OpenMP
Shared-Memory Parallel Programming
Part. II

Orian Louant
orian.louant@uliege.be

October 6, 2020

Last Time

- The `parallel` construct creates a team of threads and starts parallel execution.
- The `for` construct specifies that the iterations of a loop should be distributed among the threads in the team.
- Data-sharing attribute clauses apply to the `parallel` and `for` constructs and allow a to control the data-sharing attributes of variables referenced in these constructs.

Synchronization

Synchronization

Synchronization ensures that two or more threads do not simultaneously execute some part of the program.

Synchronization may be needed for various reasons:

- makes sure that a particular operation is only executed one time
- to avoid conflicts when accessing shared data
- ensure the order in which tasks are executed

Barrier

A `barrier` directive is a synchronization point at which the threads in a parallel region will wait until all other threads in that section reach the same point.

- When a first thread reaches the barrier, it waits
- When a second thread reaches the barrier, it does the same thing and so on
- When the last thread reaches the barrier, all the threads resume execution

Barrier

```
#pragma omp parallel private(tid, neighb)
{
  tid = omp_get_thread_num();
  neighb = tid - 1;

  if (tid == 0)
    neighb = omp_get_num_threads() - 1;

  a[tid] = a[tid] * 3.5;

  #pragma omp barrier

  b[tid] = a[neighb] + c;
}
```

Barrier

```
#pragma omp parallel private(tid, neighb)
{
  tid = omp_get_thread_num();
  neighb = tid - 1;

  if (tid == 0)
    neighb = omp_get_num_threads() - 1;

  a[tid] = a[tid] * 3.5;

  #pragma omp barrier

  b[tid] = a[neighb] + c;
}
```

ensures that the neighbour value of a is set to the correct value

Implicit Barrier

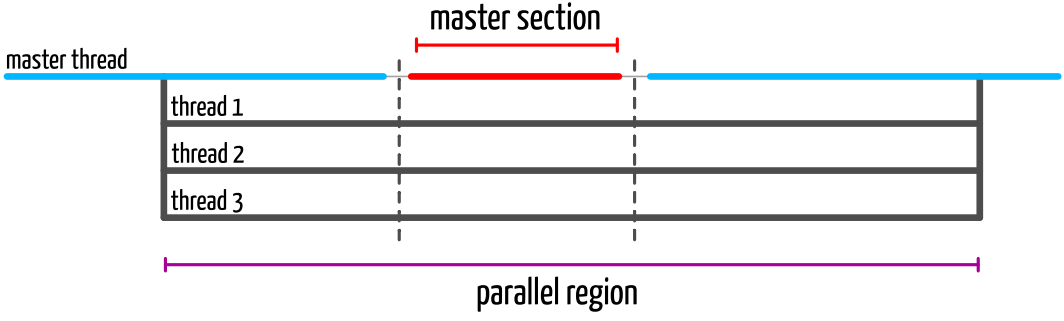
Some constructs in OpenMP have an implicit barrier. This is the case for the `parallel` and `for` constructs.

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

    // ...

  } <-- Implicit barrier, wait for all the threads to finish their iterations

  // ...

} <-- Implicit barrier, wait for all the threads to join
```

Master Directive

A `master` construct specifies a block of code that should be executed only by the master thread of the team.

```
#pragma omp master
  structured-block
```



Hello World, Master Edition

```
#pragma omp parallel
{
  printf("Hello, I'm thread %d\n",
         omp_get_thread_num());

  #pragma omp master
  {
    printf("There is %d threads in the team\n",
           omp_get_num_threads());
  }
}
```

Hello World, Master Edition

```
#pragma omp parallel
{
  printf("Hello, I'm thread %d\n",
         omp_get_thread_num());

  #pragma omp master
  {
    printf("There is %d threads in the team\n",
           omp_get_num_threads());
  }
}
```

only the master thread print the number of threads in the team

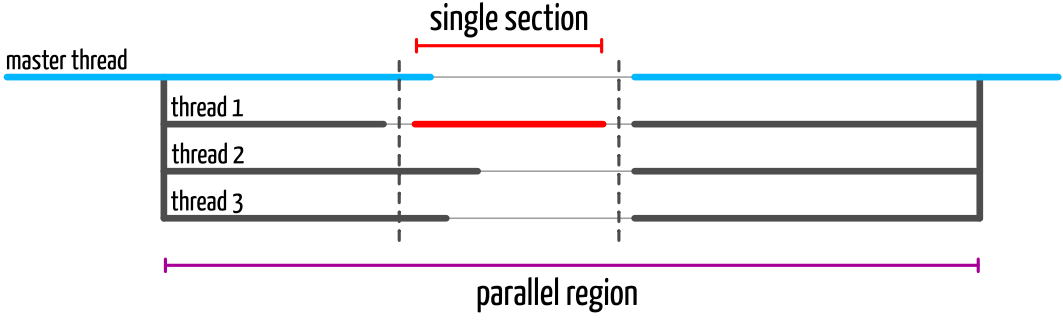
Hello World, Master Edition

```
 $ gcc -fopenmp -o example omp_helloworld_master.c
 $ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 3
Hello, I'm thread 0
There is 4 threads in the team
Hello, I'm thread 2
Hello, I'm thread 1
```

Single Directive

A `single` directive is executed by only one of the threads in the team (not necessarily the master thread). There is an implicit barrier at the end.

```
#pragma omp single
  structured-block
```



Hello World, Single Edition

```
#pragma omp parallel shared(nthreads) private(tid)
{
  #pragma omp single
  {
    nthreads = omp_get_num_threads();
  }

  tid = omp_get_thread_num();

  printf("Hello, I'm thread %d of %d.\n", tid, nthreads);
}
```

Hello World, Single Edition

```
#pragma omp parallel shared(nthreads) private(tid)
{
  #pragma omp single
  {
    nthreads = omp_get_num_threads();
  }

  tid = omp_get_thread_num();

  printf("Hello, I'm thread %d of %d.\n", tid, nthreads);
}
```

only one thread get the number of threads in the team

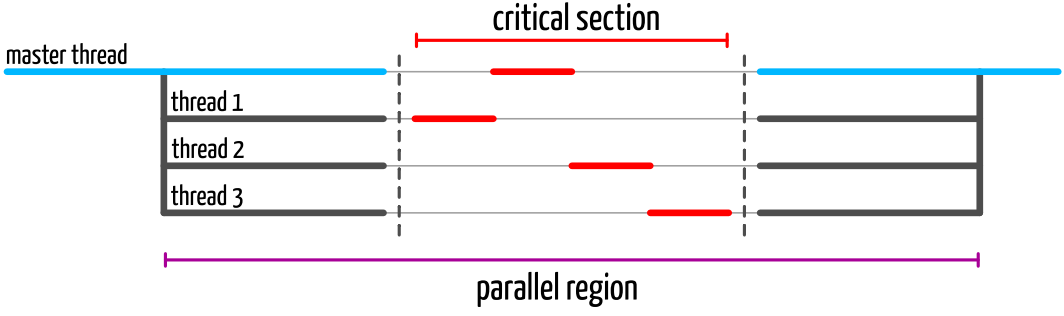
Hello World, Single Edition

```
 $ gcc -fopenmp -o example omp_helloworld_single.c
 $ OMP_NUM_THREADS=4 ./example
Hello, I'm thread 0 of 4.
Hello, I'm thread 3 of 4.
Hello, I'm thread 1 of 4.
Hello, I'm thread 2 of 4.
```

Critical Section

A `critical` section restricts execution of the associated structured block to one thread at a time.

```
#pragma omp critical
  structured-block
```



Critical Section

```
#pragma omp parallel shared(sum) private(tid, local_sum)
{
  tid = omp_get_thread_num();
  local_sum = 0;

  #pragma omp for
  for (int i = 0; i < n; ++i)
      local_sum += a[i];

  #pragma omp critical
  {
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
  }
}

printf("Sum after parallel region: %d.\n", sum);
```

Critical Section

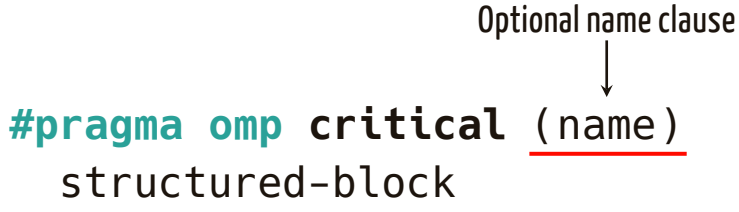
```
#pragma omp parallel shared(sum) private(tid, local_sum)
{
  tid = omp_get_thread_num();
  local_sum = 0;

  #pragma omp for
  for (int i = 0; i < n; ++i)
      local_sum += a[i];

  #pragma omp critical
  {
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
  }
}

printf("Sum after parallel region: %d.\n", sum);
```

Critical section to update the global sum.
Without the critical section,
there is a potential data race here

Critical Section

```
 $ gcc -fopenmp -o example omp_critical.c
 $ OMP_NUM_THREADS=4 ./example
Thread 0: local sum = 300, sum = 300.
Thread 3: local sum = 2175, sum = 2475.
Thread 1: local sum = 925, sum = 3400.
Thread 2: local sum = 1550, sum = 4950.
Sum after parallel region: 4950.
```

Named Critical Section



- A thread waits at the beginning of a critical section until no other thread is executing a critical section with the same name
- All unnamed critical directives map to the same name
- Critical section names are global to the program

Named Critical Section

```
#pragma omp critical (sum)
{
  sum += local_sum;
  printf("Thread %d: local sum = %d, sum = %d.\n",
          tid, local_sum, sum);
}

#pragma omp critical (max)
{
  max = MAX(max, local_max);
  printf("Thread %d: local max = %d, max = %d.\n",
          tid, local_max, max);
}
```

Named Critical Section

```
#pragma omp critical (sum)
{
  sum += local_sum;
  printf("Thread %d: local sum = %d, sum = %d.\n",
         tid, local_sum, sum);
}

#pragma omp critical (max)
{
  max = MAX(max, local_max);
  printf("Thread %d: local max = %d, max = %d.\n",
         tid, local_max, max);
}
```

First critical section for the global sum

Named Critical Section

```
#pragma omp critical (sum)
{
  sum += local_sum;
  printf("Thread %d: local sum = %d, sum = %d.\n",
         tid, local_sum, sum);
}

#pragma omp critical (max)
{
  max = MAX(max, local_max);
  printf("Thread %d: local max = %d, max = %d.\n",
         tid, local_max, max);
}
```

First critical section for the global sum

Second critical section for the global maximum.
A thread can be in the first section while an other is in the second one

Named Critical Section

```
 $ gcc -fopenmp -o example omp_critical_named.c
 $ OMP_NUM_THREADS=4 ./example
Thread 3: local sum = 2175, sum = 2175.
Thread 2: local sum = 1550, sum = 3725.
Thread 0: local sum = 300, sum = 4025.
Thread 1: local sum = 925, sum = 4950.
Thread 3: local max = 99, max = 99.
Thread 2: local max = 74, max = 99.
Thread 0: local max = 24, max = 99.
Thread 1: local max = 49, max = 99.
Sum after parallel region: 4950.
Max after parallel region: 99.
```

The nowait Clause

```
//...

  #pragma omp for
  for (int i = 0; i < n; ++i) {
      local_sum += a[i];
  }

  #pragma omp critical
  {
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
  }
}

// ...
```

There is an implicit barrier here

The nowait Clause

```
//...

  #pragma omp for
  for (int i = 0; i < n; ++i) {
      local_sum += a[i];
  }

  #pragma omp critical
  {
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
  }
}

// ...
```

There is no need to wait for the other threads to finish the iterations to execute the critical section

The `nowait` Clause

```
//...

  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      local_sum += a[i];
  }

  #pragma omp critical
  {
    sum += local_sum;
    printf("Thread %d: local sum = %d, sum = %d.\n",
            tid, local_sum, sum);
  }
}

// ...
```

We add a `nowait` clause to the directive

The implicit barrier at the end of the loop is lifted

The `nowait` Clause

The `nowait` clause applied to a `for` construct remove the implicit barrier at the end of the construct.

```
#pragma omp for nowait
  structured-block
```

The `nowait` clause can also be applied to a `single` directive.

```
#pragma omp single nowait
  structured-block
```

The `nowait` Clause

The `nowait` clause can also be convenient when two loops are independent.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      d[i] = a[i] + b[i];
  }

  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      e[i] = a[i] + c[i];
  }
}
```

The `nowait` Clause

The `nowait` clause can also be convenient when two loops are independent.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      d[i] = a[i] + b[i];
  }

  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      e[i] = a[i] + c[i];
  }
}
```

No barrier at the end of the loop

The threads start the iterations of this loop as soon as they finish their work in the first loop

The `nowait` Clause

The `nowait` clause can also be convenient when two loops are independent.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      d[i] = a[i] + b[i];
  }

  #pragma omp for nowait
  for (int i = 0; i < n; ++i) {
      e[i] = a[i] + c[i];
  }
}
```

← Implicit barrier at the end of the `parallel` region

Reduction

The `reduction` clause avoid data races when summing or combining values. This clause can be applied to the `parallel` and `for` constructs

```
reduction(op:list)
```

`op` is an operator:

- Arithmetic reductions: `+ * - max min`
- Logical operator reductions: `& && | ||`

Reduction

The sum and maximum example using critical region can be rewritten with `reduction` clauses instead

```
#pragma omp parallel for reduction(+:sum) reduction(max:max)
for (int i = 0; i < n; ++i) {
    sum += a[i];
    max = MAX(max, a[i]);
}

printf("Sum after parallel region: %d.\n", sum);
printf("Max after parallel region: %d.\n", max);
```

Reduction

```
 $ gcc -fopenmp -o example omp_reduction.c
 $ OMP_NUM_THREADS=4 ./example
Sum after parallel region: 4950.
Max after parallel region: 99.
```

Atomic operation

An atomic operation is an operation that will always be executed without any other thread being able to read or change state that is read or changed during the operation.

```
#pragma omp atomic [atomic-clause]
  expression-statement
```

Atomic operation

```
#pragma omp atomic atomic-clause
  expression-statement
```

The value of `atomic-clause` can be one of the following: `read`, `write`, `update` and `capture`. If no `atomic-clause` is specified, the default value is `update`.

Atomic operation: Read and Write

The `read` clause allows for the atomic read of `x`.

```
#pragma omp atomic read
v = x;
```

The `write` clause allows for the atomic write of `x`. Here, `expr` is an expression with scalar type, i.e. the result of the expression is a scalar.

```
#pragma omp atomic write
x = expr;
```

Atomic operation: Update

The `update` clause allows for the atomic update of `x`.

```
#pragma omp atomic update
  expression-statement
```

Expression statement

```
    x++;          x--;          ++x;       --x;

x op= expr;  x = x op expr;  x = expr op x;
```

Atomic operation: Capture

The `capture` clause allows for atomic update of the location designated by `x` while also capturing the original or final value of the location designated by `x`.

```
#pragma omp atomic update
  expression-statement
```

Expression statement

```
v = x++;  v = x--;  v = ++x;  v = --x;

 v = x op= expr;   v = x = x op expr;

          v = x = expr op x;
```

Atomic operation: Capture

The `capture` clause allows for atomic update of the location designated by `x` while also capturing the original or final value of the location designated by `x`.

```
#pragma omp atomic update
  structured-block
```

Structured block (part. 1)

`{ v = x; x op= expr; }`	`{ x op= expr; v = x; }`
`{ v = x; x = x op expr; }`	`{ v = x; x = expr op x; }`
`{ x = x op expr; v = x; }`	`{ x = expr op x; v = x; }`

Atomic operation: Capture

The `capture` clause allows for atomic update of the location designated by `x` while also capturing the original or final value of the location designated by `x`.

```
#pragma omp atomic update
  structured-block
```

Structured block (part. 2)

```
{ v = x; x++; } { v = x; ++x; } { ++x; v = x; }
{ x++; v = x; } { v = x; x--; } { v = x; --x; }
{ --x; v = x; } { x--; v = x; }
```

Atomic example

The previous example of the summation of the elements of an array using a `critical` construct can be rewritten using an `atomic` update.

```
#pragma omp for
for (int i = 0; i < n; ++i) {
    local_sum += a[i];
}

#pragma omp atomic
  sum += local_sum;
```

Atomic example

```
for (i = 0; i < 10000; ++i) {
  index[i] = i % 1000;
  y[i] = 0.0;
}

for (i = 0; i < 1000; ++i)
  x[i] = 0.0;

#pragma omp parallel for
for (i = 0; i < 10000; ++i) {
  #pragma omp atomic update
  x[index[i]] += 1.0 * i;

  y[i] += 2.0 * i;
}
```

Atomic example

```
for (i = 0; i < 10000; ++i) {
  index[i] = i % 1000;
  y[i] = 0.0;
}

for (i = 0; i < 1000; ++i)
  x[i] = 0.0;

#pragma omp parallel for
for (i = 0; i < 10000; ++i) {
  #pragma omp atomic update
  x[index[i]] += 1.0 * i;

  y[i] += 2.0 * i;
}
```

The advantage of using `atomic` in this example is that it allows updates of two different elements of `x` in parallel. If a `critical` construct were used, all updates to elements of `x` would be executed serially

Atomic vs. Critical

Safely incrementing the value of `count` in parallel can be done either by using an `atomic` or a `critical` directive

```
#pragma omp atomic
  count++;
```

- An atomic operation has much lower overhead but the set of possible operations is restricted
- It can take advantage of hardware support for atomic operations

```
#pragma omp critical
  count++
```

- A critical section can surround any arbitrary block of code
- There is a significant overhead when a thread enters and exits the critical section

Atomic vs. Reduction

Don't use `atomic` operation this way:

```
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
  #pragma omp atomic
  sum += a[i];
}
```

It is better to use a `reduction` clause:

```
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; ++i)
  sum += a[i];
```

Performance Considerations

- Avoid or minimize the use of `barrier` and `critical` sections.
- Use the `nowait` clause where possible to eliminate unnecessary barriers
- Favour the use of `master` instead of `single`

Loop Scheduling

Loop Scheduling

Loop scheduling, specify how iterations of a loops are divided into contiguous non-empty subsets (chunks), and how these chunks are distributed to the threads. Changing the loop scheduling is possible using the `schedule` clause.

```
#pragma omp for schedule(schedule-kind, chunk-size)
  for-loop
```

Where the value of `schedule-kind` can be `static`, `dynamic`, `guided` or `runtime`. The default scheduling is `static`. The optional `chunk-size` may have different behavior depending on the scheduling.

Static Loop Scheduling

With `static` loop scheduling, iterations are divided into chunks and the chunks are assigned to the threads. Each chunk contains the same number of iterations, except for the chunk that contains the last iteration, which may have fewer iterations.

```
#pragma omp for schedule(static)
  for-loop
```

Dynamic Loop Scheduling

With `dynamic` loop scheduling, the iterations are distributed to threads in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.

```
#pragma omp for schedule(dynamic)
  for-loop
```

Guided Loop Scheduling

The `guided` loop scheduling is similar to the `dynamic` scheduling except that the size of each chunk is proportional to the number of unassigned iterations, decreasing to one.

```
#pragma omp for schedule(guided)
  for-loop
```

Why Using the Scheduling Clause?

- The default scheduling, `static` with a `chunk-size` equals to `niter/nthreads` is not ideal for all workload.
- It may be the case that iterations of high index represent more work. In that case, some of the threads will finish early and have nothing to do. We have a load imbalance.
- Changing the scheduling may help to balance the amount of work between the threads.

Example: Number of Prime Numbers

```
int prime, sum = 0;
#pragma omp parallel shared(n) private(prime)
{
  #pragma omp for reduction(+:sum)
  for (int i = 2; i <= n; i++) {
    prime = 1;

    for (int j = 2; j < i; j++) {
      if ( i % j == 0 ) {
        prime = 0;
        break;
      }
    }

    sum += prime;
  }
}
```

Example: Number of Prime Numbers

```
int prime, sum = 0;
#pragma omp parallel shared(n) private(prime)
{
  #pragma omp for reduction(+:sum)
  for (int i = 2; i <= n; i++) {
    prime = 1;

    for (int j = 2; j < i; j++) {
      if ( i % j == 0 ) {
        prime = 0;
        break;
      }
    }

    sum += prime;
  }
}
```

Trip count of this loop may be very low or very high depending if the number is prime or not

Example: Number of Prime Numbers

```
int prime, sum = 0;
#pragma omp parallel shared(n) private(prime)
{
  #pragma omp for reduction(+:sum)
  for (int i = 2; i <= n; i++) {
    prime = 1;

    for (int j = 2; j < i; j++) {
      if ( i % j == 0 ) {
        prime = 0;
        break; ←—— If the number is not a prime number, we have an early exit
      }
    }

    sum += prime;
  }
}
```

Example: Number of Prime Numbers

```
$ gcc -fopenmp -o example omp_schedule_prime.c
$ OMP_NUM_THREADS=4 ./example
```

N	Pi(N)	Default Time	Static Time	Dynamic Time	Guided Time
1024	172	0.000182	0.000120	0.000104	0.000121
2048	309	0.000561	0.000359	0.000425	0.000393
4096	564	0.001987	0.001309	0.001216	0.001239
8192	1028	0.007116	0.004474	0.004375	0.005114
16384	1900	0.029730	0.015594	0.015902	0.015161
32768	3512	0.099248	0.058475	0.056940	0.057160
65536	6542	0.358250	0.218291	0.244626	0.254815
131072	12251	1.416871	0.848736	0.788619	0.819390
262144	23000	5.207946	3.193940	3.062080	3.064527
524288	43390	20.565462	12.638959	12.086839	12.102800

Example: Triangular Loop

```
#pragma omp parallel shared(a, n)
{
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    a[i] = 0.0;

    for (int j = 0; j < i; ++j) {
      a[i] += cos( -3.1 * sin( 2.3 * cos ( (double) j ))) ;
    }
  }
}
```

Example: Triangular Loop

```
$ gcc -fopenmp -o example omp_schedule_triangular.c
$ OMP_NUM_THREADS=4 ./example
```

N	Default Time	Static Time	Dynamic Time	Guided Time
1024	0.025865	0.016811	0.018739	0.018241
2048	0.100062	0.070023	0.082587	0.091206
4096	0.383107	0.238520	0.232556	0.226914
8192	1.515341	0.905186	0.895541	0.880046
16384	6.064787	3.540685	3.526388	3.590453
32768	24.041088	15.465762	14.088137	14.539937
65536	97.495829	59.291353	59.403173	60.252156

Nesting, Orphaning and Tasking

Nested Parallelism

OpenMP parallel regions can be nested inside each other but it is disabled by default, meaning that

- If nested parallelism is disabled, then the new team created by a thread encountering a parallel construct inside a parallel region consists only of the encountering thread. This is the default.
- If nested parallelism is enabled, then the new team may consist of more than one thread.
- The maximum level of nested parallelism can be set by the `OMP_MAX_ACTIVE_LEVELS` environment variable.

Nested Parallelism

```
void report_num_threads(int level) {
  #pragma omp single
  printf("Level %d - number of threads: %d\n",
          level, omp_get_num_threads());
}

#pragma omp parallel num_threads(2)
{
  report_num_threads(1);
  #pragma omp parallel num_threads(2)
  {
    report_num_threads(2);
    #pragma omp parallel num_threads(2)
    {
      report_num_threads(3);
    }
  }
}
```

Nested Parallelism : Disabled

```
 $ gcc -fopenmp -o example omp_nested.c
 $ OMP_NUM_THREADS=4 ./example
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

Nested Parallelism: Enabled

```
$ gcc -fopenmp -o example example omp_nested.c
$ OMP_NUM_THREADS=4 OMP_MAX_ACTIVE_LEVELS=3 \
  ./example
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

Nested Parallelism the Solution?

Let's consider this piece of code: the amount of iteration in the first loop is tiny. Running this loop in parallel may be inefficient.

```
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {

  for (int j = 0; j < n; ++j) {
    a[i][j] = do_something();
  }
}
```

Nested Parallelism the Solution?

Let's consider this piece of code: the amount of iteration in the first loop is tiny. Running this loop in parallel may be inefficient.

```
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {

  for (int j = 0; j < n; ++j) {
    a[i][j] = do_something();
  }
}
```

If I run this program on more than three threads, some threads will be idle

Nested Parallelism the Solution?

Let's consider this piece of code: the amount of iteration in the first loop is tiny. Running this loop in parallel may be inefficient.

```
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
  #pragma omp parallel for
  for (int j = 0; j < n; ++j) {
    a[i][j] = do_something();
  }
}
```

Why not use nested parallelism?
Mmm... We rely on the user enabling nested parallelism and, if he/she does, controlling the number of threads is tedious, we will probably have a competition for the resources

Nested Parallelism the Solution?

Let's consider this piece of code: the amount of iteration in the first loop is tiny. Running this loop in parallel may be inefficient.

```
for (int i = 0; i < 3; ++i) {
  #pragma omp parallel for
  for (int j = 0; j < n; ++j) {
    a[i][j] = do_something();
  }
}
```

Not really the best solution, we may have more overhead by creating the team of threads for the inner loop

Loop collapsing

The best way is to collaspe the two loops to increase the run trip of the loop.

```
#pragma omp parallel for
for (int ij = 0; ij < 3*n; ++ij) {
  a[ij/n][j%n] = do_something();
}
```

Or better, use the `collapse` clause

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < 3; ++i) {
  for (int j = 0; j < n; ++j) {
    a[i][j] = do_something();
  }
}
```

Loop collapsing

The `collapse` clause, collapse the iterations of the n-associated loops to which the clause applies into one larger iteration space. This clause can only apply on tightly nested loops, meaning that there is no code between the loops.

```
#pragma omp for collapse(n)
  nested-for-loops
```

Tasking

Not all calculations are expressed as loops (data parallel), in the case where units of work are generated dynamically, as in recursive structures it is better to use task parallelism.

For task parallelism, OpenMP provides the `task` construct.

```
#pragma omp task
  structured-block
```

A task is an independent unit of work that is running or going to run.

Tasking

- When a thread encounters a task construct, a new task is generated and added to the task queue.
- The moment of execution of the task is up to the OpenMP runtime, which chooses an eligible task in the task queue.
- Execution can either be immediate or delayed.
- Completion of a task can be enforced through task synchronization (with the `taskwait` directive).

Tasking Example: Fibonacci Number

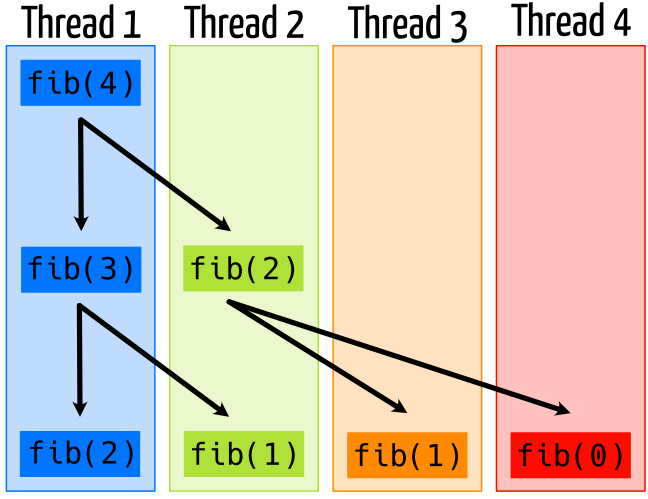
```
int fib(int n) {
  int i, j;

  if(n > 1) {
    #pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);

    #pragma omp taskwait

    return i+j;
  }

  return n;
}
```

Tasking Example: Fibonacci Number

```
int fib(int n) {
  int i, j;

  if(n > 1) {
    #pragma omp task shared(i) firstprivate(n)
    i = fib(n-1);

    #pragma omp task shared(j) firstprivate(n)
    j = fib(n-2);

    #pragma omp taskwait

    return i+j;
  }

  return n;
}
```

We need to wait for the two tasks to complete in order to compute the result

Fibonacci: Execution



Orphaning

- Directives are active in the dynamic scope of a parallel region, not just its lexical scope. This allows for orphaned directives.
- Orphaning is a situation when directives related to a parallel region are outside the lexical extent of the parallel region.
- Typical situation is calling a function containing a worksharing directive from a parallel region.

Orphaning Example

```
void ax(int n, double alpha, double* x) {
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    x[i] = alpha * x[i];
  }
}

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

  #pragma omp parallel shared(x, n)
  {
    ax(n, 3.0, x);
  }

  return 0;
}
```

Orphaning Example

```
void ax(int n, double alpha, double* x) {
  #pragma omp for
  for (int i = 0; i < n; ++i) {
    x[i] = alpha * x[i];
  }
}

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

  #pragma omp parallel shared(x, n)
  {
    ax(n, 3.0, x);
  }

  return 0;
}
```

Orphaned `for` directive that will bind to the calling parallel region

Call to a function that contain an orphaned directive

Orphaning

- If a function with an orphaned directive is called outside of a parallel region, then this function will only be executed by the master thread.
- In an orphaned directive, variables in the argument list inherit their data scope attribute from the calling routine.

False Sharing

False Sharing in Action

```
double local_sum[omp_get_max_threads()];
double sum = 0.0;

#pragma omp parallel shared(sum)
{
  int tid = omp_get_thread_num();
  local_sum[tid] = 0.0;

  #pragma omp for
  for (int i = 0; i < n; ++i)
      local_sum[tid] += 0.5 * x[i] + y[i];

  #pragma omp atomic
  sum += local_sum[tid];
}
```

False Sharing in Action

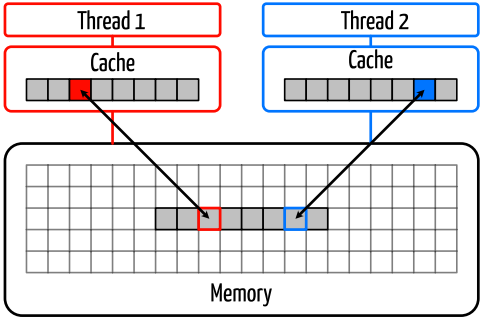
Let's measure the time spend in the parallel region (using the `omp_get_wtime()` function).

Threads	Time (s)
1	0.535418
2	0.421140
4	0.554419
8	0.597622

- The speedup from 1 thread to 2 theads is bad
- When going to 4 and 8 threads the time spend in the parallel region is worst than with 1 thread

False Sharing

False sharing is when threads impact the performance of each other while modifying independent variables sharing the same cache line



- If one core writes, the cache line holding the memory line is invalidated on other cores.
- Even though another core is not using that data, the second core will need to reload the line before it can access its own data again.

False Sharing: Solution

Solution: introduce a padding.

```
double local_sum[LINESIZE*omp_get_max_threads()];
double sum = 0.0;

#pragma omp parallel shared(sum)
{
  int tid = omp_get_thread_num();
  local_sum[LINESIZE*tid] = 0.0;

  #pragma omp for
  for (int i = 0; i < n; ++i)
      local_sum[LINESIZE*tid] += 0.5 * x[i] + y[i];

  #pragma omp atomic
  sum += local_sum[LINESIZE*tid];
}
```

False Sharing: Solution

Timing for different paddings on a CPU with a cache line size of 64 bytes.

Threads	Time (s) padding = 4	Time (s) padding = 8	Time (s) padding = 16
1	0.535418	0.535418	0.535418
2	0.601417	0.270089	0.270843
4	0.441149	0.152651	0.149363

False sharing

- When threads access global or dynamically allocated shared data structures there is a potential sources of false sharing
- False sharing may be difficult to spot. For example, when theads access completely different global variables that happen to be relatively close together in memory.
- Use thread-local copies of data when possible. The thread-local copy can be read and modified frequently and only when complete, be copied back to the global data structure

That’s all folks!