Loop Parallelism with OpenMP

OpenMP- A pragma-based approach

```
double res[10000];

for (i=0 ; i<10000 ; i++)
   compute(&res[i]) ;
```

```
#include "openmp.h"
double res[10000];
#pragma omp parallel for
for (i=0 ; i<10000 ; i++)
   compute(&res[i]) ;
```

A simple flag at compile-time enables or disables the parallelism:

- gcc –fopenmp foo.c –o foo
- Export OMP_NUM_THREADS=4
- ./foo

One more example

```
double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int th_id = omp_get_thread_num();
   compute(th_id, A);
}
 printf("Done.");
```

One more example

```
double A[10000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int th_id = omp_get_thread_num();
   compute(th_id, A);
}
 printf("Done.");
```

th_id is private.

One more example

```
double A[10000];
int th_id;
omp_set_num_threads(4);
#pragma omp parallel
{
  th_id = omp_get_thread_num();
  compute(th_id, A);
}
 printf("Done.");
```

th_id is shared.

Remember the section(s)?

```
int N = 10000;
int data[N];
init_from_file( data );      // Initialize the data
#pragma omp parallel num_threads(4)
{
#pragma omp sections
{
#pragma omp section
    compute(0,N,data) ;
#pragma omp section
    compute(1,N,data) ;
#pragma omp section
    compute(2,N,data) ;
#pragma omp section
    compute(3,N,data) ;
} }
print_vector( data );
```

```
compute(int index, int N, int* data ) {
   int i_start, i_end, i ;

   i_start = index*N/4+1;
   i_end  = (index+1)*N/4;
   printf("Working on data[%d, %d]\n",
                          i_start,i_end);
   for (i=i_start; i<i_end;i++)
      data[i] = i*i

}
```

Remember the section(s)?

```
int N = 10000;
int data[N];
init_from_file( data );      // Initialize the data
#pragma omp parallel num_threads(4)
{
#pragma omp sections
{
#pragma omp section
    compute(0,N,data) ;
#pragma omp section
    compute(1,N,data) ;
#pragma omp section
    compute(2,N,data) ;
#pragma omp section
    compute(3,N,data) ;
} }
print_vector( data );
```

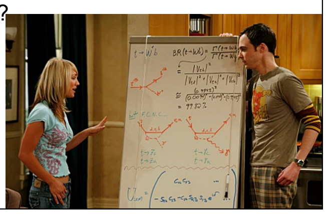
```
compute(int index, int N, int* data ) {
    int i_start, i_end, i ;

    i_start = index*N/4+1;
    i_end  = (index+1)*N/4;
    printf("Working on data[%d, %d]\n",
                                   i_start,i_end);
    for (i=i_start; i<i_end;i++)
       data[i] = i*i
}
```

NO CONFLICT

Agenda

- More about the control of the concurrent accesses to the variables.
- How do you schedule the iterations of a loop among the threads?



compute( ) reloaded

```
compute(int index, int N, int* data ) {
    int i_start, i_end, i ;
    int norm;
    i_start = index*N/4+1;
    i_end  = (index+1)*N/4;
    printf("Working on data[%d, %d]\n",
                                   i_start,i_end);
    for (i=i_start; i<i_end;i++) {
       norm += data[i]*data[i];
    }
}
```

Is there a problem?

Remember the section(s)?

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );      // Initialize the data
#pragma omp parallel num_threads(4)  private(norm)
{
#pragma omp sections
{
#pragma omp section
    compute(0,N,data) ;
#pragma omp section
    compute(1,N,data) ;
#pragma omp section
    compute(2,N,data) ;
#pragma omp section
    compute(3,N,data) ;
} }
print_vector( data );
```

```
compute(int index, int N, int* data ) {
    int i_start, i_end, i ;

    i_start = index*N/4+1;
    i_end  = (index+1)*N/4;
    printf("Working on data[%d, %d]\n",
                                   i_start,i_end);
    for (i=i_start; i<i_end;i++)
       norm += data[i]*data[i];
}
```

This will not work!

- Each thread owns a private copy of `norm`.
- Each copy is free'd when the threads finish.
- The global var `norm` that remains when only the master thread keeps running does not know anything about the former local copies.

Firstprivate/lastprivate

- The pragma FIRSTPRIVATE(var1) creates private versions of var1, copying the initial value of the global version.
- The pragma LASTPRIVATE(var1) creates private versions of var1, copying the final value(*) to the global version.
  - (*) as defined by the sequential execution.

Remember the section(s)?

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );     // Initialize the data
#pragma omp parallel num_threads(4) lastprivate(norm)
{
#pragma omp sections
{
#pragma omp section
    compute(0,N,data) ;
#pragma omp section
    compute(1,N,data) ;
#pragma omp section
    compute(2,N,data) ;
#pragma omp section
    compute(3,N,data) ;
} }
print_vector( data );
```

```
compute(int index, int N, int* data ) {
   int i_start, i_end, i ;

   i_start = index*N/4+1;
   i_end  = (index+1)*N/4;
   printf("Working on data[%d, %d]\n",
                                i_start,i_end);
   for (i=i_start; i<i_end;i++)
      norm += data[i]*data[i];

}
```

The final problem

- This still does not work....
  - You do not want the last (partial)local value of the norm,
  - What you want is the sum of all the (partial) local values.
- This is called a reduction.
  - `+`is the reduction operator.
  - You can also use *, or, and, MAX, MIN.
  - OpenMP syntax: reduction(+:norm)

Remember the section(s)?

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );     // Initialize the data
#pragma omp parallel num_threads(4) reduction(+:norm)
{
#pragma omp sections
{
#pragma omp section
    compute(0,N,data) ;
#pragma omp section
    compute(1,N,data) ;
#pragma omp section
    compute(2,N,data) ;
#pragma omp section
    compute(3,N,data) ;
} }
print_vector( data );
```

```
compute(int index, int N, int* data ) {
   int i_start, i_end, i ;

   i_start = index*N/4+1;
   i_end  = (index+1)*N/4;
   printf("Working on data[%d, %d]\n",
                                i_start,i_end);
   for (i=i_start; i<i_end;i++)
      norm += data[i]*data[i];

}
```

Parallel for

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );     // Initialize the data
#pragma omp parallel num_threads(4) reduction
   (+:norm)
{
#pragma omp for
  for (i=0; i<N;i++)
    norm += data[i]*data[i];
}
print_vector( data );
```

This pragma automatically distributes the iterations of the loop among the threads

Parallel for

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );     // Initialize the data
#pragma omp parallel num_threads(4) reduction
   (+:norm)
{
#pragma omp for
  for (i=0; i<N;i++)
    norm += data[i]*data[i];
}
print_vector( data );
```

Beware the limits of the loop!

No write conflict!

Parallel for – short version

```
int N = 10000;
int data[N]; int norm = 0;
init_from_file( data );     // Initialize the data
#pragma omp parallel for num_threads(4) reduction
   (+:norm)
  for (i=0; i<N;i++)
    norm += data[i]*data[i];
print_vector( data );
```

Nested loops

- Let us consider a matrix product
  - 3 nested loops i, j, k.
- The 2 external loops (i, j) can be run in parallel.
  - Actually, the 3rd one also (Reduce)
- So where do you put the “parallel for”?

Options

```
for (i=0 ; i<N ; i++) {

  for (j=0 ; j<N ; j++) {
    C[i][j] = 0;

    for (k=0 ; k<N; k++)
      C[i][j] +=  A[i][k]*B[][j];

  }
}
```

Option 1

```
#pragma omp parallel for
for (i=0 ; i<N ; i++) {

  for (j=0 ; j<N ; j++) {
    C[i][j] = 0;

    for (k=0 ; k<N; k++)
      C[i][j] +=  A[i][k]*B[][j];

  }
}
```

Option 1

```
#pragma omp parallel for private(j,k)
for (i=0 ; i<N ; i++) {

  for (j=0 ; j<N ; j++) {
    C[i][j] = 0;

    for (k=0 ; k<N; k++)
      C[i][j] +=  A[i][k]*B[][j];

  }
}
```

Beware the Private!

Option 2

```
for (i=0 ; i<N ; i++) {
#pragma omp parallel for private(k)
  for (j=0 ; j<N ; j++) {
    C[i][j] = 0;

    for (k=0 ; k<N; k++)
      C[i][j] +=  A[i][k]*B[][j];

  }
}
```

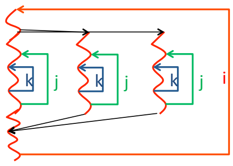
Option 2

```
for (i=0 ; i<N ; i++) {
#pragma omp parallel for private(k)
  for (j=0 ; j<N ; j++) {
    C[i][j] = 0;

    for (k=0 ; k<N; k++)
      C[i][j] +=  A[i][k]*B[][j];

  }
}
```

Beware the synch!



Option 3

```
for (i=0 ; i<N ; i++) {
#pragma omp parallel for private(k)
  for (j=0 ; j<N ; j++) {
    double tmp= 0;
#pragma omp parallel for reduction(+:tmp)
    for (k=0 ; k<N; k++)
      tmp +=  A[i][k]*B[][j];
    C[i][j] = tmp;
  }
}
```

Options 1+2+3

```
#pragma omp parallel for private(j,k)
for (i=0 ; i<N ; i++) {
#pragma omp parallel for
  for (j=0 ; j<N ; j++) {
    double tmp= 0;
#pragma omp parallel for reduction(+:tmp)
    for (k=0 ; k<N; k++)
      tmp +=  A[i][k]*B[][j];
    C[i][j] = tmp;
  }
}
```

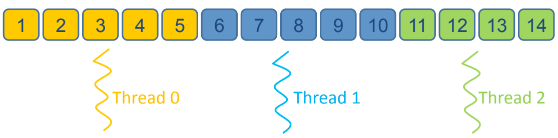
Distribution of the iterations

- How do you know which thread runs which values (between 0 and N-1) of the loop?
  - With sections the mapping was explicitly computed by the programmer!
- Do you really want to know?
  - OpenMP does it for you.

Small example

N = 14 iterations
P = 3 threads

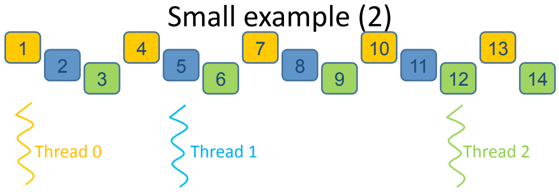
Small example



Each thread runs a block of approx. B=N/p +1 contiguous iterations:

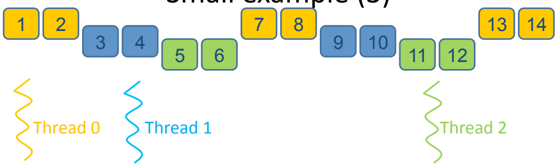
- the 1st N%p threads run B iterations,
- the last threads run B-1 iterations

Small example (2)



The allocation of iterations is cyclic (Round-Robin).
The thread j runs the iterations i such that i-1%p == j. They are not contiguous.
Each thread runs approx. B iterações.

Small example (3)



Round Robin of de blocks of K iterations among the threads – cyclic blocks.
The thread j runs the iterations j such that (i-1)/K %p == j. The blocks are not contiguous.

Pros / Cons

Grouping the iterations in block increases the locality.

Having small groups balances the load.

You want to increase K

You want to decrease K

Controlling the mapping with OpenMP

- You use schedule(static, K), together with "parallel for":

```
#pragma omp parallel for schedule(static, 5)
for (i=0; i<N; i++) { //...
```

- The first parameter specifies the schedule.
  - static, dynamic, guided
- The second parameter sets the size of the block.

Conclusion: using schedule()

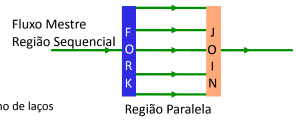
- You always should use "Schedule".
- You should experiment to find the right value for K.
- If the computation is homogeneous, then schedule(static, N/p) should do fine.
  - This is the default.

Próxima aula

Extra stuff

Paralelismo de Laços

- # pragma omp parallel for
- for (i = 0; i < n; i++){
- c[i] = a[i] + b[i];
- }



- Demonstração: exemplo de paralelismo de laços

- Restrições de acesso:

```
int fator = 0.5
#pragma omp parallel for private( fator )
        for (i = 0; i < N; i++){
                c[i] = fator * a[i];
        }
```

```
#pragma omp parallel for private( j )
        for (i = 0; i < N; i++){
                for(j = 0; j < N; j++){
                        c[i] = a[i] + b[j];
                }
        }
```

37

*firstprivate, lastprivate