HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 6

Fabian Wermelinger
Harvard University

CS205

Thursday, February 9th 2023

LAST TIME

- Race conditions in depth
- Sequential consistency
- The need for (relaxed) memory consistency models in shared memory programming
- Critical sections and mutual exclusion

TODAY

Main topic: ***Introduction to OpenMP***

Disclaimer: all references to the OpenMP specification are for version 5.1

Details:

- What is OpenMP and why we prefer it
- Discussion of reading assignment (OpenMP specification): *terminology*, *execution model* and *memory model*
- Fork/join parallel regions in OpenMP
- Work sharing constructs
- Loop scheduling (load balancing control)
- Nested loops and loop collapsing
- Reduction operator

OpenMP IN A NUTSHELL

openmp.org

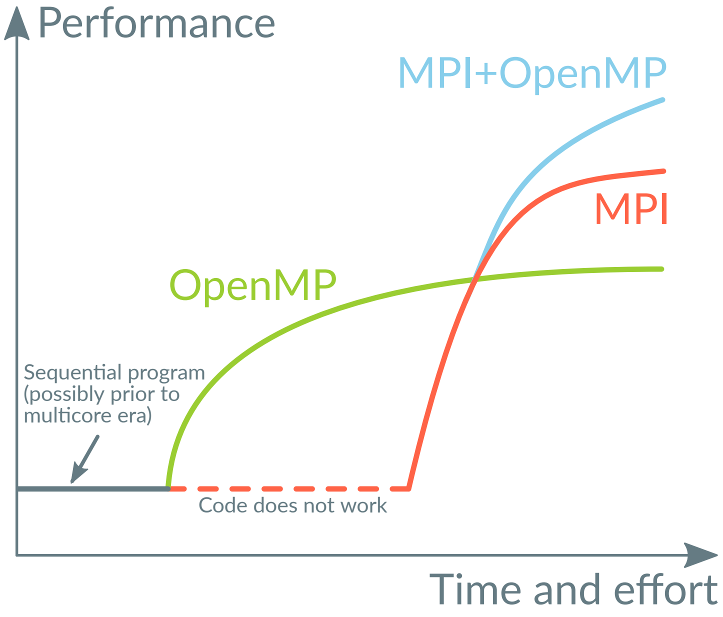
***OpenMP*:** Open Multi-Processing

- An **A**pplication **P**rogramming **I**nterface (API) for explicit direct multi-threaded parallelism on shared memory architectures
- Consists of **3** components:
  1. *Compiler directives* (not present in POSIX threads or C++11)
  2. Runtime library calls
  3. Environment variables (not present in POSIX threads or C++11)
- A *standard* widely used in industry and academic codes

- ***Is not*** required to check your code for data races, deadlocks or whether your code conforms with the standard
- Is not designed for distributed memory systems (MPI is)
- Is not designed for parallel I/O (it could be hacked with synchronization primitives)

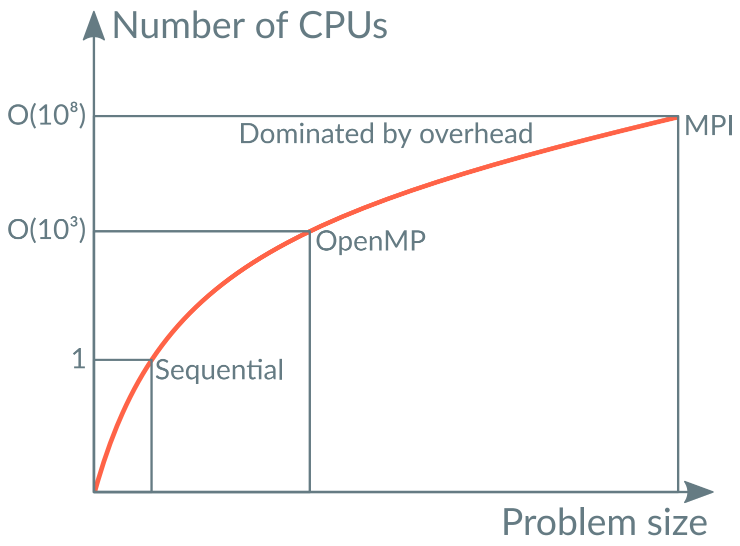
OpenMP IN A NUTSHELL

openmp.org



- OpenMP support can be added very easily to *existing* sequential code via compiler directives. (Not the case for POSIX or C++11 threads.)
- The compiler simply ignores these directives if it does not recognize them. To tell the compiler about these pragmas, you must add the `-fopenmp` flag at compile time (for GCC).
- Your code is less affected by a development stall due to the parallelization intervention. This is not the case for MPI, where the intervention will cause your code to stop running for some time (development, it might not compile, other bugs, etc.)
- Other programming models are also based on compiler directives, e.g., OpenACC.

OpenMP IN A NUTSHELL



- OpenMP (shared memory) applications target nodes that have usually around 16 to 128 cores but may as well be larger. Upper bound may be order 1000 cores, although such systems are not the norm.
- Because the amount of memory on a node is limited, the target problem size is limited as well.
- For larger problems distributed memory is necessary using MPI for example.
- Such large systems are dominated by communication overhead. A method to alleviate this issue is to combine MPI and OpenMP to a hybrid code.

WHY OpenMP ?

- Easy to use with simple *compiler directives* and allows to parallelize *existing* sequential programs with *minimum* effort.
- Parallelization can be done *incrementally*. Work on one compute kernel, benchmark the effort and then proceed with other parallelization.
- You can implement *coarse-grained* parallelism (e.g. via tasks) and *fine-grained* parallelism (e.g. via locks and atomics).
- Widely used in industry and academia with large community support.
- Portability:
  - C, C++ and Fortran
  - Supports Unix/Linux as well as Windows

N.B.: Compute-Transfer overlap is the ratio

$$\frac{C}{T} = \frac{\text{Time spent doing computation}}{\text{Time spent in communication}}$$

- $\frac{C}{T} \gg 1$: Coarse-grained parallelism
- $\frac{C}{T} \ll 1$: Fine-grained parallelism

C++11 THREADS AND PTHREADS

- Alternatives to OpenMP include C++ threads (since 2011 standard), `pthreads` or Intel TBB for example.
- C++11 threads or `pthreads` have some additional boilerplate code associated with them compared to OpenMP → *we like to avoid this.*
- To use C++11 threads you would need to include the `<thread>` header and compile with at least the `-std=c++11` standard specified (most recent compilers do this anyway) as well as the `-pthreads` flag (C++ threads depends on `pthreads`):

```
$ g++ -std=c++11 -pthread <C++ threads code>
```

- For code that uses `pthreads` you would need to include the `<pthread.h>` and add the `-pthread` flag to the compilation options:

```
$ g++ -pthread <pthreads code>
```

C++11 THREADS AND PTHREADS

C++11 threads example implementation

```cpp
#include <thread>
#define NTHREADS 8

void do_work(const size_t tid)
{
    // more local data
    // do work
}

int main(void)
{
    std::thread threads[NTHREADS];

    // spawn threads
    for (size_t t = 0; t < NTHREADS; ++t)
        threads[i] = std::thread(do_work, t);
    // join threads
    for (size_t t = 0; t < NTHREADS; ++t)
        threads[i].join();

    return 0;
}
```

pthreads example implementation:

```c
#include <pthread.h>
#define NTHREADS 8

void *do_work(void *t)
{
    /* thread ID */
    long tid = (long)t;
    /* more local data */
    /* do work */
    return NULL
}

int main(void)
{
    pthread_t threads[NTHREADS];
    void *status;
    /* spawn threads */
    for (long t = 0; t < NTHREADS; ++t)
        pthread_create(&threads[t], NULL, do_work,
            (void *)t);
    /* join threads */
    for (int t = 0; t < NTHREADS; ++t)
        pthread_join(threads[t], &status);
    return 0;
}
```

OpenMP HISTORY

openmp.org

Date	Version	Features Added
Oct 1997	Fortran 1.0	
Oct 1998	C/C++ 1.0	
Nov 1999	Fortran 1.1	
Nov 2000	Fortran 2.0	
Mar 2002	C/C++ 2.0	
May 2005	OpenMP 2.5	
May 2008	OpenMP 3.0	• Concept of *tasking* added • New features for *worksharing*
Jul 2011	OpenMP 3.1	• `final` and `mergeable` clauses for `task` construct • `taskyield` construct to allow user-defined task switching points • `min`/`max` reductions in C/C++ • `OMP_PROC_BIND` added
Jul 2013	OpenMP 4.0	• Thread affinity and `OMP_PLACES` • SIMD support added • Tasking extensions • Support for accelerators (e.g. GPUs)
Nov 2015	OpenMP 4.5	• Support for Fortran 2003 • `firstprivate`
Nov 2018	OpenMP 5.0	• Extensions to memory model • Initial support for C11, C++11, C++14, C++17 and Fortran 2008
Nov 2020	OpenMP 5.1	• Full support for C11, C++11, C++14, C++17, C++20 and Fortran 2008 • Initial support for Fortran 2018
Nov 2021	OpenMP 5.2	• Reorganisation and changes made to improve the specification (better consistency and wording)

OpenMP DIRECTIVES BASICS

openmp.org

You can do a large amount of the parallelization using OpenMP compiler directives only! The main purpose of those directives is to fork/join parallel regions and provide synchronization primitives.

C/C++:

```
#pragma omp construct [clause [clause]...]
_Pragma("omp construct [clause [clause]...]")
```

Fortran:

```
!$omp construct [clause [clause]...]
c$omp construct [clause [clause]...]
*$omp construct [clause [clause]...]
```

- Since these are all either compiler directives or comments in the case of Fortran, there are ***no changes*** made on the code itself.
- Most of these directives will execute code, in that case we call them a *construct*, e.g., `#pragma omp parallel` is a construct that executes a parallel region. See Section 2.1 in the OpenMP specification v5.1.
- Clauses are part of a directive and are *optional*. An example clause would be to specify how many threads you want to use for a parallel region: `#pragma omp parallel num_threads(2)`

OpenMP DIRECTIVES BASICS (DETAILS)

```
#pragma omp directive-name [clause [[,]clause] ...] new-line
    structured block
```

- The optional *[,]* is an artifact from the old-days. It is usually not needed but some exotic compilers might still require it.
- What follows the directive is a *structured block*. Structured blocks are an executable statement (possibly compound between "{}") and define a *single point of entry* and a *single point of exit*.

Structured block:

```
#pragma omp parallel
{ // new scope: point of entry
    int a = 0;  // automatic variable
    do_work(a); // do something in parallel
} // point of exit
```

There is a *single* point of entry at { and point of exit at }. It could also be a single statement without the "{}".

Not a structured block:

```
#pragma omp parallel
{ // new scope: point of entry
    int a = 0;         // automatic variable
    do_work(a);        // do something in parallel
    if (a) goto out; // then go crazy!
} // point of exit
out: cout << "Skipped point of exit!\n";
```

The `goto` statement *jumps* out of the block, there will be *no single point of exit!*

OpenMP EXAMPLE

openmp.org

Parallelizing existing code with OpenMP is really easy:

Sequential code:

```
int main(void)
{
    int results[100];

    for (int i = 0; i < 100; ++i) {
        do_work(results[i]);
    }

    return 0;
}
```

```
$ g++ -o main main.cpp
```

Parallel code:

```
int main(void)
{
    int results[100];

#pragma omp parallel for
    for (int i = 0; i < 100; ++i) {
        do_work(results[i]);
    }

    return 0;
}
```

```
$ g++ -fopenmp -o main main.cpp
```

- This is a *very* simple example of *worksharing* (data-level parallelism). It seems trivial to just add this one `#pragma` and then call it a day...
- For this simple case yes (assuming the function `do_work()` is *thread-safe*), ***but*** keep in mind as soon as you add the `#pragma omp parallel` line, you have signed the contract for *relaxed memory consistency* and responsibility for correct parallel code is now yours!

COMPARED TO C++11 THREADS

The distinction between *sequential* and *parallel* regions is much cleaner with OpenMP

OpenMP

```
#include <omp.h>
#define NTHREADS 8

void do_work(const int tid)
{
    // more local data
    // do work
}

int main(void)
{
#pragma omp parallel num_threads(NTHREADS)
    {
        // need omp.h header for next line
        const int t = omp_get_thread_num();
        do_work(t);
    } // join threads (implicit barrier)

    return 0;
}
```

C++11 threads

```
#include <thread>
#define NTHREADS 8

void do_work(const int tid)
{
    // more local data
    // do work
}

int main(void)
{
    std::thread threads[NTHREADS];

    // spawn threads
    for (int t = 0; t < NTHREADS; ++t)
        threads[i] = std::thread(do_work, t);
    // join threads
    for (int t = 0; t < NTHREADS; ++t)
        threads[i].join();

    return 0;
}
```

COMPILING OpenMP CODE

- To compile OpenMP code you must tell the compiler to evaluate the `#pragma omp` pragmas. You do this by simply adding the `-fopenmp` flag to the compiler invocation

```
$ g++ -fopenmp [-o main] main.cpp  # -o main creates an executable named main, otherwise it is a.out
$ gcc -fopenmp [-o main] main.c    # same story for C
```

- When you pass `-fopenmp` a macro named _OPENMP will be defined as well. This macro contains a string with the format YYYYMM which corresponds to the OpenMP release you are using. The macro is useful for *conditional compilation:*

```
#include <iostream>

int main(void)
{
#ifdef _OPENMP
    std::cout << "Compiled with -fopenmp "
              << "(OpenMP Standard: "
              << _OPENMP << ")\n";
#else
    std::cout << "Compiled without -fopenmp\n";
#endif /* _OPENMP */
    return 0;
}
```

Output:

```
$ make
g++ -O1 -fopenmp -o with_omp omp_conditional.cpp
g++ -O1 -o without_omp omp_conditional.cpp
$ ./with_omp
Compiled with -fopenmp (OpenMP Standard: 201511)
$ ./without_omp
Compiled without -fopenmp
```

(See the lecture code in `omp_conditional` to reproduce)

OpenMP READING ASSIGNMENT

The reading assignment for this class was about the following three OpenMP topics:

1. ***Terminology:*** there are many terms in the glossary, you do not need to know all by heart. We will summarize the most important next.
2. ***Execution model:*** we already touched this → fork/join model using OpenMP *directives*.
3. ***Memory model:*** you should have been able to understand this part based on what we have discussed last time.

- The document you were reading is the OpenMP *specification* (the ground truth).
- ***Recall:*** OpenMP is a standard, different compiler vendors implement this standard following the specification. There can be *implementation dependent* behavior which is mentioned in the specification when this is to be expected.
- Everything you need to know about OpenMP is written in the specification. It is your main reference but can be a bit cumbersome to comprehend at times (specifications are similar in style to law documents).
- You can find an HTML version at this link: https://www.openmp.org/spec-html/5.1/openmp.html

OpenMP READING ASSIGNMENT

openmp.org

Because OpenMP supports C, C++ and Fortran, details for each of the three languages are *interleaved* in the specification with text that applies generally. It allows you to skip certain details you are not interested in without loosing track.

2 Directives

Generally applicable

This chapter describes the syntax and behavior of OpenMP directives.

C only

C

OpenMP directives are specified with the **`#pragma`** mechanism provided by the C standard.

C

C++ only

C++

OpenMP directives are specified with attribute specifiers or the **`#pragma`** mechanism provided by the C++ standard.

C++

Fortran only

Fortran

OpenMP directives are specified with stylized comments that are identified by unique sentinels. Also, a stylized comment form is available for conditional compilation.

If a directive appears in the declarative part of a module then the behavior is as if that directive appears after any references to that module.

Fortran

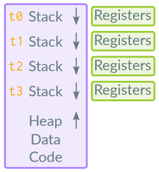
Generally applicable

Compilers can therefore ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all OpenMP directives and OpenMP conditional compilation mechanisms is enabled. In the remainder of this document, the phrase *OpenMP compilation* is used to mean a compilation with these OpenMP features enabled.

TERMINOLOGY (READING)

thread

An execution entity with a stack and associated *thread-private memory*.



thread number

A number that the OpenMP implementation assigns to a thread. `0` always the *primary* thread within a team of threads. The other child threads increment consecutively.

program order

An ordering of operations performed by *the same thread* as determined execution sequence of operators in the source code.

structured block

An executable statement, possibly compound, with a *single entry at the top* and a *single exit at the bottom* (i.e. a scope defined by "`{}`"), or an OpenMP construct.

directive

A C/C++ mechanism to specify OpenMP program behavior (e.g. compiler pragma or the `_Pragma()` macro).

conforming program

An OpenMP program that follows *all rules and restrictions* of the OpenMP specification.

barrier

A point in the execution of a program encountered by a *team of threads* beyond which no thread in the team may execute until *all* threads in the team have reached the barrier (i.e., a *synchronization point*).

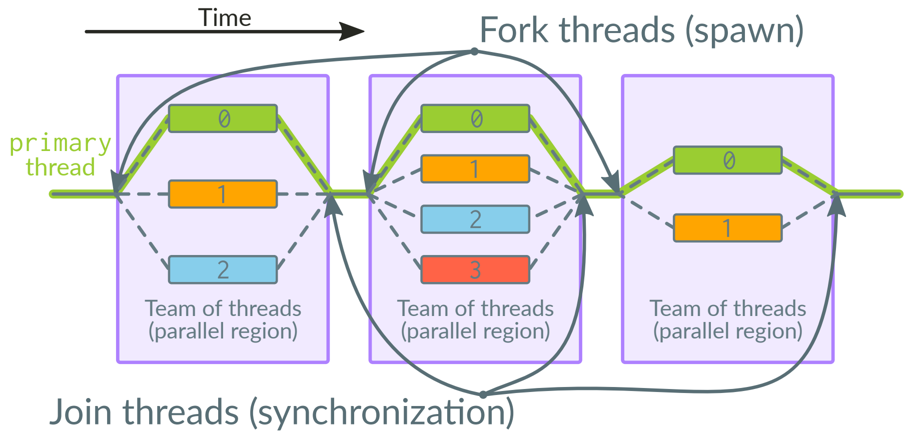
atomic operation

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

OpenMP EXECUTION MODEL (READING)

openmp.org

Fork/join model:



- Forking and joining threads is done *transparently* in OpenMP using *directives*
- Your program *always* executes with at least one thread (primary thread)
- The primary thread spawns a *team* of threads at the beginning of a *parallel region*
- The primary thread always has ID `0`

Parallel regions:

When a thread encounters a **`parallel`** construct, the thread creates a team of itself and zero or more additional threads. It becomes the *primary thread* of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **`parallel`** construct.

`parallel` regions may be arbitrarily *nested* inside each other.

Synchronization constructs and library routines are available in the OpenMP API to coordinate tasks and data access in **`parallel`** regions.

OpenMP MEMORY MODEL (READING)

- The OpenMP API provides a *relaxed* consistency shared memory model.
- Threads have a *temporary view* of the memory which includes registers, cache or other local storage.
- The temporary view allows threads to *cache* variables and therefore avoid going to main memory.
- Using caches in a multi-threaded environment introduces a new problem we have not talked about yet (lecture 9).

Types of variables inside parallel regions:

- A directive that accepts *data-sharing* attribute clauses determines *two kinds* of access variables used in the associated structured block: ***shared*** and ***private***.
- Each reference to a *shared variable* in the structured block becomes a reference to the original variable that exists outside of the structured block.
- For each *private variable* referenced in the structured block, a new version of the original variable is created in the thread private memory (of same type and size).

OpenMP PARALLEL REGIONS

- A program executes sequentially until it encounters a `parallel` directive.
- The thread that *encounters* the directive first creates a *team of threads* (children) and it becomes the *primary thread* with number `0`.
- At the end of each `parallel` region, there is an *implicit barrier* that synchronizes the threads and joins them together again. The *initial primary thread* then continues with sequential execution.

Directive to create parallel regions:

```
#pragma omp parallel [clause list] new-line
    structured block
```

Control clauses:

```
if([parallel:] scalar-expression)
num_threads(integer-expression)
reduction(reduction-identifier: list)
```

Data clauses:

```
default(data-sharing-attribute)
shared(list)
private(list)
firstprivate(list)
```

Part of next lecture

OpenMP specification Section 2.6

NUMBER OF THREADS IN PARALLEL REGIONS

The number of threads in a parallel region can be controlled by the following factors, in that order of precedence:

1. Result of the `if` clause
2. Setting the `num_threads` clause
3. Use of the `omp_set_num_threads()` library routine
4. Setting the `OMP_NUM_THREADS` environment variable at runtime
5. Fallback to *implementation default*. (This is typically the number of cores available on the machine but *do not rely on it!*)

Remember: threads within parallel regions are *always* numbered starting from `0` (primary thread) to `n-1` if there are `n` threads in the team. This also applies for *nested* parallel regions.

DYNAMIC AND STATIC MODES IN PARALLEL REGIONS

- ***Dynamic mode*** *(the default)*
  - The number of threads in different parallel regions *in the same program* ***may differ***.
  - The specified number of threads is *only an upper bound*. The actual number of threads *can be smaller* to allow for optimization of resources at the OpenMP and OS level.
- ***Static mode*** *(prefer this when running on HPC systems)*
  - The number of threads used in parallel regions is *static* and exactly equal to what the programmer specifies by means of the `num_threads` clause, `omp_set_num_threads()` routine or `OMP_NUM_THREADS` environment variable.

You can enforce *static mode* by setting the following environment variable in your launch scripts:

```
OMP_DYNAMIC=false
```

USING THREADS TO GET WORK DONE

- We now know how to spawn/fork a team of threads in OpenMP. The next step is to accomplish parallelism with that tool.
- ***Recall:*** there are two kinds of parallelism
  1. Data-level parallelism
  2. Task-level parallelism
- In OpenMP there are two concepts: ***worksharing*** and ***tasking***. The former (mostly) maps to data-level parallelism, the latter enables task-level parallelism

Worksharing: a way to distribute parallelizable work among a team of threads. Each thread then works on a *smaller subset* of the total work load in parallel.
Examples:

- Array traversal in `for`-loops (most prominent)
- Independent code sections (e.g. different independent functions)

Tasking: tasks can be used to exploit parallelism in *irregular* structures. A task consists of code, data and internal control variables. Tasks are added to a queue and may execute immediately or get scheduled for later execution. Tasks are cheap and help to avoid spawning threads dynamically.
Examples:

- Recursive functions
- List and tree traversal in parallel

OpenMP WORKSHARING CONSTRUCTS

OpenMP defines 3 worksharing constructs:

Loop construct: specifies that the iterations of *one or more* associated loops will be executed in parallel by the threads in the team. *This is worksharing construct is what you likely need most often.*

Sections construct: a *non-iterative* worksharing construct that contains a set of *structured blocks* that are to be distributed among and executed by the threads in a team.

Single construct: specifies that the associated structured block is executed by *only one* of the threads in the team (not necessarily the primary thread).

→ ***Important to remember:*** *all* worksharing constructs have an *implicit barrier* at the point of exit of the structured block, unless you use the `nowait` clause.

OpenMP LOOP CONSTRUCT

Parallelizing a loop "the raw way":

```c
#include <omp.h>

int main(void)
{
    const int N = 1000; // number of elements in array
    double A[N];        // shared data array we want to work on in parallel

#pragma omp parallel
    {
        const int tid = omp_get_thread_num();
        const int nthreads = omp_get_num_threads();
        int npoints = N / nthreads;        // work partition per thread
        const int mystart = tid * npoints; // per thread array offset
        if (tid == nthreads - 1) { // ensure to take care of remainders A
            npoints = N - mystart;
        }

        // iterate over array A in parallel
        for (int i = 0; i < npoints; ++i) {
            A[mystart + i] = ... // do something here
        }
    }
    return 0;
}
```

Tasks:

1. Get the worker information
2. Compute the work partition (similar to domain decomposition in MPI, you will see this later again)
3. Get the work done in parallel

OpenMP LOOP CONSTRUCT

Parallelizing a loop the OpenMP way:

```c
1  int main(void)
2  {
3      const int N = 1000; // number of elements in array
4      double A[N];        // shared data array we want to work on in parallel
5
6  #pragma omp parallel
7      {
8  #pragma omp for
9          for (int i = 0; i < N; ++i) {
10             A[i] = ... // do something here
11         }
12     }
13     return 0;
14 }
```

- In line 8 we tell OpenMP our intention: *share* the work (i.e. parallelize) associated to the loop that follows.
- OpenMP takes care of the work partitioning (including remainders if `N` does not divide evenly among the number of threads).
- Less code, less bugs, cleaner and more readable. See Section 2.11.4.

OpenMP LOOP CONSTRUCT

Parallelizing a loop the OpenMP way:

```c
int main(void)
{
    const int N = 1000; // number of elements in array
    double A[N];        // shared data array we want to work on in parallel

#pragma omp parallel
    {
        for (int i = 0; i < N; ++i) {
            A[i] = ... // do something here
        }
    }
    return 0;
}
```

What's wrong here?

CANONICAL LOOP STRUCTURES

The `#pragma omp for` construct works with the following *canonical* loop-structures:

```
                                  index++;
                                  ++index;
                                  index--;
                          <       --index;
for (index = start; index <= end; index += inc;       )
                          >=      index -= inc;
                          >       index = index + inc;
                          !=      index = inc + index;
                                  index = index - inc;
```

For C++ random access iterators:

since OpenMP v3.0

```
for (it = vec.begin(); it != vec.end(); it++)
```

since OpenMP v5.0

```
for (auto x : container)
```

Both of these variants satisfy the canonical loop form above. See Section 2.11.1 in the specification for the full documentation.

OpenMP SECTIONS CONSTRUCT

The `sections` construct is a non-iterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. *Each structured block is executed once* by one of the threads in the team in the context of its implicit task. See Section 2.10.1.

```
int main(void)
{
#pragma omp parallel
    {
#pragma omp sections // nowait [reduction(reduction-identifier: list)]
        {
#pragma omp section
            x_calculation(); // independent computation
#pragma omp section
            y_calculation(); // independent computation
#pragma omp section
            z_calculation(); // independent computation
        } // other threads in the team wait here. Use '#pragma omp sections
          // nowait' to let them continue without synchronization
    }
    return 0;
}
```

Note: you may also use the `reduction` clause with `sections`. (More about `reduction` in the following slides.)

OpenMP SINGLE CONSTRUCT

The `single` construct specifies that the associated structured block is executed by *only one of the threads in the team* (not necessarily the primary thread!), in the context of its implicit task. The other threads in the team, which do not execute the block, wait at an *implicit barrier* at the end of a `single` region unless a `nowait` clause is specified. See Section 2.10.2.

```c
int main(void)
{
#pragma omp parallel
    {
#pragma omp single // nowait
        {
            exchange_boundaries(); // irregularity here
        } // other threads in the team wait here w/o nowait clause

        // IFF this function does not depend on memory locations that need to
        // be visible after exchange_boundaries() has been executed, then
        // adding the 'nowait' clause to the single construct is more
        // efficient!
        do_many_things();
    }
    return 0;
}
```

BACK TO THE OpenMP LOOP CONSTRUCT

The loop construct is the most important of the OpenMP worksharing constructs. Let's look at it a bit closer:

```
#pragma omp for [clause[[,]clause]…] new-line
    for-nest
```

Control clauses:

```
schedule([modifier[, modifier]:]kind[, chunk_size])
collapse(n)
reduction([reduction-modifier,]reduction-identifier:list)
nowait
```

Data clauses:

```
private(list)
firstprivate(list)
lastprivate([lastprivate-modifier:]list)
```

Part of next lecture

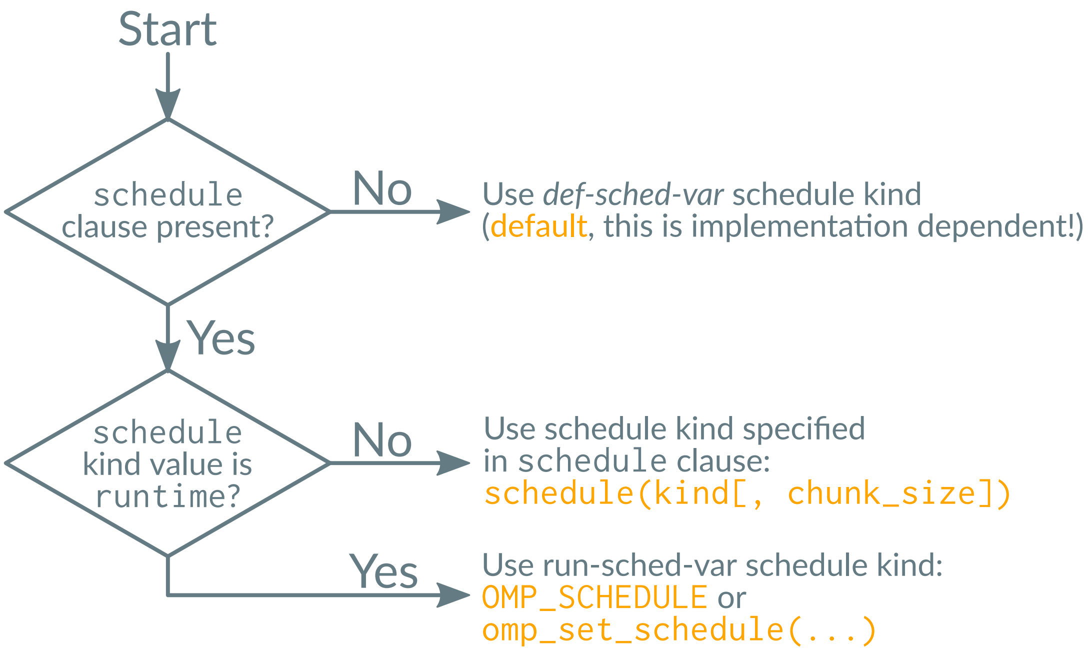
Note: there are a few more clauses that we do not cover in class. Please find a full list in Section 2.11.4 in the specification.

LOOP SCHEDULING

You have control over how work shall be distributed among threads when you parallelize `for`-loops. We ***do care*** about this because it allows to adjust the *parallel granularity* and control *load balancing*.

```
#pragma omp for schedule(kind[, chunk_size]) new-line
    for-nest
```

How does OpenMP determine the schedule:



- The *def-sched-var* is is an internal control variable and *dependents on the implementation* (`gcc` may do things differently by default than `clang` does)
- If you rely on a specific schedule, you must always specify it in your code.

LOOP SCHEDULING KIND

`static[, chunk]`

- Loop iterations are divided into segments of size `chunk` and distributed `cyclically` to the threads in the team
- If `chunk` is not specified, it *might* be similar in size to `N/nthreads` (implementation dependent, see Section 2.11.4 in specification)

`dynamic[, chunk]`

- Loop iterations are divided into segments of size `chunk`
- An *idle thread* is *dynamically* assigned the next available `chunk`. The `chunk` size defaults to `1` if not specified

`guided[, chunk]`

- Similar to `dynamic` but the `chunk` size *decreases exponentially*
- `chunk` specifies the *minimum* `chunk` size. Defaults to `1` if not specified

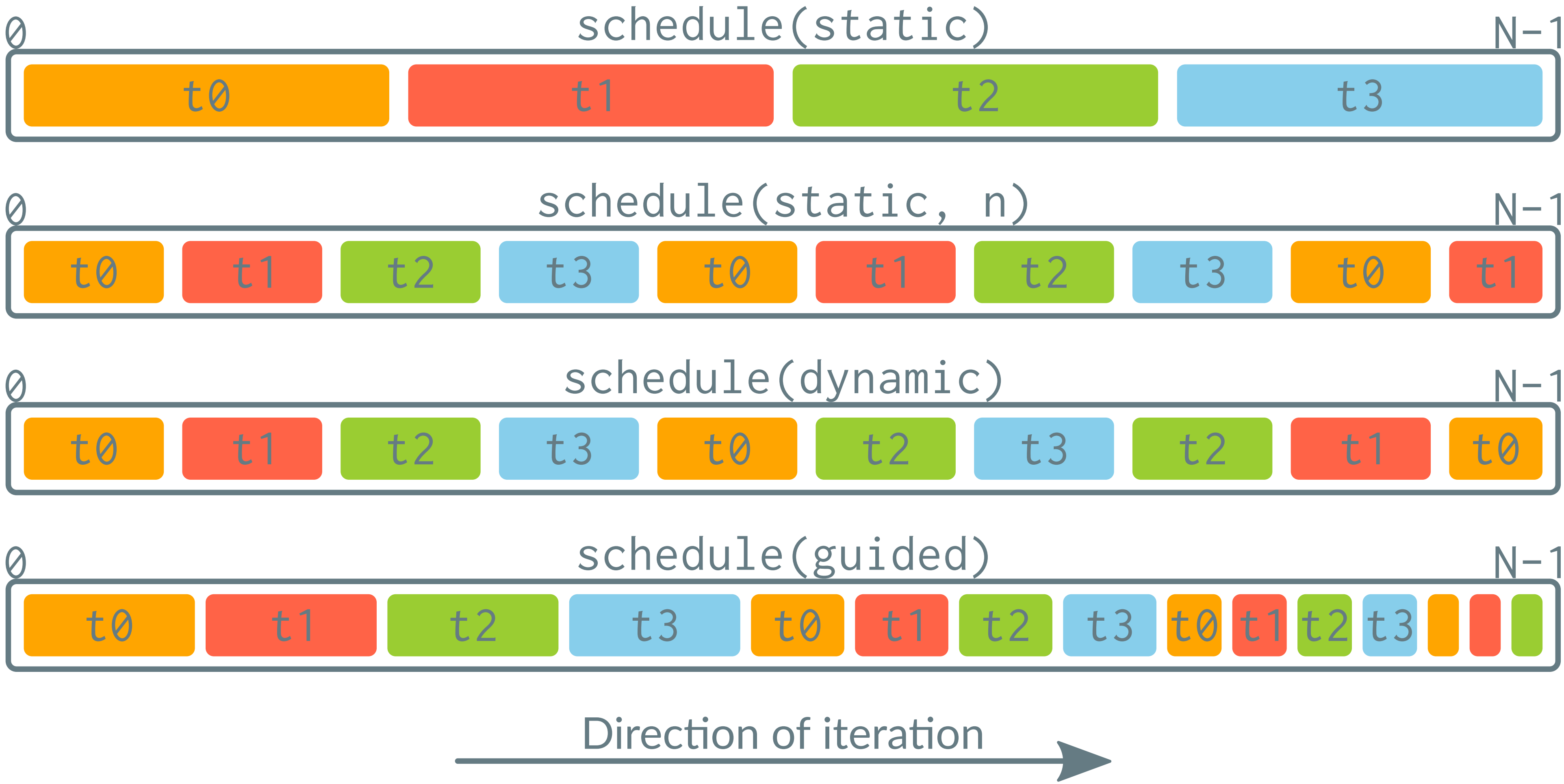
`runtime`

- Decide at runtime depending on the value of the `OMP_SCHEDULE` environment variable or the `omp_set_schedule()` library call at some point in your code

`auto`

- The decision is delegated to the compiler and/or the OpenMP runtime

LOOP SCHEDULING KIND VISUALLY



NESTED LOOPS

You can parallelize nested loops using OpenMP:

```
int main(void)
{
    const int N = 1000;
    double A[N][N]; // 2D array

#pragma omp parallel
    {
#pragma omp for
        for (int i = 0; i < N; ++i)     // this loop is run in parallel
            for (int j = 0; j < N; ++j) // each thread runs this loop from [0,N)
                A[i][j] = ...           // do something here
    }
    return 0;
}
```

- The "`#pragma omp for`" only applies to the `for`-loop that *immediately* follows it
- Each thread executes the innermost loop in full
- ***Important general remark for parallel loops:*** *recall that a structured block must have a single point of exit.* You ***cannot*** have a `break` statement in the outermost loop (the one to which "`#pragma omp for`" applies)!

COLLAPSING NESTED LOOPS

To increase the amount of work you parallelize, nested loops can be *collapsed* under certain conditions:

```
#pragma omp for collapse(n) new-line
    for-nest
```

- `n` must be a constant positive integer expression and denotes the number of loops to collapse
- You can only collapse *perfectly nested* loops

  Perfectly nested: the loop body of the outer loop can consist entirely of the inner loop
- The loops you collapse ***must not*** modify any of the loop iteration indices! (e.g. `i` and `j`)
- Benefits of collapsing loops:
  - Can achieve better resource utilization due to larger iteration space
  - Better load-balancing behavior of the parallelized loop

COLLAPSING NESTED LOOPS

Example: large difference in dimensions of array

```
int main(void)
{
    double A[5][10000]; // 2D array

#pragma omp parallel
    {
#pragma omp for collapse(2)
        for (int i = 0; i < 5; ++i)
            for (int j = 0; j < 10000; ++j)
                A[i][j] = ... // do something
    }
    return 0;
}
```

- Without the `collapse(2)` clause, *only 5 threads* work on the loop in parallel, the others are idle

Example: GEMV (matrix-vector product)

```
1  int main(void)
2  {
3      const int N = 1000;
4      double x[N], y[N]; // Vectors
5      double A[N][N];    // Matrix
6      // initialize data...
7
8      // parallel matrix vector product
9  #pragma omp parallel
10     {
11 #pragma omp for collapse(2)
12         for (int i = 0; i < N; ++i) {
13             y[i] = 0.0;
14             for (int j = 0; j < N; ++j)
15                 y[i] += A[i][j] * x[j];
16         }
17     }
18     return 0;
19 }
```

- Can you collapse the loops in line 12 and line 14?

REDUCTION

A reduction is a class of parallel algorithms that transform $\mathcal{O}(N)$ inputs to $\mathcal{O}(1)$ output by means of an associative binary operator $\oplus$. ***Examples include:***

- finding a global minimum or maximum
- summation of values or sum of squares
- product sums such as an inner product ubiquitous in linear algebra
- logical operations

See Section 2.21.5.4 in the OpenMP specification

```
#pragma omp for reduction(reduction-identifier: list) [other clauses] new-line
    for-nest
```

Reduction identifier: this is the binary operator $\oplus$. This operator can be one of the following built-in operators:

```
+, -, *, &, |, ^, &&, ||, min and max
```

- `list` is a comma separated list of reduction variables
- You can also create user-defined reduction operators. See Section 2.21.5.7
- The `reduction` clause can appear multiple times but not for the same reduction variable

Initial values: OpenMP creates *thread private* variables for the variables in the list. They are initialized as follows:

`+:`	`0`
`-:`	`0`
`*:`	`1`
`&:`	`~0`
`|:`	`0`
`^:`	`0`

`&&:`	1
`||:`	`0`
`min:`	Maximal representable number in the reduction list item type
`max:`	Minimal representable number in the reduction list item type

REDUCTION

Example: sum the elements in an array in parallel

```
1  int main(void)
2  {
3      double a[1000];
4      // initialize data...
5
6      double sum = 0.0; // sum the elements in here, must be initialized properly
7  #pragma omp parallel
8      {
9  #pragma omp for reduction(+: sum)
10         for (int i = 0; i < 1000; ++i)
11             sum += a[i];
12     }
13     return 0;
14 }
```

`line 6`

The variable `sum` is initialized to `0` and is stored in shared main memory.

`line 9`

OpenMP creates *private copies* of `sum` for each thread and initializes them to `0` (because of the `+` reduction operator).

`line 11`

Each thread sums up its array partition into its own private copy of `sum` (no race condition!). The result depends on the data the thread works on and will be different among threads.

`line 12`

OpenMP *reduces* the partial sums into the shared `sum` variable. It will *add* these partial sums to whatever the value of `sum` was before the reduction.

REDUCTION

Example: sum the elements in an array in parallel

```
int main(void)
{
    double a[1000];
    // initialize data...

    double sum = 0.0; // sum the elements in here, must be initialized properly
#pragma omp parallel
    {
#pragma omp for reduction(+: sum) nowait
        for (int i = 0; i < 1000; ++i)
            sum += a[i];
        process_result(sum);
    }
    return 0;
}
```

Is this code correct?

RECAP

- Introduction to OpenMP and the specification for the standard (your main reference)
- Basic terminology for OpenMP and shared memory systems
- Worksharing constructs in OpenMP are the main tools to parallelize your code
- Clauses to control work scheduling of parallelized `for`-loop (parallel granularity and load-balancing)
- How to collapse nested loops to increase resource utilization
- Reductions in OpenMP

Further reading:

- Sections 2.1, 2.4, 2.6, 2.10 and 2.21.5 in *OpenMP API Specification v5.1*, OpenMP, 2020 (pdf in `git` repo as well)
- Sections 5.1, 5.2, 5.3, 5.4, 5.5, 5.6 and 5.7 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Sections 17, 18, 19, 20, 21 and 22 in Eijkhout, "*Parallel Programming for Science and Engineering*", free pdf