OpenMP

Notas de aulas baseada nos slides do professor Yogish Sabharwal, IIT Delhi

Curso:

Introduction to Parallel Programming in OpenMP

https://www.youtube.com/watch?v=a8R784VtXBg&list=PLJ5C_6qdAvBFMAko9JTyDJDIt1W48Sxmg

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      printf(“Hello world\n”);
   }
   return 0;
}
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      printf("Hello world\n");
   }
   return 0;
}
```

```
$ gcc -fopenmp hello.c -o hello
$ ./hello
Hello world
Hello world
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      printf("Hello world\n");
   }
   return 0;
}
```

```
$ export OMP_NUM_THREAD=4
$./hello
Hello world
Hello world
Hello world
Hello world
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      int numt = omp_get_num_threads();
      int tid = omp_get_thread_num();

      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      int numt = omp_get_num_threads();
      int tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      int numt = omp_get_num_threads();
      int tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 3 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
```

Variáveis privadas e compartilhadas

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

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
      int numt = omp_get_num_threads();
      int tid = omp_get_thread_num();

      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

O que acontece se declararmos numt e tid fora do bloco?

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

```
Hello from thread 1 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

Como corrigir?

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel private (tid) shared (numt)
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel private (tid) shared (numt)
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();

      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

Faça o código gerar erro com tid compartilhado.

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();
      for (int j = 0; j < 1000000000; j++);
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

Faça o código gerar erro com tid compartilhado.

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();
      for (int j = 0; j < 1000000000; j++);
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel private (tid) shared (numt)
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();
      for (int j = 0; j < 1000000000; j++);
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
Hello from thread 2 of 4
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

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

int main(int argc, char *argv[]) {

   int numt, tid; // fora do bloco paralelo

   #pragma omp parallel private (tid) shared (numt)
   {
      numt = omp_get_num_threads();
      tid = omp_get_thread_num();
      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

Como numt é compartilhado podemos inicializar fora do bloco?

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

int main(int argc, char *argv[]) {

   int numt = omp_get_num_threads();

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();

      printf(“Hello from thread %d of %d.\n”, tid, numt);
   }
   return 0;
}
```

Como numt é compartilhado podemos inicializar fora do bloco?

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

int main(int argc, char *argv[]) {

   int numt = omp_get_num_threads();

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 1
Hello from thread 0 of 1
Hello from thread 1 of 1
Hello from thread 3 of 1
```

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

int main(int argc, char *argv[]) {

   int numt = omp_get_num_threads();

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();

      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

Como resolver isso? Iniciando apenas uma vez.

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

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0)
         numt = omp_get_num_threads();
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

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

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0)
         numt = omp_get_num_threads();
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 3 of 4
```

```
Hello from thread 2 of -231321984
Hello from thread 3 of -231321984
Hello from thread 1 of 4
Hello from thread 0 of 4
```

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

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0)
         numt = omp_get_num_threads();
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

Como fazer o programa falhar várias vezes?

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

int main(int argc, char *argv[]) {
   int numt,j;
   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0){
         for (j=0; j<1000000000; j++);
         numt = omp_get_num_threads();
      }
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

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

int main(int argc, char *argv[]) {
   int numt,j;
   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0){
         for (j=0; j<1000000000; j++);
         numt = omp_get_num_threads();
      }
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
   return 0;
}
```

```
Hello from thread 2 of 4950
Hello from thread 3 of 4950
Hello from thread 1 of 4950
Hello from thread 0 of 4
```

```
… // use #pragma parallel (join) como mecanismo de sincronização
   int numt,j;

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0){
         for (j=0; j<1000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
```

```
… // use #pragma parallel (join) como mecanismo de sincronização
   int numt,j;

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      if (tid == 0){
         for (j=0; j<1000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   {
      int tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);
   }
```

```
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
```

Variáveis thread private

```
...
int tid;                          /* fazer o id persistente entre seções paralelas */
#pragma omp threadprivate(tid)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j;
      tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   printf("Hello from thread %d of %d.\n", tid, numt);

   return 0;
}
```

```
...
int tid;
#pragma omp threadprivate(tid)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j;
      tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   printf("Hello from thread %d of %d.\n", tid, numt);

   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
```

```
/* threadprivate só é aplicado a escopo de arquivos/variáveis estáticas */

static int glb;
#pragma omp threadprivate(glb)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp threadprivate(numt)


   ...

}
```

```
/* threadprivate só é aplicado a escopo de arquivos/variáveis estáticas */

static int glb;
#pragma omp threadprivate(glb)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp threadprivate(numt)


   ...

}
```

```
$ gcc -fopenmp hello.c
In function ‘main’:
error: automatic variable ‘numt’ cannot be ‘threadprivate’
```

```
...
int tid;                          /* Revisitando threadprivate */
#pragma omp threadprivate(tid)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j;
      tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   printf("Hello from thread %d of %d.\n", tid, numt);

   return 0;
}
```

```
...
int tid;                              /* Revisitando threadprivate */
#pragma omp threadprivate(tid)
int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j;
      tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
   }

   #pragma omp parallel default (shared)
   printf("Hello from thread %d of %d.\n", tid, numt);

   return 0;
}
```

Evite usar #pragma omp parallel para sincronizar os processos

```
...
/* Use barreira explícita para sincronização */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }

      #pragma omp barrier
      printf("Hello from thread %d of %d.\n", tid, numt);

   }


   return 0;
}
```

```
...
/* Use barreira explícita para sincronização */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid = omp_get_thread_num();
      if (tid == 0){
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }

      #pragma omp barrier
      printf("Hello from thread %d of %d.\n", tid, numt);

   }


   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
```

```
...
/* Use abordagem mais conveniente para trabalho que possa ser feito por uma thread */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp single
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
...
/* Use abordagem mais conveniente para trabalho que possa ser feito por uma thread */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp single
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
Hello from thread 2 of 4
Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 0 of 4
```

```
...
/* Ilustração de como a diretiva nowait remove a sincronização */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp single nowait
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
...
/* Ilustração de como a diretiva nowait remove a sincronização */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp single nowait
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
Hello from thread 2 of 4095
Hello from thread 3 of 4095
Hello from thread 1 of 4095
Hello from thread 0 of 4
```

```
...
/* Ilustração de como a diretiva master funciona */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp master
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
...
/* Ilustração de como a diretiva master funciona */

int main(int argc, char *argv[]) {
   int numt;
   #pragma omp parallel default (shared)
   {
      int j, tid;

      #pragma omp master
      {
         for (j = 0; j < 10000000000; j++);
         numt = omp_get_num_threads();
      }
      tid = omp_get_thread_num();
      printf("Hello from thread %d of %d.\n", tid, numt);

   }

   return 0;
}
```

```
Hello from thread 2 of 4095
Hello from thread 3 of 4095
Hello from thread 1 of 4095
Hello from thread 0 of 4
```

Diretiva master

- Indica que um bloco deve ser executado pelo thread master
- Outros pulam o bloco e continuam a execução

```
#pragma omp master
```

funções para medir o tempo decorrido

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

int main(int argc, char *argv[]) {

   printf("omp_get_wtime() = %g \n", omp_get_wtime());
   printf("omp_get_wtick() = %g \n", omp_get_wtick());

   return 0;
}
```

```
omp_get_wtime() = 6.98887e+06
omp_get_wtick() = 1e-09
```

compute a soma dos elementos do vetor

```
#include<omp.h>                                  /* Código sequencial */
#include<stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main(int argc, char *argv[]) {









   return 0;
}
```

```
#include<omp.h>
#include<stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main(int argc, char *argv[]) {
   int i;

   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;









   return 0;
}
```

```
#include<omp.h>
#include<stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main(int argc, char *argv[]) {
   int i;

   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;


   for (i = 0; i < ARR_SIZE; i++)
      sum += a[i];




   printf(“Soma dos elementos do vetor: %d\n”, sum);

   return 0;
}
```

```
#include<omp.h>
#include<stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main(int argc, char *argv[]) {
   int i;
   double t1, t2;
   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;

   t1 = omp_get_wtime();
   for (i = 0; i < ARR_SIZE; i++)
      sum += a[i];

   t2 = omp_get_wtime();


   printf("Soma dos elementos do vetor: %d. Tempo = %g \n", sum, t2-t1);

   return 0;
}
```

```
#include<omp.h>
#include<stdio.h>
#define ARR_SIZE 1000000000
int a[ARR_SIZE];
int main(int argc, char *argv[]) {
   int i;
   double t1, t2;
   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;

   t1 = omp_get_wtime();
   for (i = 0; i < ARR_SIZE; i++)
      sum += a[i];

   t2 = omp_get_wtime();


   printf("Soma dos elementos do vetor: %d. Tempo = %g \n", sum, t2-t1);

   return 0;
}
```

```
Soma dos elementos do vetor:1000000000. Tempo = 13.8598
```

```
/* Soma dos elementos do vetor. Código paralelo */

   int i, tid, numt, sum = 0;
   double t1, t2;

   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;

   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();
      for (i = 0; i < ARR_SIZE; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
   printf("Soma dos elementos do vetor: %d. Tempo = %g \n", sum, t2-t1);
```

```
/* Soma dos elementos do vetor. Código paralelo */

   int i, tid, numt, sum = 0;
   double t1, t2;

   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;

   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();
      for (i = 0; i < ARR_SIZE; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
   printf("Soma dos elementos do vetor: %d. Tempo = %g \n", sum, t2-t1);
```

```
Soma dos elementos do vetor: 937329534. Tempo = 31.1103
```

```
/* Soma dos elementos do vetor. Código paralelo */

   int i, tid, numt, sum = 0;
   double t1, t2;

   for (i = 0; i < ARR_SIZE; i++)
      a[i] = 1;

   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();
      for (i = 0; i < ARR_SIZE; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
   printf("Soma dos elementos do vetor: %d. Tempo = %g \n", sum, t2-t1);
```

```
Soma sequencial:100000000.
Tempo = 13.8598
```

```
Soma dos elementos do vetor: 937329534. Tempo = 31.1103
```

Distribuição da carga de trabalho

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid == numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <= to; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid == numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
tid = 2, numt = 4, range from = 50000000, to = 749999999
tid = 0, numt = 4, range from = 0, to = 249999999
tid = 1, numt = 4, range from = 25000000, to = 499999999
tid = 3, numt = 4, range from = 75000000, to = 999999999
Soma dos elementos do vetor:290073353. Tempo = 4.0214
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid == numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
Soma sequencial:1000000000.
Tempo = 13.8598
```

```
tid = 2, numt = 4, range from = 50000000, to = 749999999
tid = 0, numt = 4, range from = 0, to = 249999999
tid = 1, numt = 4, range from = 25000000, to = 499999999
tid = 3, numt = 4, range from = 75000000, to = 999999999
Soma dos elementos do vetor:290073353. Tempo = 4.0214
```

Seção crítica

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid = numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <= to; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid = numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <= to; i++)
         #pragma omp critical
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = (ARRAY_SIZE/numt) * tid;
      to = (ARRAY_SIZE/numt) * (tid + 1) - 1;
      if (tid = numt-1)
         to = ARRAY_SIZE-1;

      printf("tid = %d, numt = %d, range from = %d, to = %d\n", tid, numt, from, to);
      for (i = from; i <
         #pragma omp cri
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
tid = 2, numt = 4, range from = 50000000, to = 749999999
tid = 0, numt = 4, range from = 0, to = 249999999
tid = 1, numt = 4, range from = 25000000, to = 499999999
tid = 3, numt = 4, range from = 75000000, to = 999999999
Soma dos elementos do vetor:1000000000. Tempo = 671.0214
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to, psum = 0;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = ...
      to = ...

      for (i = from; i <= to; i++)
         psum += a[i];

      #pragma omp critical
      sum += psum;                    // Sincronizar o somatório das somas parciais
   }
   t2 = omp_get_wtime();
```

```
   int i, tid, numt, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) private(i, tid)
   {
      int from, to, psum = 0;
      tid = omp_get_thread_num();
      numt = omp_get_num_threads();

      from = ...
      to = ...

      for (i = from; i <= to; i++)
         psum += a[i];

      #pragma omp critical
      sum += psum;                    // Sincronizar o somatório das somas parciais
   }
   t2 = omp_get_wtime();
```

```
Soma dos elementos do vetor:1000000000. Tempo = 3.6542
```

Diretivas omp for

for paralelo

```
   int i, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared)
   {
      int psum = 0;

      #pragma omp for
      for (i = 0; i <= ARRAY_SIZE; i++) // variável i é automaticamente privada
         psum += a[i];

      #pragma omp critical
      sum += psum;                      // Sincronizar o somatório das somas parciais
   }
   t2 = omp_get_wtime();
```

Redução

```
   int i, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) reduction(+:sum)
   {
      #pragma omp for
      for (i = 0; i <= ARRAY_SIZE; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
   int i, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared) reduction(+:sum)
   {
      #pragma omp for
      for (i = 0; i <= ARRAY_SIZE; i++)
         sum += a[i];
   }
   t2 = omp_get_wtime();
```

```
Soma dos elementos do vetor:1000000000. Tempo = 3.48449
```

```
   int i, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel default(shared)
      #pragma omp for reduction(+:sum)
      for (i = 0; i <= ARRAY_SIZE; i++)
         sum += a[i];

   t2 = omp_get_wtime();
```

```
   int i, sum = 0;
   double t1, t2;
...
   t1 = omp_get_wtime();
   #pragma omp parallel for default(shared) reduction(+:sum)
   for (i = 0; i <= ARRAY_SIZE; i++)
      sum += a[i];

   t2 = omp_get_wtime();
```

Tarefas

```
#define ARRAY_SIZE 600
#define STEP_SIZE 100
int a[ARRAY_SIZE];
```

```
int i, sum = 0;
double t1, t2;
...
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i <= ARRAY_SIZE; i += STEP_SIZE)
   {
      int j, start = i, end = i + STEP_SIZE - 1;
      printf("Computando soma(%d, %d), tid=%d\n", start, end, omp_get_num_thread());
      #pragma omp task
      {
         int psum = 0;
         printf("Tarefa computando soma(%d, %d), tid=%d\n", start, end, omp_get_num_thread());
         for (j = start; j <= end; j++)
            psum += a[j];
         #pragma omp critical
         sum += psum;
      }
   }
}
printf("Sum=%d\n", sum);
```

```
int i, sum = 0;
double t1, t2;
...
#pragma omp parallel
{
   #pragma omp for
   for (i = 0; i <= ARRAY_SIZE; i += STEP_SIZE)
   {
      int j, start =
      printf("Computa
      #pragma omp tas
      {
         int psum = 0
         printf("Tare                                                    ead());
         for (j = sta
            psum += a
         #pragma omp
         sum += psum;
      }
   }
}
printf("Sum=%d\n", sum);
```

```
#define ARRAY_SIZE 600
#define STEP_SIZE 100
int a[ARRAY_SIZE];
```

```
Computando soma(0, 99), tid=0
Computando soma(100, 199), tid=0
Tarefa computando soma(0, 99), tid=3
Computando soma(400, 499), tid=2
Tarefa computando soma(100, 199), tid=0
Computando soma(200, 299), tid=1
Computando soma(500, 599), tid=2
Tarefa computando soma(500, 599), tid=3
Tarefa computando soma(400, 499), tid=0
Tarefa computando soma(200, 299), tid=2
Computando soma(300, 399), tid=1
Tarefa computando soma(300, 399), tid=2
Sum = 600
```

Usando tarefas para divisão recursiva

```
#define ARRAY_SIZE 600
#define STEP_SIZE 100
int a[ARRAY_SIZE];

int main(int argc, char *argv[])
{
   int i, sum=0;

   for (i = 0; i < ARRAY_SIZE; i++)
      a[i] = 1;

   #pragma omp parallel
   #pragma omp single
   sum = do_sum(0, ARRAY_SIZE-1);
   printf("Sum = %d\n", sum);

   return 0;
}
```

```
int do_sum(int start, int end){
   int mid, x, y, res;

   if (end == start)
      res = a[start];
   else {
      mid = (start + end) / 2;

      #pragma omp task shared(x)
      x = do_sum(start, mid);

      #pragma omp task shared(y)
      y = do_sum(mid+1, end);

      #pragma omp taskwait
      res = x + y;
   }
   return res;
}
```

```
/* taskwait não é barreira */
int i, sum = 0;
#pragma omp parallel
{
   #pragma omp for nowait
   for (i = 0; i <= ARRAY_SIZE; i += STEP_SIZE)
   {
      int j, start = i, end = i + STEP_SIZE - 1;
      printf("Computando soma...");
      #pragma omp task
      {
         int psum = 0;
         printf("Tarefa computando...");
         for (j = start; j <= end; j++)
            psum += a[j];
         #pragma omp critical
         sum += psum;
      }
   }
   #pragma omp taskwait
   #pragma omp master
   printf("Sum=%d\n", sum);
}
```

```
#define ARRAY_SIZE 600
#define STEP_SIZE 100
int a[ARRAY_SIZE];
```

Locks

```
int i, sum = 0, prod = 1;

#pragma omp parallel default(shared)
{
   int psum = 0, pprod = 1

   #pragma omp for
   for (i = 0; i < ARRAY_SIZE; i++) {
      psum += a[i];
      pprod *= a[i];
   }

   #pragma omp critical
   sum += psum;

   #pragma omp critical
   prod *= pprod;
}
```

```
#pragma omp parallel default(shared)
{
   int psum = 0, pprod = 1
   #pragma omp for
   for (i = 0; i < ARRAY_SIZE; i++) {
      psum += a[i];   pprod *= a[i];
   }
   #pragma omp critical(section1)
   {
      printf("In CS 1\n");
      for (j = 0; j < 1000000000; j++);
      sum += psum;
      printf("Out CS 1\n");
   }
   #pragma omp critical
   {
      printf("In CS 2\n");
      for (j = 0; j < 1000000000; j++);
      prod *= pprod;
      printf("Out CS 2\n");
   }
}
```

```
#pragma omp parallel default(shared)
{
   int psum = 0, pprod = 1
   #pragma omp for
   for (i = 0; i < ARRAY_SIZE; i++) {
      psum += a[i];   pprod *= a[i];
   }
   #pragma omp critical(section1)
   {
      printf("In CS 1\n");
      for (j = 0; j < 1000000000; j++);
      sum += psum;
      printf("Out CS 1\n");
   }
   #pragma omp critical
   {
      printf("In CS 2\n");
      for (j = 0; j < 1000000000; j++);
      prod *= pprod;
      printf("Out CS 2\n");
   }
}
```

```
In CS 1
Out CS 1
In CS 1
In CS 2
Out CS 2
Out CS 2
In CS 2
Out CS 2
```

https://www.youtube.com/watch?v=GX1DpC37LeM&list=PLJ5C_6qdAvBFMAko9JTyDJDIt1W48Sxmg&index=22

https://www.youtube.com/channel/UCNp-uk36t-bnvHr3A_snQtg/videos