Boas práticas de programação em **OpenMP**

Delimitando a região paralela

```
main()
{
#pragma omp parallel
   {
      work1(); /*-- Executed in parallel --*/
      work2(); /*-- Executed in parallel --*/
   }

#pragma omp parallel
   work1(); /*-- Executed in parallel  --*/
   work2(); /*-- Executed sequentially --*/
}
```

Otimização no uso de barreiras

- Barreiras de sincronização entre threads são operações caras.
- **Objetivo:** minimizar o uso de barreiras
- **Solução:** utilizar ***nowait*** sempre que possível

```
#pragma omp parallel
{
        .........
  #pragma omp for
    for (i=0; i<n; i++)
        .........
  #pragma omp for nowait
    for (i=0; i<n; i++)

} /*-- End of parallel region - barrier is implied --*/
```

Otimização no uso de barreiras

```
#pragma omp parallel default(none) \
        shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)
      a[i] += b[i];

   #pragma omp for nowait
   for (i=0; i<n; i++)
      c[i] += d[i];

   #pragma omp barrier

   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++)
      sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Custo na criação de threads

Um exemplo simples utilizando OpenMP:

```
#include <stdio.h>
int main()
{
   int done = 4, done2 = 5;

    #pragma omp parallel for lastprivate(done, done2) num_threads(2)
    for(int a=0; a<8; ++a)
    {
      if(a==2) done=done2=0;
      if(a==3) done=done2=1;
    }
    printf("%d,%d\n", done,done2);
}
```

Custo na criação de threads

Código ilustrativo da compilação em OpenMP:

```
#include <stdio.h>
int main()
{
   int done = 4, done2 = 5;
   OpenMP_thread_fork(2);
   {
       int this_thread = omp_get_thread_num(), num_threads = 2;
       int my_start = (this_thread  ) * 8 / num_threads;
       int my_end   = (this_thread+1) * 8 / num_threads;

       int priv_done, priv_done2; // not initialized, no firstprivate

       for(int a=my_start; a<my_end; ++a)
       {
           if(a==2) priv_done=priv_done2=0;
           if(a==3) priv_done=priv_done2=1;
       }
       if(my_end == 8)
       {
          // assign the values back, because this was the last iteration
          done  = priv_done;
          done2 = priv_done2;
       }
   }
   OpenMP_join();
}
```

Minimizar regiões paralelas

- Minimizar o custo de criação de regiões paralelas.
- Ao invés de criar uma região paralela para cada loop, as mesmas threads são utilizadas entre diversos loops.

```
#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 1 --*/
}

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop 2 --*/
}
       .........

#pragma omp parallel for
for (.....)
{
    /*-- Work-sharing loop N --*/
}
```

```
#pragma omp parallel
{
    #pragma omp for  /*-- Work-sharing loop 1 --*/
    { ..... }

    #pragma omp for  /*-- Work-sharing loop 2 --*/
    { ..... }

       .........

    #pragma omp for  /*-- Work-sharing loop N --*/
    { ..... }
}
```

Minimizar regiões paralelas

- Nesse exemplo, ao minimizar a criação de regiões paralelas, estamos evitando o custo de criação de n^2 regiões paralelas.
- É mantido apenas o custo de particionamento do *for* e a sincronização após seu término.
- Oferece maior oportunidade para uso de cache
- Região com maior contexto para otimizações de compilação

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

```
#pragma omp parallel
   for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           #pragma omp for
           for (k=0; k<n; k++)
               { .........}
```

Minimizar regiões paralelas

Aplicação: Even-Odd Sort Tamanho do array: 150000
SpeedUp: 35,280 / 20,794 = 1,69 (2 cores)

```
void evenOddSort(int A[], int begin, int end){
    for(int i = begin; i<end; i++){
        int first = i%2;
        #pragma omp parallel for
        for(int j = begin+first; j<end-1; j += 2){
            if(A[j]>A[j+1]){
                swap(A[j],A[j+1]);
            }
        }
    }
}
```

Minimizar regiões paralelas

Aplicação: Even-Odd Sort Tamanho do array: 150000
SpeedUp: 35,280 / 18,130 = 1,95 (2 cores)
Melhoria de quase 15%

```
void evenOddSort(int A[], int begin, int end){
    #pragma omp parallel
    for(int i = begin; i<end; i++){
        int first = i%2;
        #pragma omp for
        for(int j = begin+first; j<end-1; j += 2){
            if(A[j]>A[j+1]){
                swap(A[j],A[j+1]);
            }
        }
    }
}
```

Evite grandes regiões críticas

- Quanto maior a região crítica:
  - Maior a chance de uma thread precisar esperar por outra executando a região crítica;
  - Maior o tempo que uma thread precisa esperar ociosamente.
- **Objetivo:** Minimizar a computação dentro de regiões críticas.
  - Utilizar variáveis locais para armazenar valores temporários antes de atualizar variáveis compartilhadas.

Evite grandes regiões críticas

- Exemplo de computação crítica desnecessária:

```
#pragma omp parallel shared(a,b) private(c,d)
{
      ......
#pragma omp critical
     {
       a += 2 * c;
       c = d * d;
     }
} /*-- End of parallel region --*/
```

Variáveis privadas e compartilhadas

- É recomendável evitar variáveis compartilhadas sempre que possível:
  - Oportunidade para utilizar valores em registradores ou cache;
  - Evitar erro de acesso concorrente em variáveis compartilhadas (condição de disputa).

Variáveis privadas e compartilhadas

Condição de disputa no acesso à *x* (compartilhada por *default*).

```
void compute(int n)
{
  int    i;
  double h, x, sum;

  h = 1.0/(double) n;
  sum = 0.0;
 #pragma omp for reduction(+:sum) shared(h)
  for (i=1; i <= n; i++) {
      x = h * ((double)i - 0.5);
      sum += (1.0 / (1.0 + x*x));
  }
  pi = h * sum;
}
```

Variáveis privadas e compartilhadas

Variável de controle *i* apenas do loop paralelizado é privada por default.
Variável *j* foi definida como compartilhada (por *default*).

```
int i, j;
#pragma omp parallel for
for (i=0; i<n; i++)
    for (j=0; j<m; j++) {
        a[i][j] = compute(i,j);
    }
```

Variáveis privadas e compartilhadas

- Variáveis privadas não são inicializadas ao entrar na região paralela;
- Variáveis privadas não são atualizadas ao sair da região paralela;

```
void main ()
{
   ..............
   #pragma omp parallel for private(i,a,b)
   for (i=0; i<n; i++)
   {
         b++;
         a = b+i;
   } /*-- End of parallel for --*/
   c = a + b;
   ..............
}
```

Variáveis privadas e compartilhadas

Solução:
- firstprivate: inicializa variável privada com valor original.
- lastprivate: atualiza variável original com o valor final da local.

```
void main ()
{
   ..............
   #pragma omp parallel for private(i), firstprivate(b) \
           lastprivate(a,b)
   for (i=0; i<n; i++)
   {
       b++;
       a = b+i;
   } /*-- End of parallel for --*/
   c = a + b;
   ..............
}
```

Funções thread-safe

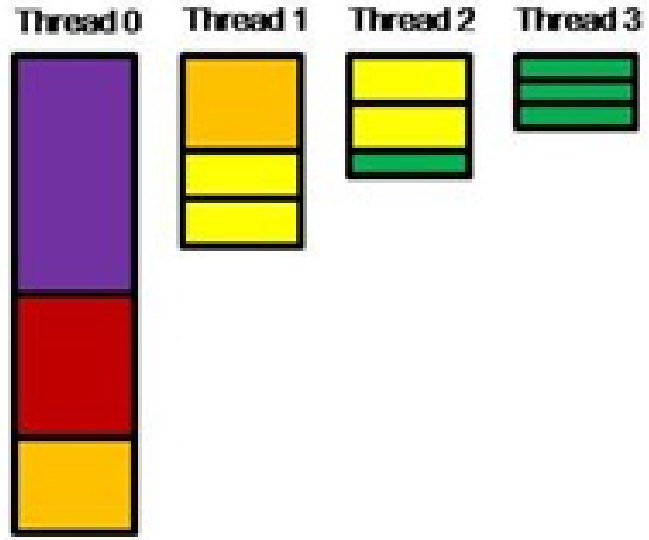
Uma função thread-safe é segura para ser executada em paralelo.

Exemplo de uma função que **não** é thread-safe:

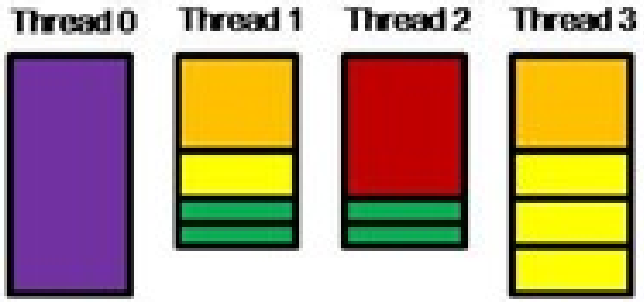
```
int icount;

void lib_func()
{
  icount++;
  do_lib_work();
}

main ()
{
   #pragma omp parallel
   {
     lib_func();
   } /*-- End of parallel region -- */
}
```

Balanceamento de carga

Problemas de balanceamento de carga ocorrem quando a carga de trabalho para cada thread não é homogênea.



(a) Unbalanced assignment of tasks to threads



(b) Balanced assignment of tasks to threads

Particionamento considerando apenas o número elementos para cada thread, sem considerar o custo de processamento de cada elemento.

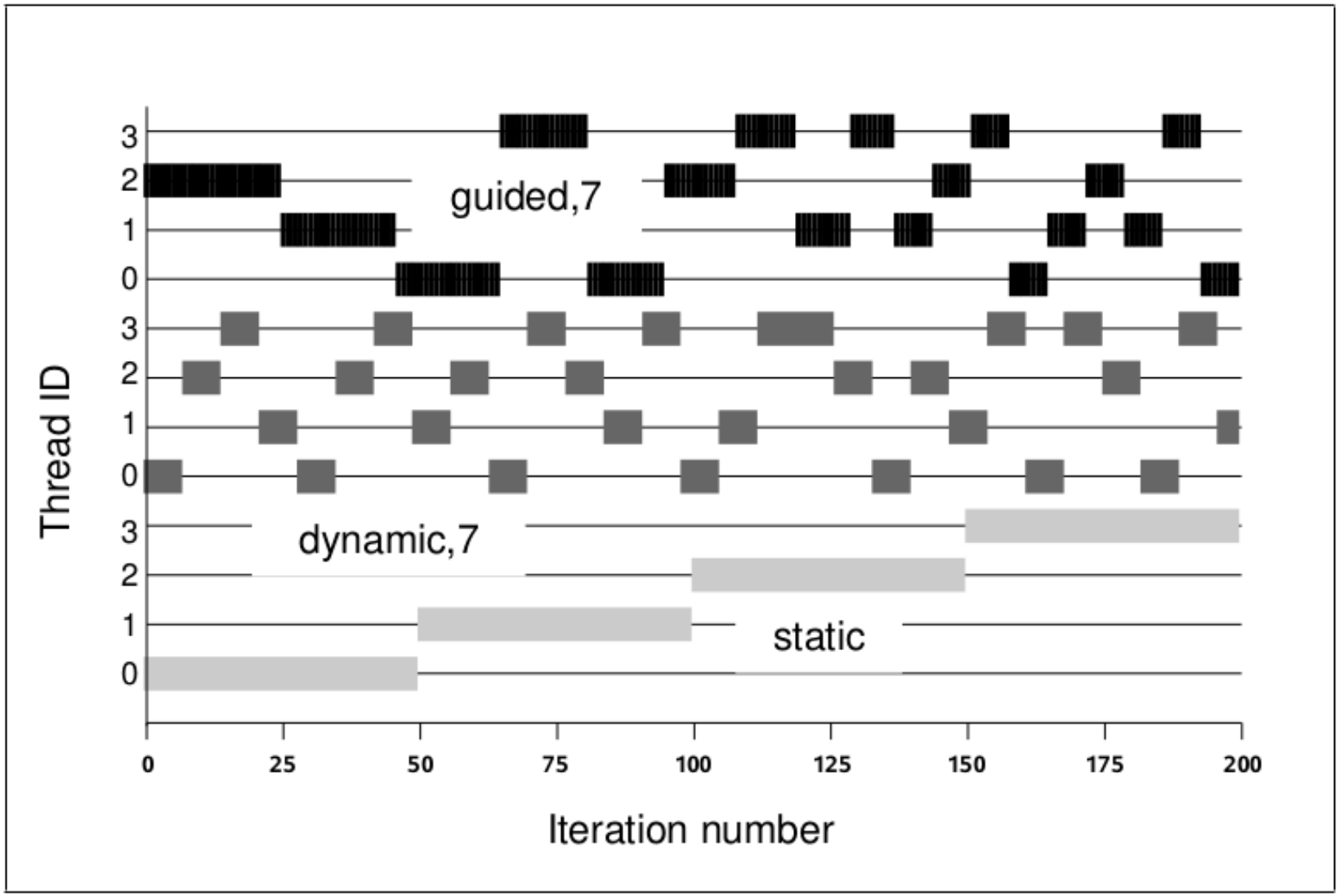
Balanceamento de carga

Solução para loops: utilização de escalonadores apropriados

Kind	Description
static	Divide the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. By default, chunk size is loop_count/number_of_threads.Set chunk to 1 to interleave the iterations.
dynamic	Use the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue. By default, the chunk size is 1. Be careful when using this scheduling type because of the extra overhead involved.
guided	Similar to dynamic scheduling, but the chunk size starts off large and decreases to better handle load imbalance between iterations. The optional chunk parameter specifies them minimum size chunk to use. By default the chunk size is approximately loop_count/number_of_threads.
auto	When schedule (auto) is specified, the decision regarding scheduling is delegated to the compiler. The programmer gives the compiler the freedom to choose any possible mapping of iterations to threads in the team.
runtime	Uses the OMP_schedule environment variable to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as would appear on the parallel construct.

Balanceamento de carga

Comparação entre os escalonadores possíveis:



Balanceamento de carga

- Escalonador **static**:
  - controle interno mais eficiente;
  - considera que a carga é homogênea;
  - sujeito à problemas de desbalanceamento de carga;

- Escalonadores **dynamic** e **guided**:
  - apresenta uma distribuição mais dinâmicos da carga de trabalho entre as threads;
  - maior custo de controle interno gerenciando o escalonamento;
  - oferece uma solução para o problema de desbalanceamento de carga (evitando que threads fiquem ociosas por muito tempo);