Paralelismo de Tarefas em OpenMP: Diretiva `task`

Informações do Autor

- **Nome:** Roussian Gaioso
- **Data de atualização**: 18/10/2025

Introdução: uso de tarefas

> O `omp for` distribui as iterações de um laço entre as threads do time.

- Ele é eficiente quando o número de iterações é conhecido e cada uma executa uma quantidade semelhante de trabalho.
- Nessas condições, o agendador consegue dividir as iterações de forma equilibrada, evitando que algumas threads terminem antes das outras e fiquem ociosas.

> Em muitos problemas, *o volume e a natureza do trabalho variam durante a execução*.
> Isso ocorre em algoritmos que percorrem grafos, exploram árvores ou usam divisão e conquista.

- Nesses casos, o OpenMP oferece a diretiva `task` que cria unidades de trabalho de forma dinâmica.
- Cada tarefa pode ser executada por qualquer thread disponível, permitindo adaptação à carga de execução.
- O OpenMP mantém o modelo fork/join.
    - A diretiva `#pragma omp parallel` cria um grupo de threads para executar um bloco de código.
    - No fim do bloco ocorre a junção das threads e o programa retoma o fluxo sequencial.

        > Dentro dessa região paralela, a diretiva `#pragma omp task` cria tarefas que são enfileiradas e executadas pelo runtime.
        >
        > O agendador distribui as tarefas entre as threads do grupo conforme os recursos disponíveis.

Quando preferir tarefas em vez de laços paralelos

1. O programa precisa gerar novos trabalhos durante a execução, como em algoritmos recursivos ou percursos de árvore.
2. Quando cada parte do trabalho leva um tempo diferente para terminar, o que torna a divisão fixa do `omp for` ineficiente

3. O número de unidades de trabalho é desconhecido ou muda durante a execução (quando não há um laço simples que represente todo o processamento a ser feito).
4. Quando uma parte do programa depende do resultado de outra e essas relações são mais fáceis de expressar separando o trabalho em tarefas.

> Se o número de iterações é fixo e cada uma tem custo semelhante, o `omp for` continua sendo a opção mais simples e eficiente.

Criação e sincronização

`task`: criar uma unidade de trabalho

Quando o programa encontra `#pragma omp task`, a thread que está executando não roda o bloco imediatamente.

- Em vez disso, ela cria um objeto tarefa, que contém o código a ser executado e as variáveis necessárias e coloca essa tarefa em uma *fila interna*.
- Qualquer thread do grupo pode retirar a tarefa dessa fila e executá-la depois.

> Tarefas normalmente são criadas dentro de uma região `parallel`; fora dela, todas as tarefas são executadas pela mesma thread, sem paralelismo.

Regras práticas para criar tarefas

1. **Controle de variáveis**
    - Especifique quais variáveis a tarefa deve enxergar.
    - Use `firstprivate(...)` para copiar valores de entrada, como índices de laço
    - Use `shared(...)` para variáveis cujo resultado deve ser visível fora da tarefa.
    - Variáveis declaradas dentro do bloco da tarefa são sempre privadas a ela.
2. **Sincronização**
    - Se o código após as tarefas depende dos resultados produzidos por elas, é necessário sincronizar.
    - Use `#pragma omp taskwait` para esperar todas as tarefas criadas pelo thread atual terminarem antes de continuar
    - Quando as tarefas geram outras tarefas recursivamente, utilize `taskgroup` para esperar a conclusão de toda a hierarquia.

Exemplo-1

Duas tarefas independentes somam metades diferentes do mesmo vetor. Após o `taskwait`, o programa combina os resultados parciais. Esse exemplo demonstra:

- Criação de tarefas dentro de uma região `parallel` e um bloco `single`.

- Uso de variáveis `shared` para armazenar resultados das tarefas.
- Necessidade de sincronizar com `taskwait` antes de acessar os resultados.

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

int main(void) {
  const int N = 1000000;
  double *A = (double*)malloc(N * sizeof(double));

  // Preenche o vetor: soma esperada = N
  for (int i = 0; i < N; i++)
    A[i] = 1.0;

  double sumL = 0.0;  // soma da metade esquerda
  double sumR = 0.0;  // soma da metade direita

  #pragma omp parallel
  {
    #pragma omp single
    {
      // Tarefa 1: soma A[0 .. N/2-1]
      #pragma omp task shared(sumL, A)
      {
        double acc = 0.0;
        for (int i = 0; i < N/2; i++)
          acc += A[i];
        sumL = acc; // escrita isolada, sem corrida
      }

      // Tarefa 2: soma A[N/2 .. N-1]
      #pragma omp task shared(sumR, A)
      {
        double acc = 0.0;
        for (int i = N/2; i < N; i++)
          acc += A[i];
        sumR = acc; // escrita isolada, sem corrida
      }

      // Espera o fim das duas tarefas
      #pragma omp taskwait

      double total = sumL + sumR;
```

```
      printf("sumL=%.0f  sumR=%.0f  total=%.0f (esperado=%.0f)\n",
             sumL, sumR, total, (double)N);
    }
  }

  free(A);
  return 0;
}
```

Explicação:

- Cada tarefa grava em uma variável diferente (`sumL` e `sumR`), evitando condições de corrida.
- O vetor A é compartilhado apenas para leitura, o que dispensa sincronização adicional.
- O `taskwait` garante que as duas tarefas terminem antes de combinar os resultados.
- A combinação `parallel` + `single` indica que apenas uma thread cria as tarefas, mas todas as threads do time podem executá-las.
- **Observação:** esse exemplo não mostra vantagem real sobre `omp parallel for`, já que a divisão em duas metades é fixa e previsível.

Exemplo-2

Este exemplo usa o algoritmo recursivo de Fibonacci para demonstrar a criação dinâmica de tarefas.

- Em funções recursivas, cada chamada pode gerar novas tarefas, e essas tarefas, por sua vez, podem criar outras.
- Isso mostra como o paralelismo pode se adaptar automaticamente à estrutura do problema.

Porém, se cada chamada criasse novas tarefas sem restrição, a sobrecarga de criação seria maior que o ganho.

- Por isso é comum definir um limite (cutoff): a partir de um certo nível de recursão, as chamadas passam a ser executadas de forma sequencial, mantendo a granularidade adequada.

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

int fib_task(int n) {
  if (n < 2)
```

```
    return n;

  int a = 0, b = 0;

  if (n > 20) { // limite: evita tarefas muito pequenas
    #pragma omp task shared(a) firstprivate(n)
    { a = fib_task(n - 1); }

    #pragma omp task shared(b) firstprivate(n)
    { b = fib_task(n - 2); }

    #pragma omp taskwait
  } else {
    a = fib_task(n - 1);
    b = fib_task(n - 2);
  }

  return a + b;
}

int fib_parallel(int n) {
  int result = 0;
  #pragma omp parallel
  {
    #pragma omp single
    { result = fib_task(n); }
  }
  return result;
}

int main(void) {
  int n = 35;
  double t0 = omp_get_wtime();
  int f = fib_parallel(n);
  double t1 = omp_get_wtime();
  printf("fib(%d) = %d  tempo=%.3f s\n", n, f, t1 - t0);
  return 0;
}
```

Explicação:

- Cada chamada `fib_task` pode gerar até duas novas tarefas.
- As tarefas são criadas apenas para chamadas “grandes” (`n > 20`), evitando tarefas pequenas demais.

- As tarefas usam `shared(a)` e `shared(b)` porque as variáveis estão fora do escopo local das tarefas e precisam armazenar os resultados.
- O `taskwait` garante que as duas tarefas terminem antes de somar os resultados.
- A região `parallel` com `single` cria apenas uma thread produtora de tarefas, enquanto as demais executam as tarefas geradas.

`taskgroup`: sincronizar uma árvore de tarefas

Quando uma função recursiva cria novas tarefas dentro de outras tarefas, o uso de `taskwait` nem sempre é suficiente.

> O `taskwait` faz a thread esperar apenas pelas tarefas criadas diretamente por ela.

- Se essas tarefas gerarem novas tarefas internas, o `taskwait` não aguarda os "netos".
- Como resultado, a função pode retornar antes de toda a subárvore de tarefas terminar, produzindo resultados incorretos.

O `taskgroup` resolve esse problema.

- Ele permite delimitar um bloco de código em que todas as tarefas criadas — tanto as diretas quanto as criadas dentro de outras — devem ser concluídas antes de prosseguir.

> Tecnicamente, o `#pragma omp taskgroup` cria um escopo de sincronização.

- Ao sair desse bloco, o runtime garante que todas as tarefas associadas ao grupo foram finalizadas.

Essa abordagem é importante em padrões de divide e conquistar, como ordenações recursivas ou percursos de árvores, em que cada chamada cria novas tarefas.

> O `taskgroup` fornece um ponto único e confiável para aguardar toda a hierarquia de tarefas.

Exemplo-3

Esse exemplo mostra o uso do `taskgroup` para garantir que todas as tarefas criadas dentro de uma estrutura recursiva terminem antes de prosseguir.

- A função `work` cria novas tarefas em chamadas recursivas.
- Sem o `taskgroup`, o programa poderia encerrar a região paralela antes que todas as tarefas internas fossem concluídas.

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

```c
void work(int depth) {
  if (depth <= 0)
    return;

  // Cada chamada cria uma nova tarefa recursiva
  #pragma omp task firstprivate(depth)
  {
    printf("tarefa em profundidade %d executada pela thread %d\n",
           depth, omp_get_thread_num());
    work(depth - 1);
  }
}

int main(void) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      // O grupo de tarefas inclui
      // todas as tarefas
      // criadas dentro deste bloco
      #pragma omp taskgroup
      {
        for (int i = 0; i < 4; i++) {
          #pragma omp task firstprivate(i)
          { work(5 + i); }
        }
      }
      // Somente após o término
      // de todas as tarefas (e suas descendentes)
      // o programa chega a este ponto
      printf("grupo de tarefas concluído\n");
    }
  }
  return 0;
}
```

Explicação:

- A função `work` cria novas tarefas recursivamente, simulando uma árvore de tarefas.
- Cada iteração do `for` cria uma tarefa inicial, que por sua vez cria outras.
- O `taskgroup` delimita o escopo de sincronização. Ao sair dele, o programa garante que todas as tarefas geradas — diretas e indiretas — já terminaram.

- A mensagem “grupo de tarefas concluído” só é impressa após o término completo da hierarquia.

Gerenciamento de dados em tarefas: cláusulas de escopo

Ao criar uma tarefa, o OpenMP precisa definir como cada variável será acessada quando a tarefa for executada. Isso é necessário porque a tarefa pode começar depois, em outra thread, quando o contexto original já tiver mudado.

Regras gerais de escopo em tarefas

- Se uma variável é compartilhada no contexto onde a tarefa é criada, ela continua sendo compartilhada dentro da tarefa.
- Se uma variável é privada no contexto onde a tarefa é criada, a tarefa recebe uma cópia do valor dessa variável no momento da criação.
    - Nessa situação, ela se comporta como `firstprivate` por padrão.
    - Isso evita que a tarefa use uma variável local que já mudou ou saiu de escopo quando for executada.

Observações práticas

- `firstprivate` copia o valor atual da variável.
    - Se a variável for um ponteiro, copia-se apenas o ponteiro; o conteúdo apontado continua sendo compartilhado.
- `shared` faz a tarefa usar exatamente a mesma instância da variável.
    - Se mais de uma tarefa escrever nessa variável, é necessário sincronizar o acesso.
- `private` cria uma nova variável não inicializada, visível apenas dentro da tarefa.

Cláusula	**Comportamento dentro da tarefa**	**Uso recomendado**
shared(x)	A tarefa acessa a mesma instância de x (memória compartilhada).	Quando várias tarefas precisam ler o mesmo dado, ou quando apenas uma tarefa escreve em x. Em caso de escritas concorrentes, sincronizar.
private(x)	A tarefa cria uma nova instância de x, não inicializada.	Variáveis temporárias internas, sem necessidade de valor inicial.
firstprivate(x)	A tarefa cria uma nova instância de x, inicializada com o valor de x no momento da criação.	Quando a tarefa precisa capturar parâmetros de entrada (como índices ou limites) que não devem mudar durante sua execução.

Exemplo-4

Este exemplo mostra que cada tarefa deve capturar o valor de `i` no momento da criação.

- Sem `firstprivate(i)`, todas as tarefas compartilhariam a mesma variável `i`, que continua sendo atualizada a cada iteração do laço.
- O resultado seria imprevisível, pois as tarefas poderiam usar valores de `i` diferentes dos esperados.

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

int main(void) {
  const int N = 10;
  int *out = (int*)malloc(N * sizeof(int));

  #pragma omp parallel
  {
    #pragma omp single
    {
      for (int i = 0; i < N; i++) {
        #pragma omp task firstprivate(i) shared(out)
        {
          // i é copiado no momento da criação da tarefa
          out[i] = i * i;
        }
      }
      #pragma omp taskwait
    }
  }

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

  free(out);
  return 0;
}
```

Explicação:

- Cada tarefa recebe uma cópia independente do valor de `i` no instante da criação (`firstprivate(i)`).

- Sem `firstprivate`, `i` seria compartilhado entre todas as tarefas e continuaria sendo incrementado pelo laço principal, produzindo valores incorretos.
- O `taskwait` garante que todas as tarefas terminem antes da leitura dos resultados.

Exemplo-5

Este exemplo mostra que, ao aplicar `firstprivate` a um ponteiro, apenas o valor do ponteiro é copiado para a tarefa, enquanto o conteúdo apontado continua sendo compartilhado entre todas as threads.

- Cada tarefa modifica uma parte exclusiva do vetor, o que evita condições de corrida.

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

int main(void) {
  const int N = 16;
  double *A = (double*)malloc(N * sizeof(double));

  for (int i = 0; i < N; i++)
    A[i] = 0.0;

  int chunks = 4;
  int blk = N / chunks;

  #pragma omp parallel
  {
    #pragma omp single
    {
      for (int c = 0; c < chunks; c++) {
        int begin = c * blk;
        int end   = (c == chunks - 1) ? N : begin + blk;

        #pragma omp task firstprivate(begin, end, A)
        {
          // A é copiado por valor (mesmo endereço),
          // o vetor subjacente é compartilhado.
          // Cada tarefa escreve em sua própria faixa.
          for (int i = begin; i < end; i++)
            A[i] = (double)i;
        }
      }
      #pragma omp taskwait
```

```
    }
  }

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

Explicação:

- `firstprivate(A)` copia apenas o valor do ponteiro, ou seja, o endereço do vetor.
- O vetor apontado por A continua sendo uma única área de memória compartilhada.
- Cada tarefa escreve em uma faixa distinta `[begin, end)`, eliminando disputas de escrita.
- O `taskwait` garante que todas as tarefas terminem antes da leitura final dos resultados.

Exemplo-6

Este exemplo mostra o efeito de múltiplas tarefas atualizando simultaneamente uma variável `shared`.

- Quando várias tarefas escrevem no mesmo objeto sem coordenação, ocorre condição de corrida: o valor final depende da ordem de execução e pode ficar incorreto.
- O uso de `#pragma omp atomic` força que cada atualização aconteça de forma indivisível, garantindo o resultado correto.

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

int main(void) {
  const int N = 100000;
  long total = 0;  // variável compartilhada

  #pragma omp parallel
  {
    #pragma omp single
    {
      for (int i = 0; i < N; i++) {
        #pragma omp task shared(total)
        {
          // Exemplo artificial:
          // incrementar total
          #pragma omp atomic
```

```
            total += 1;
          }
        }
        #pragma omp taskwait
      }
    }

    printf("total=%ld (esperado=%d)\n", total, N);
    return 0;
}
```

Explicação:

- A variável `total` é compartilhada entre todas as tarefas.
- Sem `atomic`, várias tarefas poderiam ler e escrever `total` ao mesmo tempo, perdendo incrementos.
- A diretiva `atomic` garante que cada operação `total += 1` seja executada isoladamente, preservando o valor correto.
- Esse padrão é apenas ilustrativo; em aplicações reais, somas desse tipo são feitas com reduções (`reduction`) em vez de tarefas individuais.

Dependências entre tarefas: cláusula depend

Quando múltiplas tarefas acessam os mesmos dados, é possível controlar a ordem de execução usando dependências explícitas.

- A cláusula `depend` informa ao runtime que uma tarefa só pode iniciar quando determinadas variáveis estiverem prontas, evitando condições de corrida sem precisar de `atomic` ou `critical`.
- O modelo é declarativo: cada tarefa anuncia quais dados consome e quais dados produz e o OpenMP organiza automaticamente a execução respeitando essas relações.

Formato geral

```
#pragma omp task depend(in: lista_in) depend(out: lista_out) depend(
```

- `in`: a tarefa lê os dados.
- `out`: a tarefa produz ou escreve os dados.
- `inout`: a tarefa lê e escreve os mesmos dados.

O runtime cria uma dependência:

- uma tarefa com `depend(in:x)` só começa após a conclusão de outra que declarou `depend(out:x)` ou `depend(inout:x)` sobre o mesmo objeto;
- tarefas que acessam dados independentes podem ser executadas em paralelo.

Exemplo-7

Neste exemplo, três tarefas manipulam um vetor.

- A segunda depende da primeira
- A terceira depende das duas anteriores.
- A cláusula **depend** organiza automaticamente a sequência correta de execução.

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

int main(void) {
  double A[3] = {0.0, 0.0, 0.0};

  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp task depend(out: A[0])
      {
        A[0] = 1.0;
        printf("Tarefa 1: A[0] = %.1f\n", A[0]);
      }

      #pragma omp task depend(in: A[0]) depend(out: A[1])
      {
        A[1] = A[0] + 2.0;
        printf("Tarefa 2: A[1] = %.1f\n", A[1]);
      }

      #pragma omp task depend(in: A[0], A[1]) depend(out: A[2])
      {
        A[2] = A[0] + A[1];
        printf("Tarefa 3: A[2] = %.1f\n", A[2]);
      }

      #pragma omp taskwait
      printf("Resultado final: A[2] = %.1f\n", A[2]);
    }
  }
```

```
  return 0;
}
```

Explicação:

- A tarefa 2 depende do término da tarefa 1 (`in: A[0]`).
- A tarefa 3 depende das duas anteriores (`in: A[0], A[1]`).
- O runtime executa apenas as tarefas que já têm todas as dependências resolvidas.
- A sincronização é automática; não há necessidade de `taskwait` entre tarefas dependentes.

Observações importantes

- Dependências são verificadas apenas entre tarefas criadas dentro da mesma região paralela.
- O controle é feito a nível de endereços de memória; duas variáveis diferentes com mesmo endereço implicam dependência.
- `depend` pode ser usado em padrões complexos de fluxo de dados, substituindo o uso de travas, seções críticas ou `atomic` em muitos casos.

Exemplo-8

Este exemplo simula um pipeline em três etapas:

1. Leitura de dados;
2. Processamento;
3. Escrita de resultados.

- Cada etapa depende da anterior.
- As tarefas de iterações diferentes podem se sobrepor, mantendo a ordem correta dentro de cada item.

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

#define N 5

int main(void) {
  double input[N], output[N];

  for (int i = 0; i < N; i++)
    input[i] = i * 1.0;

  #pragma omp parallel
```

```
  {
    #pragma omp single
    {
      for (int i = 0; i < N; i++) {
        // Etapa 1: leitura
        #pragma omp task depend(out: input[i])
        {
          printf("Lendo item %d (thread %d)\n",
                 i, omp_get_thread_num());
          // simula leitura e preparação
          input[i] += 1.0;
        }

        // Etapa 2: processamento
        #pragma omp task depend(in: input[i]) \\
        depend(out: output[i])
        {
          printf("Processando item %d (thread %d)\n",
                i, omp_get_thread_num());
          // simula computação
          output[i] = input[i] * 2.0;
        }

        // Etapa 3: escrita
        #pragma omp task depend(in: output[i])
        {
          printf("Gravando item %d (valor=%.1f)"+
                 "(thread %d)\n",
                  i, output[i], omp_get_thread_num());
        }
      }
    }
  }

  return 0;
}
```

Explicação:

- Cada item passa pelas três etapas em sequência: leitura → processamento → escrita.
- As dependências garantem que a etapa seguinte de um item só inicie quando a anterior terminar.
- Diferentes itens (`i` distintos) podem estar em estágios diferentes simultaneamente. Por exemplo, o item 0 pode estar sendo gravado enquanto o item 1 está sendo

processado e o item 2 está sendo lido.
- O runtime cuida automaticamente da sincronização entre tarefas.
- Não há necessidade de `taskwait` explícito; o término da região paralela já garante a conclusão de todas as tarefas pendentes.

Geração de tarefas a partir de laços com `taskloop`

A diretiva `#pragma omp taskloop` cria tarefas a partir das iterações de um laço.

- Cada bloco de iterações se torna uma tarefa independente, que pode ser executada por qualquer thread do time.

Diferente de `omp for`, que divide as iterações entre as threads no momento da execução, o `taskloop` gera unidades de trabalho autônomas.

- Essas tarefas entram em uma fila e são agendadas dinamicamente pelo runtime.
- Esse modelo é vantajoso quando o custo das iterações é irregular ou quando o corpo do laço pode criar novas tarefas.

Por padrão, o `taskloop` contém um ponto de sincronização implícito ao final.

- Isso significa que, ao sair do `taskloop`, todas as tarefas criadas já foram concluídas.
- Se for necessário continuar a execução sem essa espera, use a cláusula `nogroup` para remover a sincronização automática.

Cláusulas de controle de granularidade

- `grainsize(k)`: define aproximadamente quantas iterações cada tarefa deve conter.
- `num_tasks(t)`: solicita que o laço seja dividido em cerca de `t` tarefas.
- É possível combinar `grainsize` e `num_tasks` para ajustar a granularidade conforme a carga de trabalho.

Exemplo-9

- O `taskloop` permite criar tarefas automaticamente a partir das iterações de um laço.
- Cada grupo de iterações é convertido em uma tarefa independente, executada por qualquer thread disponível.
- No exemplo a seguir, o custo das iterações varia conforme o índice, o que torna o `taskloop` mais eficiente que um `omp for` convencional, já que o agendador distribui as tarefas dinamicamente.

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

```c
static inline int heavy(int i) {
  // custo variável: mais lento quando i % 1000 ≈ 999
  int iters = 100 + (i % 1000);
  int acc = 0;
  for (int k = 0; k < iters; k++)
    acc += (k & 7);
  return acc;
}

int main(void) {
  const int N = 200000;
  int *out = (int*)malloc(N * sizeof(int));

  #pragma omp parallel
  {
    #pragma omp single
    {
      // grainsize controla
      // quantas iterações
      // compõem cada tarefa
      #pragma omp taskloop grainsize(4096) shared(out)
      for (int i = 0; i < N; i++) {
        out[i] = heavy(i);
      }
      // Ao sair do taskloop,
      // todas as tarefas já foram concluídas
    }
  }

  printf("out[0]=%d  out[%d]=%d\n", out[0], N - 1, out[N - 1]);
  free(out);
  return 0;
}
```

Quando preferir `taskloop` em vez de `omp for`

- Quando o tempo de execução das iterações varia de forma imprevisível, tornando vantajoso deixar o runtime decidir dinamicamente quais threads executam cada bloco de iterações.
- Quando o corpo do laço contém criação de novas tarefas, o que inviabiliza o uso de `omp for`, que exige iterações independentes.
- Quando é necessário controlar explicitamente o tamanho das tarefas geradas a partir do laço, ajustando a granularidade por meio das cláusulas `grainsize` ou `num_tasks`.

Escolha de granularidade

- Tarefas grandes demais reduzem o balanceamento entre threads.
- Tarefas pequenas demais aumentam o custo de criação e agendamento.
- Um ponto de partida prático é usar algumas milhares de iterações por tarefa e ajustar conforme o tempo medido.

Boas práticas

- Crie tarefas apenas quando o custo do trabalho justificar a sobrecarga de criação.
- Prefira granularidades médias: poucas tarefas grandes reduzem paralelismo; muitas tarefas pequenas aumentam o custo de agendamento.
- Use `firstprivate` para variáveis de laço e `shared` apenas quando o dado for realmente comum a várias tarefas.
- Utilize `depend` ou `taskgroup` para coordenar tarefas com relações complexas.
- Sempre teste o desempenho variando o tamanho das tarefas — o melhor ponto depende da máquina, número de threads e da natureza do problema.
