OpenMP - Construto `for`

Informações do Autor

- **Nome:** Roussian Gaioso
- **Data de atualização**: 01/10/2025

1) Introdução ao paralelismo em laços

- Grande parte do tempo de execução em ciência/engenharia está em laços (varrer vetores/matrizes, aplicar a mesma operação em muitos elementos).
- O OpenMP usa o modelo fork–join: o programa começa com 1 thread; ao entrar numa região paralela, cria-se um time de threads; ao final, todas se juntam e a execução volta a ser sequencial.
- Para paralelismo de dados, o caminho natural é dividir as iterações do laço entre várias threads. É aqui que entram os construtos:
  - `#pragma omp parallel` → cria o time de threads;
  - `#pragma omp for` → reparte as iterações entre as threads já existentes;
  - `#pragma omp parallel for` → atalho que faz as duas coisas de uma vez.

2) Anatomia: `parallel`, `for` e a forma combinada

2.1 `#pragma omp parallel`

Cria uma região paralela: todas as threads do time executam o mesmo bloco de código entre chaves.

```
#pragma omp parallel
{
  /* cada thread executa este bloco */
}
```

> Importante: se você colocar um laço comum aqui dentro sem um construto de divisão de trabalho (worksharing), todas as threads executarão o mesmo laço inteiro (duplicação de trabalho). Por isso usamos `for`.

`#pragma omp for`

É um worksharing: não cria threads, apenas reparte as iterações do laço entre as threads que já existem no time.

- Cada iteração executa exatamente uma vez por uma thread.
- Índice do laço é privado a cada thread.
- Há barreira implícita no fim do `for` (ninguém sai antes de todas as iterações terminarem).

```
#pragma omp parallel
{
  #pragma omp for
  for (long i = 0; i < N; i++) {
    /* corpo do laço */
  }  /* <- barreira implícita aqui */
}
```

`#pragma omp parallel for`

Atalho: cria o time e divide o laço na mesma linha. Equivale a escrever `parallel { #pragma omp for ... }`.

```
#pragma omp parallel for
for (long i = 0; i < N; i++) {
  /* corpo do laço */
}  /* <- barreira implícita aqui */
```

Notas Importantes

1. Independência: a iteração `i` não depende do resultado de outras iterações.
2. Sem conflito de escrita: "1 iteração → 1 endereço" (ex.: `A[i]`), sem duas iterações escrevendo o mesmo lugar.
3. Ordem irrelevante: OpenMP não garante ordem das iterações.
4. Barreira implícita no fim do `for`: o trecho seguinte só roda quando todas as iterações acabarem.
5. Acúmulos globais** (soma, mínimo, máximo) não devem ser feitos com um único escalar compartilhado; use **reductions**.

Exemplos

Preencher um vetor

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

int main(void) {
  const long N = 1000000;
  static double A[1000000];

  #pragma omp parallel for
  for (long i = 0; i < N; i++) {
    // cada iteração escreve em A[i]
    A[i] = 1.0 / (i + 1);
  } // barreira implícita aqui

  printf("A[0]=%.3f  A[%ld]=%.6f\n", A[0], N-1, A[N-1]);
  return 0;
}
```

- Cada iteração mexe apenas em `A[i]` (endereços distintos) → sem corrida.
- A barreira garante que o vetor está completo quando o `printf` roda.

Duas fases em sequência

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

int main(void) {
  const int N = 100000;
  static int A[100000], B[100000];

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

    #pragma omp for
    for (int i = 0; i < N; i++) {
      B[i] = 2 * A[i]; // Fase 2: consumir A com segurança
    } // <- barreira implícita
  }

  printf("B[123]=%d\n", B[123]);
  return 0;
}
```

Matriz: varrer por linhas

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

int main(void) {
  const int N = 1000, M = 1000;
  //Com o intuito de facilitar o código,
  //pense A, B e C como se fossem matrizes
  //linearizadas, criadas com malloc
  static double A[N*M], B[N*M], C[N*M];

  // inicializa A e B
  for (int i = 0; i < N*M; i++)
    { A[i] = 1.0; B[i] = 2.0; }

  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    // percorre linhas
    // e colunas (stride 1 no j)
    for (int j = 0; j < M; j++) {
      C[i*M + j] = A[i*M + j] + B[i*M + j];
    }
  }

  printf("C[0]=%.1f  C[%d]=%.1f\n", C[0], N*M-1, C[N*M-1]);
  return 0;
}
```

Notas

- Índice do laço (`i`, `j`) → privado a cada thread (cada uma tem sua cópia).
- Vetores/matrizes (alocados no heap ou como `static`) → compartilhados por todas as threads.
- Variáveis declaradas dentro do corpo do `for` → ficam na stack da thread → privadas.

> Regra prática: "variáveis do laço → privadas; dados grandes → compartilhados; cada iteração escreve sua própria posição."

- Quando não usar `parallel for` (ou usar com cuidado)
  - Se existe dependência entre iterações (ex.: `A[i]` depende de `A[i-1]`), paralelizar direto pode quebrar o algoritmo.
  - Se várias iterações atualizam o mesmo endereço, você terá corrida de dados (a não ser que use técnicas apropriadas — *reductions*, reestruturação do algoritmo, etc.).

- Se o laço tem pouquíssimas iterações, o custo de criar/gerir threads pode superar o ganho.

Gerenciamento de Dados: Cláusulas de Escopo

Quando várias threads acessam a mesma memória ao mesmo tempo, você pode ter condições de corrida (resultados errados, intermitentes). As cláusulas de escopo do OpenMP dizem quem enxerga o quê dentro das regiões paralelas.

---

1) `shared(list)` — dados vistos por todas as threads

O que faz: as variáveis listadas são compartilhadas; qualquer thread enxerga as mesmas localizações de memória.

Quando usar: leitura compartilhada (ex.: vetor de entrada), ou escrita sem conflito (cada thread escreve em **índices diferentes**).

Cuidado: se duas threads escrevem o mesmo endereço, há risco de corrida.

Exemplo (correto: leitura compartilhada, escrita sem conflito):

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

int main(void) {
  const int N = 8;
  static int A[8], B[8];

  for (int i = 0; i < N; i++)
    A[i] = i;

  #pragma omp parallel for default(none) shared(A,B)
  for (int i = 0; i < N; i++) {
    // cada thread escreve B[i] (índice distinto)
    B[i] = 2 * A[i];
  }

  for (int i = 0; i < N; i++)
    printf("%d ", B[i]);

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

---

2) `private(list)` — uma cópia por thread

O que faz: cada thread ganha sua própria cópia da variável; o valor *não é inicializado com o da original*.

Quando usar: variáveis temporárias do corpo do laço, contadores auxiliares, acumuladores locais por iteração.

Exemplo:

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

int main(void) {
  const int N = 8;
  static int A[8], B[8];

  for (int i = 0; i < N; i++)
    A[i] = i;

  // se ficar compartilhado,
  // várias threads disputam 'tmp'
  int tmp = 0;

  #pragma omp parallel for default(none) shared(A,B) private(tmp)
  for (int i = 0; i < N; i++) {
    // cada thread tem seu próprio 'tmp'
    tmp = A[i] * A[i];
    B[i] = tmp + 1;
  }

  for (int i = 0; i < N; i++)
  printf("%d ", B[i]);
  puts("");
  return 0;
}
```

> Observação: se você declarar `tmp` dentro do laço, ele já seria privado por natureza; aqui mostramos `private(tmp)` para o caso em que ele é declarado fora.

---

3) `firstprivate(list)` — `private` com valor inicial

O que faz: igual ao `private`, mas a cópia de cada thread começa com *o valor da variável original*.

Quando usar: quando cada thread precisa de um valor inicial comum (ex.: parâmetro/escala/semente) para trabalhar localmente.

Exemplo:

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

int main(void) {
  const int N = 8;
  static int A[8], B[8];
  for (int i = 0; i < N; i++)
    A[i] = i;

  // definido fora da região
  int escala = 3;

  #pragma omp parallel for default(none) shared(A,B) firstprivate(
  for (int i = 0; i < N; i++) {
    // cada thread tem sua própria
    // cópia inicializada com 3
    B[i] = escala * A[i];
    // modifica a CÓPIA local,
    // não afeta as outras threads
    escala += 1;
  }

  for (int i = 0; i < N; i++)
    printf("%d ", B[i]);
  puts("");
  // aqui, 'escala' fora
  // da região continua valendo 3
  return 0;
}
```

---

4) `lastprivate(list)` — mantém o valor da última iteração

O que faz: ao final do `for`, a variável privada da última iteração (na ordem sequencial do laço) é copiada de volta para a variável original.

Quando usar: quando você precisa saber qual foi o último valor calculado no laço (ex.: último índice que satisfaz uma condição).

Exemplo:

- Capturar o último múltiplo de 3 em `[0..N)`

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

int main(void) {
  const int N = 20;
  int ultimo = -1;  // fora do for

  #pragma omp parallel for default(none) lastprivate(ultimo)
  for (int i = 0; i < N; i++) {
     // 'ultimo' aqui é privado por thread
    if (i % 3 == 0) ultimo = i;
  } // <- ao sair, 'ultimo' recebe o
    // valor da ÚLTIMA iteração (i=N-1) em ordem seq.

  printf("ultimo multiplo de 3 em [0,%d): %d\n", N, ultimo);
  // deve ser 18
  return 0;
}
```

> Observação: o `lastprivate` copia o valor da variável na última iteração (em ordem). Se você precisa "o último que satisfaz", esse padrão funciona porque a iteração percorre em ordem crescente e cada vez atualiza o candidato.

---

5) `default(none)` (obrigatório listar tudo)

O que faz: desliga os padrões implícitos; precisa-se declarar `shared(...)`, `private(...)`, etc., para cada variável usada.

Quando usar: sempre que possível durante o aprendizado e em código sério — ajuda a evitar bugs de escopo difíceis de ver.

Exemplo:

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

int main(void) {
  const int N = 8;
  static int A[8], B[8];
  for (int i = 0; i < N; i++)
    A[i] = i;

  #pragma omp parallel for default(none) shared(A,B)
  for (int i = 0; i < N; i++) {
    // 't' é local do bloco -> privado por natureza
    int t = A[i] * 2;
    B[i] = t + 1;
  }

  for (int i = 0; i < N; i++)
    printf("%d ", B[i]);
  puts("");
  return 0;
}
```

---

6) `reduction(op: list)`

1. Problema sem `reduction`

- Quando várias threads executam uma operação de acumulação em uma mesma variável compartilhada, ocorre condição de corrida (*race condition*).
- Isso acontece porque todas as threads tentam atualizar o mesmo endereço de memória simultaneamente, resultando em valores incorretos.

Exemplo incorreto:

```c
double soma = 0.0;

#pragma omp parallel for
for (long i = 0; i < N; i++) {
    soma += 1.0 / (i + 1); // condição de corrida
}
```

---

2. Funcionamento do `reduction`

A cláusula `reduction(op: var)` organiza o acesso concorrente em três etapas:

1. **Cópia privada por thread**

- Cada thread recebe uma cópia privada da variável `var`.
- Essa cópia é inicializada com o valor neutro da operação (exemplo: `0` para soma, `1` para produto).

2. **Acumulação independente**
   - Durante a execução do bloco paralelo, cada thread atualiza apenas a sua cópia privada.
   - Não ocorre disputa de memória, pois não há acessos concorrentes à variável global.
3. **Combinação final**
   - Ao término da região paralela, o *runtime* do OpenMP combina as cópias locais aplicando a operação especificada (`+`, `*`, `max`, `min`, etc.).
   - O resultado final é armazenado na variável original.

---

3. Operações disponíveis

As operações mais utilizadas em reduções são:

- `+` soma
- `*` produto
- `max` máximo
- `min` mínimo
- `&`, `|`, `^` operações bit a bit
- `&&`, `||` operações lógicas

Cada operação possui um valor neutro utilizado na inicialização:

- Soma → `0`
- Produto → `1`
- Máximo → menor valor representável
- Mínimo → maior valor representável

---

4. Exemplo — soma harmônica

```
#pragma omp parallel for reduction(+:soma)
for (long i = 0; i < N; i++) {
    soma += 1.0 / (i + 1);
}
```

- Cada thread acumula localmente os valores da série.
- Ao final, o OpenMP combina todas as somas parciais em uma única variável `soma`.
- O resultado é correto, sem necessidade de travas ou sincronizações manuais.

Otimizando a Execução e o Controle do Laço

Além da corretude (escopos de dados), pode-se ganhar (ou perder) muito desempenho decidindo como as iterações do laço são distribuídas entre as threads. O OpenMP oferece cláusulas para:

- **Mapear iterações → threads**: `schedule(...)`
- **Distribuir laços aninhados**: `collapse(n)`
- **Evitar esperas desnecessárias**: `nowait`
- **Forçar uma ordem de saída quando necessário**: `ordered`

Agendamento de iterações com `schedule(...)`

A cláusula `schedule` define como as iterações de um `for` paralelo são divididas entre as threads e qual o tamanho do bloco de iterações atribuído a cada uma. O objetivo é equilibrar o trabalho e reduzir tempo ocioso.

Tipos de agendamento

1. `static`

- As iterações são pré-divididas de forma fixa entre as threads.
- Ideal quando todas as iterações custam o mesmo.
- Menor overhead.

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++)
{ ... }
```

Com chunk (`static,chunk`)

- Divide em blocos de tamanho `chunk`.
- Atribui os blocos em ordem *round-robin* (thread 0 pega [0..chunk-1], thread 1 pega [chunk..2*chunk-1], e assim por diante).
- Usar `schedule(static, chunk)` com blocos pequenos faz com que as iterações sejam distribuídas de maneira intercalada entre as threads.
    - Isso ajuda a evitar que algumas threads fiquem sobrecarregadas em regiões específicas (`hotspots`) e também diminui os conflitos de cache provocados pelo false sharing.

---

2. `dynamic`

- As threads pegam blocos sob demanda.
- Quando uma termina, solicita o próximo bloco.

- Balanceia automaticamente se as iterações têm custo desigual.
- Tem overhead maior que `static`.

```
#pragma omp parallel for schedule(dynamic, 512)
for (int i = 0; i < N; i++) { ... }
```

---

3. `guided`

- Semelhante ao `dynamic`, mas o tamanho dos blocos diminui ao longo do tempo.
- Começa com blocos grandes (menos overhead) e vai afinando a divisão para melhorar o balanceamento no final.

```
#pragma omp parallel for schedule(guided, 256)
for (int i = 0; i < N; i++)
{ ... }
```

Fórmula aproximada

O tamanho do próximo bloco é calculado em função do número de iterações restantes / número de threads.

Exemplo:

```
chunk = max( (iterações_restantes / num_threads), chunk_min )
```

Ou seja:

- no início (muitas iterações), o bloco é grande;
- no final (poucas iterações), o bloco se aproxima de chunk_min (parâmetro fornecido).

Exemplo aplicado (N=32, 4 threads, guided, chunk=2):

1. **Passo 1 (início)**
   - 32 iterações disponíveis → cada thread pega ~8 iterações.
2. **Passo 2 (restam 16)**
   - Próxima rodada: blocos menores (~4 iterações cada).
3. **Passo 3 (restam 8)**
   - Agora blocos de 2 (atingiu o mínimo especificado).

Distribuição aproximada:

```
Thread 0: [0..7], depois [16..19], depois [28..29]...
Thread 1: [8..15], depois [20..23], depois [30..31]...
```

---

4. `runtime`

- A decisão de qual política usar é deixada para a execução, via variável de ambiente `OMP_SCHEDULE`.
- Permite testar diferentes políticas sem recompilar o código.

```
#pragma omp parallel for schedule(runtime)
for (int i = 0; i < N; i++) { ... }
```

Exemplo de execução:

```
OMP_SCHEDULE="static" ./prog
OMP_SCHEDULE="static,1024" ./prog
OMP_SCHEDULE="dynamic,512" ./prog
OMP_SCHEDULE="guided,256" ./prog
```

Chunk (tamanho do bloco)

- **Chunk grande**
  - Menos overhead (menos trocas de bloco).
  - Melhor localidade de memória (blocos contíguos).
  - Bom para laços regulares.
- **Chunk pequeno**
  - Mais flexibilidade para equilibrar trabalho desigual.
  - Maior overhead (mais requisições de blocos).
  - Pode prejudicar localidade de cache.

Dicas

- **Laços regulares** (todas as iterações custam o mesmo): Use `schedule(static)` ou `schedule(static, chunk grande)`.
- **Laços irregulares** (algumas iterações custam muito mais): Use `schedule(dynamic, chunk moderado)` ou `schedule(guided, chunk moderado)`.
- **Exploração experimental**: Use `schedule(runtime)` e ajuste com `OMP_SCHEDULE` para comparar sem recompilar.

Tabela dos Tipos de schedule

Tipo	O que faz	Quando usar
`static[,chunk]`	Divide o intervalo em blocos fixos. Sem chunk, cada thread recebe ~1 fatia contígua. Com chunk, distribui em "rodízio" (round-robin) blocos de tamanho chunk.	Custo por iteração parecido (melhor localidade, menor overhead).
`dynamic[,chunk]`	Threads pegam blocos na hora conforme vão terminando.	Custo muito variável por iteração; precisa de bom balanceamento.
`guided[,chunk]`	Parecido com `dynamic`, mas começa com blocos grandes e vai diminuindo até chegar no chunk.	Meio-termo: bom balanceamento com menos overhead que `dynamic`.
`runtime`	Decide em tempo de execução via `OMP_SCHEDULE` (sem recompilar).	Experimentar/Ajustar sem recompilar; scripts e benchmarks.

Exemplo — `static`

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

int main(void) {
  const int N = 100000;
  static double A[100000], B[100000];

  for (int i = 0; i < N; i++)
    A[i] = i;

  double t0 = omp_get_wtime();
  #pragma omp parallel for schedule(static)
    // ou schedule(static, 1024)
  for (int i = 0; i < N; i++) {
    B[i] = 2.0 * A[i];
  }
  double t1 = omp_get_wtime();
  printf("tempo(static) = %.3f s  (B[99999]=%.1f)\n",
         t1 - t0, B[N-1]);
  return 0;
}
```

Exemplo 2 — `dynamic`

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

int main(void) {
  const int N = 120000;
  static int OUT[120000];

  double t0 = omp_get_wtime();
  #pragma omp parallel for schedule(dynamic, 512)
  for (int i = 0; i < N; i++) {
    // custo variável por iteração:
    // simula “trabalho” diferente em cada i
    // varia de 100 a 10.099
    int iters = 100 + (i % 10000);
    int acc = 0;
    for (int k = 0; k < iters; k++)
        acc += (k & 7);
    // grava por índice distinto (sem corrida)
    OUT[i] = acc;
  }
  double t1 = omp_get_wtime();
  printf("tempo(dynamic,512) = %.3f s (OUT[12345]=%d)\n",
         t1 - t0, OUT[12345]);
  return 0;
}
```

Por que `dynamic` aqui? Se algumas iterações demoram bem mais, `dynamic` repassa trabalho às threads que terminam cedo, equilibrando a carga (custa um pouco mais de overhead).

Exemplo 3 — `guided`

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

int main(void) {
  const int N = 120000;
  static int OUT[120000];

  double t0 = omp_get_wtime();
  #pragma omp parallel for schedule(guided, 256)
  for (int i = 0; i < N; i++) {
    int iters = 100 + (i % 10000);
    int acc = 0;
    for (int k = 0; k < iters; k++)
        acc += (k & 7);
    OUT[i] = acc;
  }
  double t1 = omp_get_wtime();
  printf("tempo(guided,256) = %.3f s  (OUT[54321]=%d)\n",
        t1 - t0, OUT[54321]);
  return 0;
}
```

Exemplo 4 — `runtime` (trocar política sem recompilar)

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

int main(void) {
  const int N = 100000;
  static double C[100000];

  double t0 = omp_get_wtime();
  #pragma omp parallel for schedule(runtime)
  for (int i = 0; i < N; i++) {
    C[i] = 1.0 / (i + 1);
  }
  double t1 = omp_get_wtime();
  printf("tempo(schedule=runtime) = %.3f s  (C[99999]=%.6f)\n",
         t1 - t0, C[N-1]);
  return 0;
}
```

Execute variando a política:

```
OMP_NUM_THREADS=8 OMP_SCHEDULE="static"         ./sch_runtime
OMP_NUM_THREADS=8 OMP_SCHEDULE="static,1024"    ./sch_runtime
OMP_NUM_THREADS=8 OMP_SCHEDULE="dynamic,512"    ./sch_runtime
OMP_NUM_THREADS=8 OMP_SCHEDULE="guided,256"     ./sch_runtime
```

Distribuindo laços aninhados: `collapse(n)`

Problema sem `collapse`

Quando há laços aninhados, o OpenMP só paraleliza o laço mais externo.

Exemplo:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)// paralelo
{
  for (int j = 0; j < M; j++)
  {  // sequencial dentro da thread
      ...
  }
}
```

- Aqui o paralelismo acontece só sobre as `N` iterações externas.
- Se `N` for pequeno (ex.: 4) e `M` muito grande (ex.: 10000), apenas 4 threads trabalharão, mesmo que existam 16 ou 32 threads disponíveis.
- Resultado: má distribuição de carga.

O que `collapse(n)` faz

`collapse(n)` achata os `n` primeiros laços e transforma em um único espaço de iterações.

No exemplo com 2 laços (`i` e `j`):

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        ...
    }
}
```

Isso é equivalente a “linearizar” o espaço `(i,j)` em um único loop com `N*M` iterações:

```
#pragma omp parallel for
for (int k = 0; k < N*M; k++) {
    int i = k / M;
    int j = k % M;
    ...
}
```

Assim, o OpenMP consegue dividir N*M iterações entre as threads, aproveitando melhor o paralelismo.

E se houver 3 laços?

Basta usar `collapse(3)` para achatar os três.

Exemplo:

```
#pragma omp parallel for collapse(3)
for (int i = 0; i < X; i++) {
    for (int j = 0; j < Y; j++) {
        for (int k = 0; k < Z; k++) {
            ...
        }
    }
}
```

O OpenMP enxerga isso como um único loop de `X*Y*Z` iterações:

```
for (int t = 0; t < X*Y*Z; t++) {
    int i = t / (Y*Z);
    int j = (t / Z) % Y;
    int k = t % Z;
    ...
}
```

Assim todas as combinações `(i,j,k)` podem ser distribuídas entre as threads.

Evitando esperas desnecessárias: `nowait`

Todo `for` tem barreira implícita ao fim. Se não existe dependência entre dois laços consecutivos, você pode remover a espera do primeiro com `nowait`.

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

int main(void) {
  const int N = 100000;
  static int A[100000], B[100000];

  #pragma omp parallel
  {
    // Laços independentes: podem avançar
    // sem esperar o outro terminar
    #pragma omp for nowait
    for (int i = 0; i < N; i++)
        A[i] = i;

    #pragma omp for
    for (int i = 0; i < N; i++)
        B[i] = 2*i;
  } // barreira no fim do parallel

  printf("A[5]=%d  B[5]=%d\n", A[5], B[5]);
  return 0;
}
```

> Só use `nowait` quando tiver certeza de que não há dependência entre os laços. Em caso de dúvida, não remova a barreira.

Quando se precisa de ordem: `ordered`

Se o seu laço deve imprimir ou registrar em ordem crescente de `i`, use `ordered`. (É raro em HPC, mas útil para debug/demonstração.)

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

int main(void) {
  const int N = 16;

  #pragma omp parallel for ordered schedule(static)
  for (int i = 0; i < N; i++) {
    // trabalho (paralelo) aqui …
    #pragma omp ordered
    printf("%d ", i);
    // sai em ordem: 0 1 2 3 ...
  }
  puts("");
  return 0;
}
```

Notas Finais sobre variáveis

1. Variáveis `static`

- Alocadas no **segmento de dados**, não na stack.
- Usadas para evitar *stack overflow* em arrays grandes.
- São **compartilhadas** entre threads por padrão.
- Inicializam com zero se não forem explicitamente inicializadas.

```c
static double A[1000000]; // fora da stack
```

---

2. Variáveis constantes (`const`)

- `const int N = 8;` → substituída pelo valor literal em tempo de compilação.
- Não precisa ser listada em `shared(...)` quando se usa `default(none)`.
- Se fosse `int N = 8;` → é uma variável real → precisa de `shared(N)`.

```c
const int N = 8; // vira constante
int M = 8;       // variável, deve ser listada em shared(M)
```

---

3. Variáveis locais no `main` (antes da região paralela)

- Vivem no **stack da thread principal** (frame de ativação do `main`).

- Todas as threads enxergam o mesmo endereço → **shared por padrão**.
- Podem causar *race condition* se não forem declaradas como `private`.

```
int tmp = 0; // está no stack da main → shared por padrão
#pragma omp parallel for private(tmp) // força cópia por thread
```

---

4. Variáveis declaradas dentro da região paralela

- Cada thread tem sua **própria stack**.
- Variáveis locais declaradas dentro do bloco paralelo são **private por natureza**.

```
#pragma omp parallel
{
    int tmp; // cada thread tem sua própria versão
}
```

---

5. Índices de loop (`for`)

- Em `#pragma omp parallel for`, o índice (`i`) é sempre **private automático**.
- Não precisa declarar manualmente.

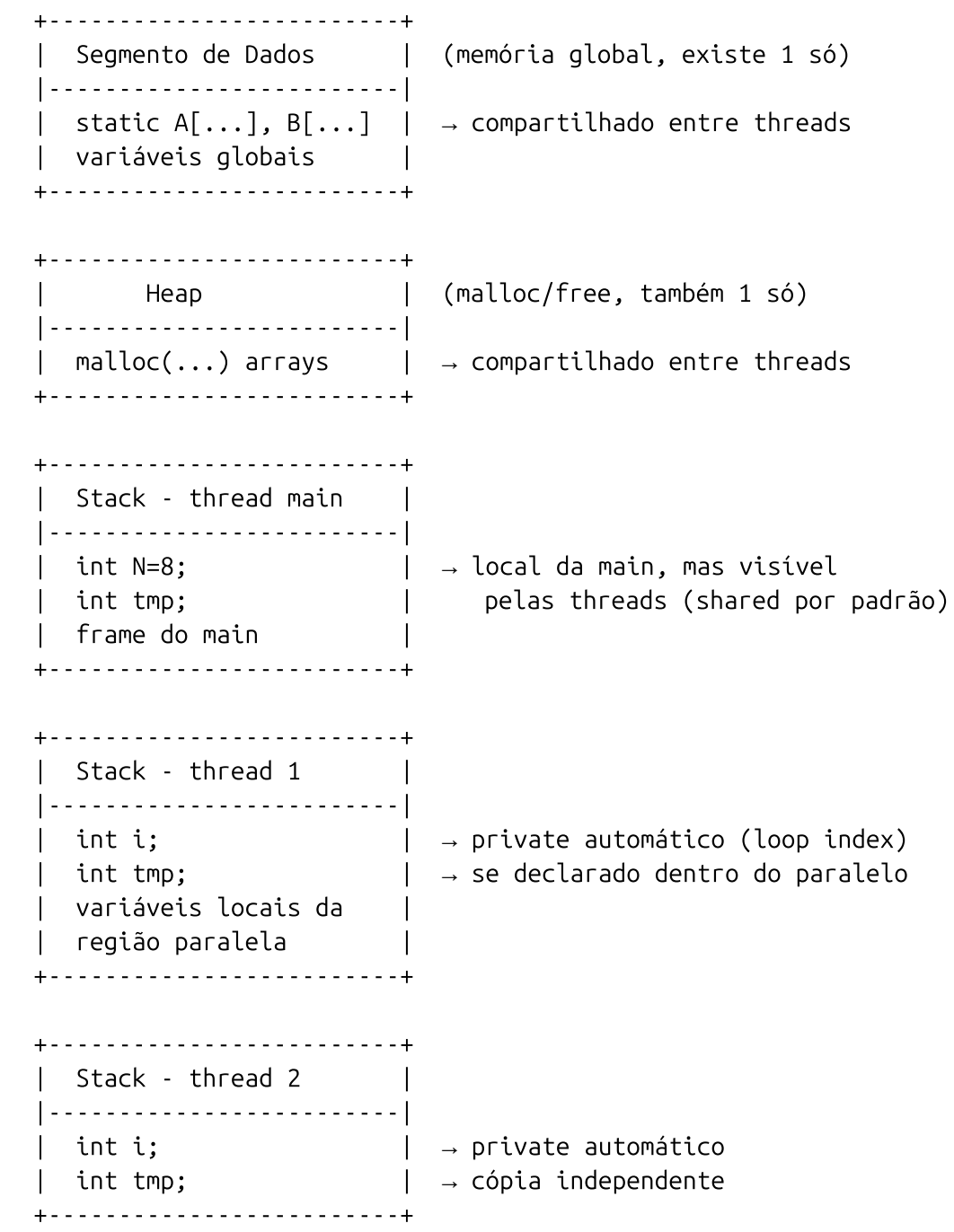
```
#pragma omp parallel for
for (int i = 0; i < N; i++) { ... } // i já é private
```

---

6. `default(none)`

- Exige que todas as variáveis (exceto índices de loop) sejam declaradas explicitamente como `shared`, `private`, etc.
- Garante clareza e evita bugs de concorrência.

```
#pragma omp parallel for default(none) shared(A,B) private(tmp)
```


Segmento de Dados
(memória global, existe 1 só)
static A[...], B[...]
variáveis globais
→ compartilhado entre threads
Heap
(malloc/free, também 1 só)
malloc(...) arrays
→ compartilhado entre threads
Stack - thread main
int N=8;
int tmp;
frame do main
→ local da main, mas visível
pelas threads (shared por padrão)
Stack - thread 1
int i;
int tmp;
variáveis locais da
região paralela
→ private automático (loop index)
→ se declarado dentro do paralelo
Stack - thread 2
int i;
int tmp;
→ private automático
→ cópia independente