exemplo: decidir em tempo de execução
  int t = omp_get_num_procs();  // nº de CPUs visíveis
  if (argc > 1) t = atoi(argv[1]) // ou vir de parâmetro/arquivo d
  if (t < 1) t = 1;

  // define o "padrão" para as próximas regiões
  omp_set_num_threads(t);

  #pragma omp parallel
  { /* ... */ }

  return 0;
}
```

- Controle programático: ajusta o nº de threads com base em entrada do usuário, tamanho do problema ou detecção de hardware (`omp_get_num_procs()`), sem depender do ambiente onde o binário foi lançado.

---

3. **Por variável de ambiente**

   ```
   export OMP_NUM_THREADS=8
   ./meu_programa
   ```

   Mantém **o mesmo binário** e só muda o número de threads no ambiente (sem recompilar).

> Prioridade prática: `num_threads(...)` na diretiva vence o valor definido por `omp_set_num_threads(...)`. A variável de ambiente `OMP_NUM_THREADS` apenas inicializa o “default”; o `omp_set_num_threads(...)` pode substituí-lo durante a execução.
