Parallel Algorithms and Programming

Introduction to OpenMP

Thomas Ropars

thomas.ropars@univ-grenoble-alpes.fr

2021

Agenda

Agenda

References

The content of these lectures is inspired by:

- The lecture notes of F. Desprez.
- The presentation of F. Broquedis:
  - `http://smai.emath.fr/cemracs/cemracs16/images/FBroquedis.pdf`
  - A large number of slides are directly borrowed from this presentation
- OpenMP tutorial at LLNL: `https://computing.llnl.gov/tutorials/openMP/`

What is OpenMP?

- A de-facto standard API to write shared memory parallel applications in C, C++ and Fortran
  - Supported by the main compilers
- Consists of:
  - compiler directives
  - runtime routines
  - environment variables
- Specification maintained by the *OpenMP Architecture Review Board* (`http://www.openmp.org`)

Advantages of OpenMP

- A mature standard
  - *Speeding-up your applications since 1998*
- Portable
  - Supported by many compilers, ported on many architectures
- Allows **incremental parallelization**
- Imposes low to no overhead on the sequential execution of the program
  - Just tell your compiler to ignore the OpenMP pragmas and you get back to your sequential program
- Supported by a wide and active community
  - The specifications have been moving fast in the recent years to support :
    - new kinds of parallelism (tasking)
    - new kinds of architectures (accelerators)

Parallelizing a program with OpenMP

The main steps

- The programmer inserts *pragmas* to tell the compiler how to parallelize the code
- The code is compiled with the appropriate flags
  - `-fopenmp` for gcc
  - The compiler makes the required code transformations and library calls
- The generated executable can be run as it.
  - The execution can be customized by defining environment variables
    - Number of threads
    - Scheduling policy
  - The runtime library is in charge of scheduling the work to be done and synchronizing the threads

OpenMP vs pthreads

Pthreads

- Low-level API
- The programmer has the flexibility to do whatever he wants

OpenMP

- Often we just want to:
  - Parallelize one loop
  - Hand-off a piece of computation to another thread
- OpenMP allows doing this easily
  - One can sometimes get a huge speed-up by modifying a single line in the source code
  - **The use of pragmas is not always easy**
    - It can be challenging to figure out what the compiler is going to do exactly

Why we need OpenMP

Case of loop parallelization

The compiler may not be able to do the parallelization fully automatically in the way you would like to see it:

- It cannot find parallelism
  - The data dependence analysis is not able to determine whether it is safe to parallelize or not
- The granularity is not appropriate
  - The compiler lacks information to parallelize at the highest possible level

This is where explicit parallelization through OpenMP directives comes into the picture

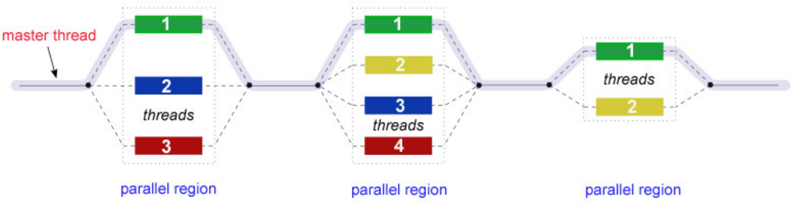
Agenda

Introduction

OpenMP basics: parallelizing for loops

OpenMP tasks

A fork-join execution model



- Initially a single master thread
- A team of threads is created to execute a parallel region
- The number of threads and the number of parallel regions can be arbitrary large

A fork-join execution model

```
int main(void)
{
    some_statements();

    #pragma omp parallel
    {
        printf("Hello,␣world!\n");
    }

    other_statements();

    #pragma omp parallel
    {
        printf("Bye\n");
    }

    return EXIT_SUCCESS;
}
```

- Entering a parallel region will create some threads (*fork*)
  - If no further directives, the code inside a parallel regions is executed by all threads
- Leaving a parallel region will terminate them (*join*)
- Any statement executed outside parallel regions are executed sequentially

The OpenMP memory model

- All threads have access to the same, globally shared, memory
- Data can be shared or private:
  - Shared data is accessible by all threads
  - Private data can only be accessed by the thread that owns it
  - The programmer is responsible for providing the corresponding data-sharing attributes
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit

Data-sharing Attributes

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (p282)

The visibility of each variable that appears inside a parallel region should be set using **data-sharing attributes** :

- **shared**: The data can be read and written by any thread of the team. All changes are visible to all threads.
- **private**: Each thread is working on its own version of the data that cannot be accessed by other threads of the team. The variable is uninitialized at the beginning of the parallel region.

- **firstprivate**: The variable is private. It is initialized using the value it had before entering the parallel region.
- **lastprivate**: The variable is private. At the end of the parallel region, the variable has the same value as in the last iteration of the sequential code.

Data-sharing Attributes

https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf (p270)

If the visibility of a variable is not explicitly set, the compiler infers a default visibility for variables:

- The visibility depends on the characteristics of variables, when it is declared, how it is allocated, etc. Check the documentation.
- Most variables are **shared** by default.
- Iterator variables of parallel loops are **private**.

Putting Threads to Work: A Parallel Loop

```
void simple_loop(int N,
                 float *a,
                 float *b)
{
    int i;
    // i, N, a and b are shared by default
    #pragma omp parallel firstprivate(N)
    {
        // i is private by default
        #pragma omp for
        for (i = 1; i <= N; i++) {
            b[i] = (a[i] + a[i-1]) / 2.0;
        }
    }
}
```

- **omp for** : distribute the iterations of a loop over the threads of the parallel region.
- Here, assigns `N/P` iterations to each thread, `P` being the number of threads of the parallel region.
- **omp for** comes with an implicit **barrier synchronization** at the end of the loop. Can be removed with the **nowait** keyword.

The reduction clause

The reduction clause allows defining a combiner operation to be applied on a variable:

- A private variable is created for each thread of the parallel region
- At the end of the parallel region, the value of the variable for each thread is combined to compute a single value
- Possible combiners include: '+', '-', '*', 'and', 'or', 'max', etc.

```c
int count_zeros(int N, float *a)
{
    int i, count;
    #pragma omp parallel for reduction(+:count)
    for (i = 0; i < N; i++) {
        if(a[i] == 0){
            count++;
        }
    }
    return count;
}
```

OpenMP Loop Schedulers: Definitions

The **schedule** clause of the **for** construct specifies the way loop iterations are assigned to threads. The loop scheduler can be set to one of the following :

- **schedule(static, chunk_size)**: assign fixed chunks of iterations in a round robin fashion[1].
- **schedule(dynamic, chunk_size)**: fixed chunks of iterations are dynamically assigned to threads at runtime, depending on the threads availability.
- **schedule(guided, chunk_size)**: like dynamic, but with a chunk size that decreases over time (until reaching `chunk_size`)
- **runtime**: the loop scheduler is chosen at runtime thanks to the `OMP_SCHEDULE` environment variable.

[1]When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case.

OpenMP Loop Schedulers: Chunks

The `chunk_size` attribute determines the granularity of iterations chunks the loops schedulers are working with.

legend: 🟩 thread0 🟨 thread1 🟦 thread2 🟥 thread3

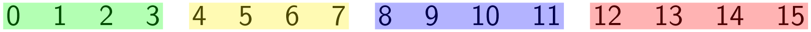


Figure: `static` scheduler, 16 iterations, 4 threads, **default** `chunk_size`

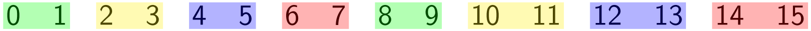
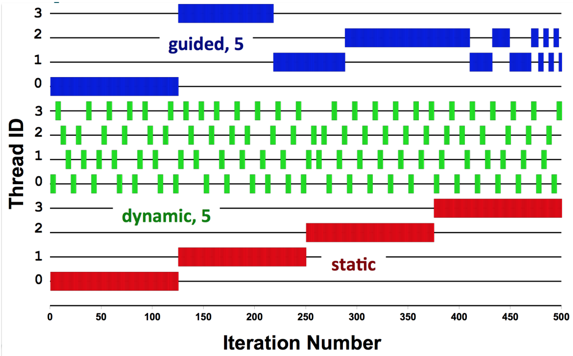


Figure: `static` scheduler, 16 iterations, 4 threads, `chunk_size`**=2**

OpenMP Loop Schedulers

Example on 500 iterations, 4 threads[1]



Warning: The figure represents the assignment of iterations to threads, not the execution over time.

[1]For a chunk_size of k with *guided* scheduling, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to k

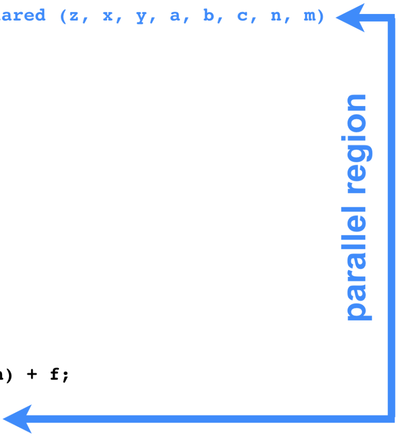
An Example to Illustrate OpenMP Capabilities

```
f = 1.0


for (i = 0; i < N; i++)
  z[i] = x[i] + y[i];


for (i = 0; i < M; i++)
  a[i] = b[i] + c[i];

...


scale = sum (a, 0, m) + sum (z, 0, n) + f;
...
```

Our job for today: parallelize this code using OpenMP

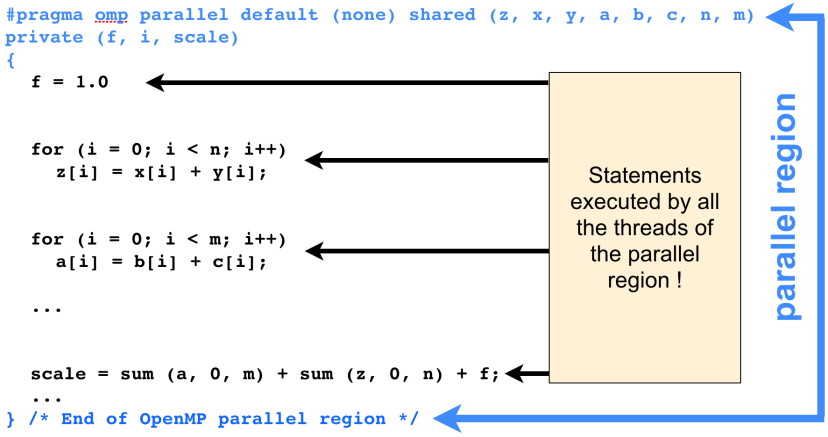
An Example to Illustrate OpenMP Capabilities

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0


  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];


  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...


  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```



First create the parallel region and define the data-sharing attributes

An Example to Illustrate OpenMP Capabilities

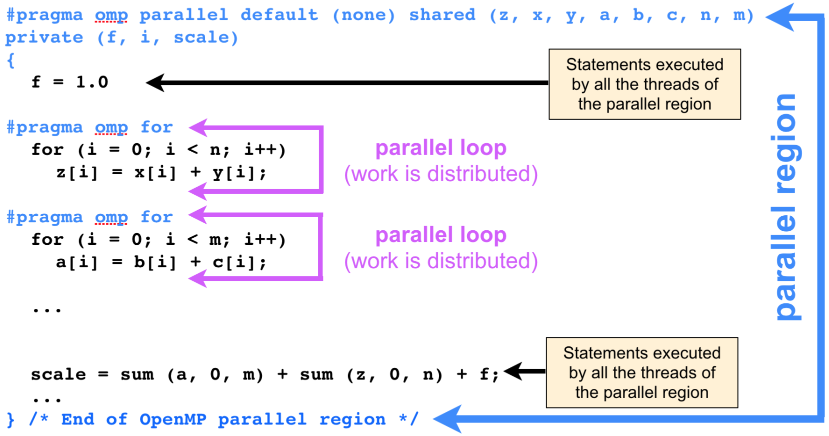


```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0


  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];


  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...


  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region !
parallel region

At this point, all the threads execute the whole program (you won't get any speed-up from this!)

An Example to Illustrate OpenMP Capabilities



```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0

#pragma omp for
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...


  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region
parallel loop
(work is distributed)
parallel loop
(work is distributed)
parallel region
Statements executed by all the threads of the parallel region

Now distribute the loop iterations over the threads using omp for.

Optimization #1: Remove Unnecessary Synchronizations

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
  f = 1.0

#pragma omp for nowait
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for nowait
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...

#pragma omp barrier
  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

parallel region

There are no dependencies between the two parallel loops, we remove the implicit barrier between the two.

Optimization #2: Don't Go Parallel if the Workload is Small

```
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold)
{
  f = 1.0

#pragma omp for nowait
  for (i = 0; i < n; i++)
    z[i] = x[i] + y[i];

#pragma omp for nowait
  for (i = 0; i < m; i++)
    a[i] = b[i] + c[i];

  ...

#pragma omp barrier
  scale = sum (a, 0, m) + sum (z, 0, n) + f;
  ...
} /* End of OpenMP parallel region */
```

We don't want to pay the price of thread management if the workload is too small to be computed in parallel.

Additional comments

Note that:

- The default(none) clause requires that each variable that is referenced in the construct must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause.
    - If left unspecified, a variable declared outside of a parallel region is shared by default.

- When no schedule kind is defined, the default scheduling policy is applied.
    - The default policy is implementation-dependent.

Agenda

Extending the Scope of OpenMP with Task Parallelism

omp for has made OpenMP popular and remains for most users its central feature. But what if my application was not written in a loop-based fashion?

```
int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        i = fib(n - 1);
        j = fib(n - 2);
        return i + j;
    }
}
```

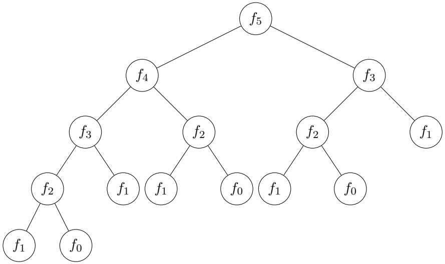
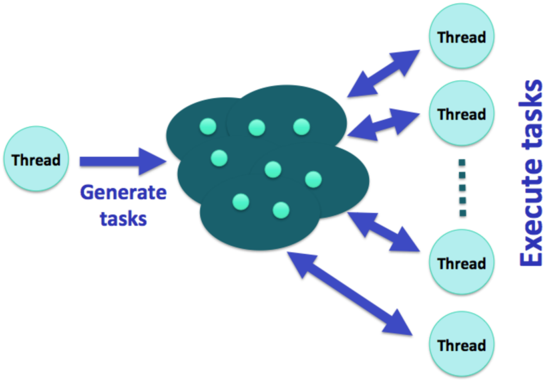


Figure: Call graph of fib(5)

Tasking in OpenMP

Basic concept

The OpenMP tasking concept : tasks generated by one OpenMP thread can be executed **by any of the threads** of the parallel region.



Basic Concept

- The application programmer specifies regions of code to be executed in a task with the **#pragma omp task** construct
- All tasks can be executed *independently*
- When any thread encounters a task construct, a task is generated
- Tasks are executed **asynchronously** by any thread of the parallel region
- Completion of the tasks can be guaranteed using the **taskwait** synchronization construct

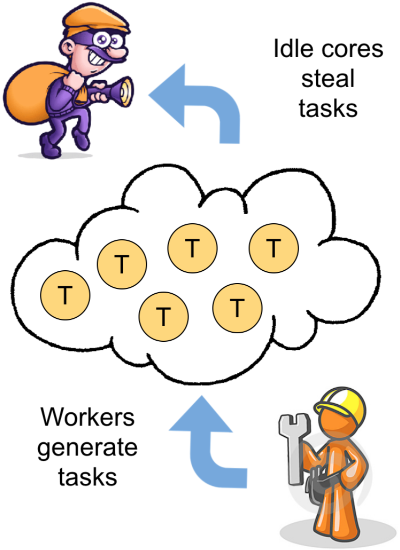
```
int main(void) {
    ...
    #pragma omp parallel
    {
        #pragma omp single
        res = fib(50);
    }
    ...
}

int fib(int n) {
    int i, j;
    if (n < 2) {
        return n;
    } else {
        #pragma omp task
        i = fib(n - 1);
        #pragma omp task
        j = fib(n - 2);
        #pragma omp taskwait
        return i + j;
    }
}
```

Tasking in OpenMP: Execution Model

The Work-Stealing execution model

- Each thread has its own *task queue*
- Entering an `omp task` construct pushes a task to the thread's local queue
- When a thread's local queue is empty, it steals tasks from other queues

Tasks are well suited to applications with irregular workload.



Your Very First OpenMP Tasking Experience (1/5)

```
int main(void)
{
    printf("A␣");
    printf("race␣");
    printf("car␣");

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `"A race car"` or `"A car race"` using tasks.

Program output:

```
$ OMP_NUM_THREADS=2 ./task-1
$ A race car
```

Your Very First OpenMP Tasking Experience (2/5)

```
int main(void)
{
    #pragma omp parallel
    {
        printf("A␣");
        printf("race␣");
        printf("car␣");
    }

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `"A race car"` or `"A car race"` using tasks.
    1. **Create the threads that will execute the tasks**

Program output:

```
$ OMP_NUM_THREADS=2 ./task-2
$ A race A race car car
```

Your Very First OpenMP Tasking Experience (3/5)

```c
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A␣");
            #pragma omp task
            printf("race␣");
            #pragma omp task
            printf("car␣");
        }
    }

    printf("\n");
    return 0;
}
```

- We want to use OpenMP to make this program print either `"A race car"` or `"A car race"` using tasks.

  1. Create the threads that will execute the tasks
  2. **Create the tasks and make one of the threads generate them**

Program output:

```
$ OMP_NUM_THREADS=2 ./task-3
$ A race car
$ OMP_NUM_THREADS=2 ./task-3
$ A car race
```

Your Very First OpenMP Tasking Experience (4/5)

```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A␣");
            #pragma omp task
            printf("race␣");
            #pragma omp task
            printf("car␣");

            printf("is␣fun␣");
            printf("to␣watch␣");
        }
    }

    printf("\n");
    return 0;
}
```

- We would like to print `"is fun to watch"` at the end of the output string.

Your Very First OpenMP Tasking Experience (4/5)

```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A␣");
            #pragma omp task
            printf("race␣");
            #pragma omp task
            printf("car␣");

            printf("is␣fun␣");
            printf("to␣watch␣");
        }
    }

    printf("\n");
    return 0;
}
```

- We would like to print `"is fun to watch"` at the end of the output string.
- **Beware of the asynchronous execution of tasks.**

Program output:

```
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch race car
$ OMP_NUM_THREADS=2 ./task-4
$ A is fun to watch car race
```

Your Very First OpenMP Tasking Experience (5/5)

```
int main(void)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            printf("A␣");
            #pragma omp task
            printf("race␣");
            #pragma omp task
            printf("car␣");
            #pragma omp taskwait
            printf("is␣fun␣");
            printf("to␣watch␣");
        }
    }

    printf("\n");
    return 0;
}
```

- We would like to print `"is fun to watch"` at the end of the output string.
- **We explicitly wait for the completion of the tasks with taskwait**

Program output:

```
$ OMP_NUM_THREADS=2 ./task-5
$ A race car is fun to watch
$ OMP_NUM_THREADS=2 ./task-5
$ A car race is fun to watch
```

What About Tasks with Dependencies on Other Tasks?

```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        write_data(&x); // Task A
        #pragma omp task
        write_data(&y); // Task B

        #pragma omp taskwait

        #pragma omp task
        print_results(x); // Task C
        #pragma omp task
        print_results(y); // Task D
    }
}
```

- Task A writes data that will be processed by task C. Same for task B and task D.
- The **taskwait** construct makes sure task C won't execute before task A and D before B.

What About Tasks with Dependencies on Other Tasks?

```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        write_data(&x); // Task A
        #pragma omp task
        write_data(&y); // Task B

        #pragma omp taskwait

        #pragma omp task
        print_results(x); // Task C
        #pragma omp task
        print_results(y); // Task D
    }
}
```

- Task A writes data that will be processed by task C. Same for task B and task D.
- The **taskwait** construct makes sure task C won't execute before task A and D before B.
- Side effect: **unnecessary dependency** between B and C.
  - C won't execute until B is over

OpenMP Tasks Dependencies

The **depend** clause provide information on the way a task will access data, and thus, defines dependencies between tasks of a parallel region.

Access mode

The depend clause is followed by an access mode that can be in, out or inout.

OpenMP Tasks Dependencies

Examples

- depend(in: x, y, z): the task will read variables x, y and z
- depend(out: res): the task will write variable res, any previous value of res will be ignored and overwritten
- depend(inout: k, buffer[0:n]): the task will both read and write variable `k` and the content of n elements of `buffer` starting from index 0

The OpenMP runtime system dynamically decides whether a task is ready for execution or not considering its dependencies.

OpenMP Tasks Dependencies : Some Trivial Example

```
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: x)
        write_data(&x); // Task A
        #pragma omp task depend(out: y)
        write_data(&y); // Task B

        #pragma omp task depend(in: x)
        print_results(x); // Task C
        #pragma omp task depend(in: y)
        print_results(y); // Task D
    }
}
```

- Task C can be executed before task B, as long as the execution of task A is over.
    - More potential parallelism

Other OpenMP features

- Support for accelerators
  - Ability to instruct the compiler and runtime to offload a block of code to the device (for instance, a GPU)
- SIMD transformations