HIGH PERFORMANCE COMPUTING FOR SCIENCE AND ENGINEERING

LECTURE 7/8

Fabian Wermelinger

Harvard University

CS205

Tuesday, February 14th/16th 2023

LAST TIME

- Introduction to OpenMP
- Fork/join model for threads
- Work sharing constructs
- Control of parallel granularity and load imbalance
- Nested loops
- Reductions with OpenMP

TODAY

Main topic: ***Data environments, synchronization and non-directive features in OpenMP***

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

Details:

- Combined constructs and lexical extent of parallel regions
- Data environment in OpenMP: what is shared and what is private?
- Synchronization primitives in OpenMP (***always remember:*** OpenMP uses a *relaxed* consistency model)
- OpenMP library routines and environment variables

AGENDA CHECK:

- Quiz 2 this Thursday covers lectures 4-7 in schedule. 15 minutes quiz with 10 multiple choice/true/false questions.
- Project milestone 2 → High-level project description. Please include how you plan to do the parallelization (shared memory, distributed memory, both?). Please be brief, aim at about half a page, no more than one page. Please see https://harvard-iacs.github.io/2023-CS205/pages/project.html#M2 for the details. Due February 21nd 11:59pm. Commit the report to `milestone/2/hl_summary.pdf` in your team repo at https://code.harvard.edu/CS205.

DIRECTIVE SCOPING

Remember (and recall from last time): none of the worksharing constructs we have seen so far *create* teams of threads to execute in parallel. They *all* must be executed *inside* a parallel region, where the team already exists!

```
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 }
```

- Lines 6, 7 and 12 fork and join a parallel region, respectively.
- Lines 8-11 create a *worksharing construct*. It assumes to execute *inside* a parallel region.

DIRECTIVE SCOPING

- OpenMP allows to *extend* (existing) code in *multiple files*
- OpenMP directives that appear *outside* a parallel region are termed *orphaned constructs*

main.cpp

```cpp
#include "myfunc.h"
#include <omp.h>

int main(void)
{
    const int N = 1000;
    int A[N];

#pragma omp parallel
    {
        const int tid = omp_get_thread_num();
        myfunc(A, N, tid);
    }
    // The `#pragma omp for` directive has no effect in the dynamic extent
    myfunc(A, N);
    return 0;
}
```

myfunc.h

```cpp
void myfunc(int A[], const int N, const int tid = 0)
{
#pragma omp for
    for (int i = 0; i < N; ++i)
        A[i] = tid;
}
```

- Static (lexical) extent of parallel region
- Dynamic extent of parallel region
- Orphaned construct

IMPLICIT BARRIERS

Important: worksharing constructs have an *implicit barrier* at the end!
Recall: a barrier is a *synchronization* and we must avoid them if they are not required for program correctness.

What is bad in this code?

```c
int main(void)
{
    const int N = 1000;
    double A[N];

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

- There are ***two*** synchronizations happening here (barriers)
- One is implicitly added after the worksharing construct (after the structured block that follows `#pragma omp for`)
- Another one required to join the threads when the parallel region ends

IMPLICIT BARRIERS

Important: worksharing constructs have an *implicit barrier* at the end!
Recall: a barrier is a *synchronization* and we must avoid them if they are not required for program correctness.

If you do not need to wait, don't:

```c
int main(void)
{
    const int N = 1000;
    double A[N];

#pragma omp parallel
    {
#pragma omp for nowait
        for (int i = 0; i < N; ++i) {
            A[i] = // do something here
        } // synchronization not necessary
    } // necessary synchronization (join threads)
    return 0;
}
```

- Synchronization after the `for`-loop is clearly not necessary here
- If you ignore it you will get worse code. The compiler may generate different code, *you cannot rely on the compiler to help you out here*. The `nowait` clause is there for a reason!
- You can skip the *implicit barriers* of all OpenMP worksharing constructs by adding the `nowait` clause.

COMBINED CONSTRUCTS

There is *only one single* `for`-loop in this parallel region. Do we need all this boilerplate code? ***No***, ***combined constructs*** are shortcuts offered by OpenMP to write only one construct that is immediately *nested* inside another construct. You can apply this to the `for` and `sections` constructs we have seen so far. See Section 2.16 in the specification.

This code:

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

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

Is identical to:

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

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

Can you apply the `nowait` clause here?

COMBINED CONSTRUCTS

Depending on your algorithm of course, you may not always be able to combine constructs. If you can, it is for readability of your code and performance.

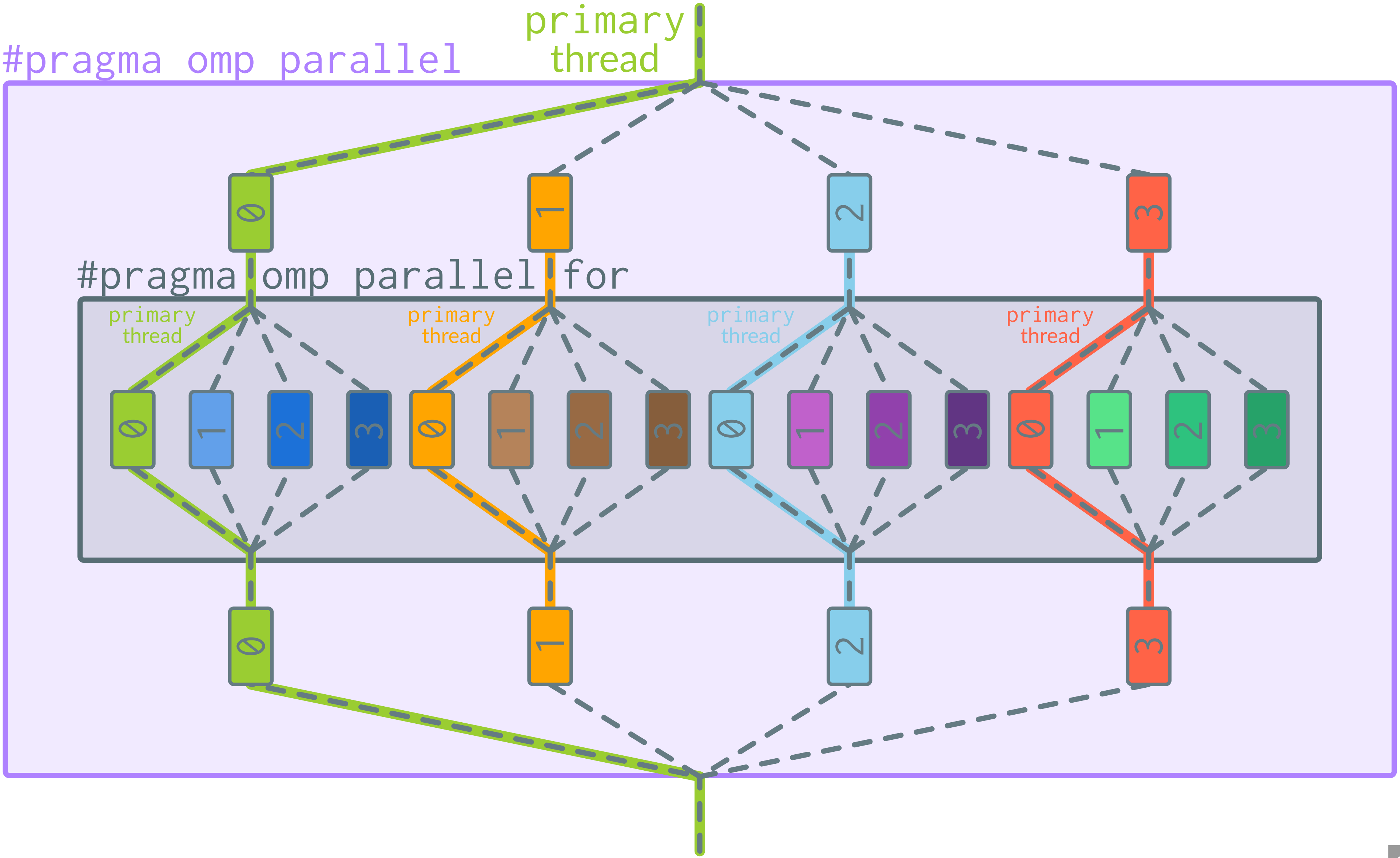
Be careful with the following (this is a ***common beginners mistake***).
What is happening in this code?

```c
int main(void)
{
    const int N = 1000;
    double A[N];

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

- This results in ***nested parallelism***. OpenMP allows for this form of parallelism when the environment variable `OMP_MAX_ACTIVE_LEVELS` is set to a positive integer larger than 1. The default is *implementation dependent*. (***Note:*** `OMP_NESTED` is *deprecated*.)
- Each thread in the team of the parallel region will fork a new team of threads that work cooperatively on the *whole* loop → ***not parallel!***
- You have data races and most likely resource over-subscription.

NESTED PARALLELISM



OpenMP DATA ENVIRONMENT

How can you control the shared memory environment with OpenMP?

- Variables in the scope outside of a parallel region are *shared by default* inside a parallel region (as in shared memory).
- Global variables or variables created inside a parallel region which have *static storage* duration are *shared* (i.e. using the `static` keyword).
- Other variables of *automatic storage* duration created inside a parallel region are *private* for each thread. This includes for example functions you call in a parallel region (e.g. the function arguments).
- Loop index variables are always private.

Recall: we must avoid synchronization. Working with many shared variables necessitates a lot of synchronization which can often be avoided if you work with *thread private* variables instead. You must use the tools of the OpenMP data environment to come up with clever solutions that minimize synchronization points.

OpenMP DATA SHARING ATTRIBUTES

The data-sharing attributes of variables that are referenced in a *construct* can be (see Section 2.21.1.1 in the specification):

1. ***Predetermined*** (e.g. loop indices or `threadprivate` variables)
2. ***Explicitly determined*** (anything you specify explicitly in the clause of a construct)
3. ***Implicitly determined*** (everything else reverts back to defaults)

Note: attributes specified via data-sharing clauses can only be used within the *lexical extent* of the directive where you specify them.

For the data-sharing attributes of variables that are referenced in a region, *but not in a construct* see Section 2.21.1.2 for the full list. Some important ones listed below: (a *region* is a function call for example)

- Variables with static storage duration that are declared in called routines in the region are *shared*
- Static data members are shared unless they appear in a `threadprivate` directive
- Objects with dynamic storage duration are *shared* (memory allocated on heap)
- Other variables declared in called routines in the region are private

OpenMP DATA SHARING ATTRIBUTES

Data-sharing clauses we skipped in last lecture:

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. *Not all of the clauses listed below are valid on all directives.* The set of clauses that is valid on a particular directive is described with the directive.

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

- These two can only be used with the `parallel` directive (as well as the `teams` and `task` constructs which we did not discuss in class).
- See Section 2.21.4 in the specification for full details.

```
private(list)
firstprivate(list)
lastprivate(list)
```

- These can be used with all the constructs we have seen so far.
- *Exception:* `lastprivate` cannot be used with the `parallel` construct.

`default` DATA ENVIRONMENT

This data environment clause allows you to explicitly specify the attributes of variables you reference in `parallel`, `teams` or `task` constructs. If not specified explicitly the attributes are determined implicitly according to the rules on the previous slide. See Section 2.21.4.1 in the specification.

```
default(shared | private | firstprivate | none)
```

Example:

```c
int main(void)
{
    int var;
#pragma omp parallel default(none) shared(var)
    {
        // The variable `var` is explicitly shared.
        // Without the two clauses above, the following
        // is identical:
        //   #pragma omp parallel
        // but now `var` is implicitly determined!
    }
    return 0;
}
```

- The `default` clause must only appear once on a construct.
- `none` means that every variable you reference in the `parallel` region must have *explicitly determined* data-sharing attributes (except if they are *predetermined*).

(See the lecture code `omp_default.cpp` to find out what happens when you remove `shared(var)` and reference `var` inside the parallel region.)

`shared` AND `private` DATA ENVIRONMENTS

The `shared` and `private` clauses declare one or more list items to be shared or thread private, respectively. See Sections 2.21.4.2 and 2.21.4.3 in the specification.

```
shared(list)  (mostly the default behavior in OpenMP)
private(list)
```

`private` example:

```
1  int main(void)
2  {
3      int sum = 0;
4  #pragma omp parallel for private(sum)
5      for (int i = 0; i < 1000; ++i)
6      {
7          // threads have not initialized sum, its value is
8          // undefined! Inside this region, each thread
9          // observes a different address for the sum
10         // variable.
11         sum = sum + i; // NOT a race condition!
12     }
13     // Because sum has been declared private in the
14     // parallel region, the original list item has
15     // the same value after the parallel region as it
16     // has before entering the parallel region.
17     return 0;
18 }
```

- Creates a *private copy* of each variable in the list.
- The private copies are new instances of the same type (no data races!).
- The private copies *are **not** initialized*. There is *no association* to the original list item
- The initial value of `sum` in line 11 is *undefined*. See Section 2.21.3 List Item Privatization.

`firstprivate` DATA ENVIRONMENTS

The `firstprivate` clause declares one or more list items to be `private` to a thread, and *initializes* each of them with the value that the corresponding original item has when the construct is encountered. See Section 2.21.4.4 in the specification.

```
firstprivate(list)
```

`firstprivate` example:

```c
1  int main(void)
2  {
3      int sum = 0;
4  #pragma omp parallel for firstprivate(sum)
5      for (int i = 0; i < 1000; ++i)
6      {
7          // sum has been initialized to sum = 0. Inside
8          // this region, sum is again thread private but
9          // now properly initialized.
10         sum = sum + i;
11     }
12     // Because sum has been declared firstprivate in the
13     // parallel region, its value after the parallel
14     // region is the same as it was before entering the
15     // region.
16     return 0;
17 }
```

- Special case of `private(`*`list`*`)`
- The private copy *is* initialized to the value of the original list item.
- The association of the private copy with the original list item is its *initialization*.
- As opposed to the example on the previous slide, the updates in line 10 do not produce undefined behavior this time.

`lastprivate` DATA ENVIRONMENTS

The `lastprivate` clause declares one or more list items to be `private` to a thread, and causes the corresponding original list item to be updated after the end of the region. See Section 2.21.4.5 in the specification.

```
lastprivate(list)
```

`lastprivate` example:

```c
int main(void)
{
    int a = 0;
#pragma omp parallel for lastprivate(a)
    for (int i = 0; i < 100; ++i)
        a = i;
    // after the parallel region: a = 99
    return 0;
}
```

- Another special case of `private(`*`list`*`)`
- The private copy *is not initialized* (same as in `private(`*`list`*`)`)

```c
int main(void)
{
    int a = 0;
#pragma omp parallel sections lastprivate(a)
    {
#pragma omp section
        // lexically first section
        a = 1;
#pragma omp section
        // lexically second section (and last)
        a = 2;
    }
    // after the parallel region: a = 2
    return 0;
}
```

The association of the private copy with the original list item is its *update after the parallel region* (***careful:*** *data races!*).

`lastprivate` DATA ENVIRONMENTS

`lastprivate` example: with `nowait` clause

```
1  int main(void)
2  {
3      int a = 0;
4  #pragma omp parallel
5      {
6  #pragma omp sections lastprivate(a) nowait
7          {
8  #pragma omp section
9              // lexically first section
10             a = 1; // `a` is a private copy in this region
11 #pragma omp section
12             // lexically second section (and last)
13             a = 2; // `a` is a private copy in this region
14         } // lastprivate(a) ends here
15         // references to `a` from this point on are to the shared `a` on line 3.
16         // What is the value of `a` if you reference it here?
17     }
18     return 0;
19 }
```

What is the value of a *immediately after* the `sections` construct?

OpenMP SYNCHRONIZATION PRIMITIVES

Recall: we need synchronization primitives to ensure program correctness due to a relaxed memory consistency model. They are *expensive* and not parallel, only make use of them when necessary. See Section 2.19 in the specification.

What we look at next:

```
#pragma omp critical new-line
#pragma omp atomic new-line
#pragma omp barrier new-line
#pragma omp master new-line (deprecated)
#pragma omp masked new-line
```

- The last two *are not* synchronization primitives, but we discuss them in this section of the slides.
- There are a few more synchronization constructs that we are not going to discuss in class.
- You can find them all in Section 2.19 of the OpenMP specification (v5.1)

`critical` CONSTRUCT

The `critical` construct provides *mutual exclusion* in critical sections. See Section 2.19.1 in the specification.

- Under the hood it uses an OpenMP lock to provide *mutual exclusion* (operations when the lock is acquired are guaranteed mutually exclusive).
- The `critical` construct is a convenience shortcut.
- ***Recall:*** a critical section is *sequential* (no parallelism there)

```
#pragma omp critical [(name) [[,] hint(hint-expression)]] new-line
    structured-block
```

- You can *name* a critical section for more fine-grained control. Any thread waiting to enter a critical section with *name* ***must wait*** until no other thread executes code ***in any other or the same*** critical section with the same *name*.
- If you do not specify a *name*, then a unique global name is used for *all* your critical sections.
- You can provide a synchronization *hint-expression* to configure the underlying lock. In most cases you don't need this. See Section 2.19.12 for more details.

`critical` CONSTRUCT

```
#pragma omp critical [(name) [[,] hint(hint-expression)]] new-line
    structured-block
```

Example:

```c
#include <omp.h>
int main(void)
{
    int a = 0;
    int b = 0;
#pragma omp parallel
    {
        const int tid = omp_get_thread_num();
#pragma omp critical(critical_a)
        {
            a += tid;
        }
#pragma omp critical(critical_b)
        {
            // if the operation would involve a read or
            // write from/to `a`, then this would be a
            // race condition
            b += tid;
        }
    }
    return a + b;
}
```

- If you *name* `critical` sections you sign a contract with OpenMP (*you are responsible to use them correctly without generating race conditions*).
- It is possible that the two named `critical` sections execute in parallel because they both use different underlying locks.
- This can be useful in task based parallelism.
- See the `omp_critical` lecture code.

`atomic` CONSTRUCT

> The `atomic` construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values. See Section 2.19.7 in the specification.

- The `atomic` is similar to `critical` but *much more efficient* at the cost of simplicity.
- Efficient locks rely on special hardware instructions called compare-and-swap (CAS) that allow to check and possibly modify a memory location atomically using *transactional memory* (it is *exactly one* instruction, granularity cannot get smaller than this → hence atomic). A lock (mutex) can provide mutual exclusion for *a large region of code with many statements*.
- The `atomic` construct uses CAS instructions *directly on the memory location you want to update*. This is much more efficient but limited to only *one* memory location that you can modify at a time using simple arithmetic, update or compare operations.

`atomic` CONSTRUCT

The `atomic` construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values. See Section 2.19.7 in the specification.

```
#pragma omp atomic [clause[[,] clause]...] new-line
    statement
```

Clauses: can be an *atomic-clause*, *memory-order-clause* (relates to memory consistency model) or a further operation related clause.

Atomic-clauses:

- `read`
- `write`
- `update` (default if not specified)

Operation related clauses:

- `capture` (***hint:*** could be used in HW1 P2 for best efficiency)
- `compare`
- And a few more advanced ones not discussed here (See Section 2.19.7)

Example:

```c
#include <omp.h>
int main(void)
{
    double result = 0.0;
#pragma omp parallel
    {
        const int tid = omp_get_thread_num();
        const double my_result = do_work(tid);
        // `update` is default clause if not
        // specified
#pragma omp atomic update
        // simple (update) statements only:
        result += my_result;
    }
    return 0;
}
```

`barrier` CONSTRUCT

The `barrier` construct specifies an explicit barrier at the point at which the construct appears. The `barrier` construct is a stand-alone directive. See Section 2.19.2 in the specification.

```
#pragma omp barrier new-line
```

- Threads are only allowed to continue *once all threads in the team* have executed the barrier.
- → Your code will *deadlock* if this requirement is not met!
- The `barrier` construct is a *standalone* directive (no structured block associated).

Example:

```
1  #include <omp.h>
2  int main(void)
3  { // arrays `a` and `b` are defined globally
4  #pragma omp parallel
5      {
6          const int tid = omp_get_thread_num();
7          a[tid] = big_calculation(tid);
8  #pragma omp barrier // must synchronize here
9  #pragma omp for nowait
10         for (int i = 0; i < 1000; ++i)
11             b[i] = another_big_calculation(i, a);
12     }
13     return 0;
14 }
```

- A `barrier` is required in this code because the next computation in line 11 passes array `a` as an argument to the function.
- ***Remember:*** worksharing constructs in OpenMP have an implicit barrier at their point of exit. You can use the `nowait` clause to skip these barriers but beware of race conditions.

masked CONSTRUCT

The `masked` construct specifies a structured block that is executed by a *subset* of the threads of the current team. ***Note:*** *this is not a synchronization construct → there is* ***no implicit barrier*** *at the point of exit.* See Section 2.8 in the specification.

```
#pragma omp masked [filter(integer-expression)] new-line
    structured-block
```

- Only the threads that are masked by the filter expression execute the `masked` region.
- Without the `filter` clause, `#pragma omp masked` is identical to `#pragma omp master` (*deprecated*). Only the *primary* thread executes this region.
- No implied barrier occurs either on entry to, or exit from, the `masked` construct.
- The `masked` construct is very recent and introduced in specification v5.1. It is implemented in `gcc` v12.0 only. On the academic cluster, if you use a GCC version older than that you will use OpenMP v4.5 for which only the `master` construct is implemented. See https://gcc.gnu.org/wiki/openmp for more details.

Example:

```
int main(void)
{
#pragma omp parallel
    {
        do_work();
// Note: this is identical:
// #pragma omp single nowait
#pragma omp master
        {
            irregularities();
        } // no barrier here!
        // do_more_work must be
        // thread-safe w/o barrier
        do_more_work();
    }
    return 0;
}
```

OpenMP RUNTIME

- 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

- *All we talked about so far were compiler directives!*
- For the most part when working with OpenMP, the directives are the tools you need to get the parallelization done.
- In the remaining slides we talk about some tools from the runtime environment in OpenMP. These tools are useful for more fine-grained control over your implementation.
- The runtime library routines are documented in Chapter 3 and the environment variables are documented in Chapter 6 in the specification.

OpenMP LIBRARY ROUTINES

- OpenMP is an Application Programming Interface (API).
- That means it is a *library* which provides routines (functions) to perform certain tasks.
- To use the API you must include the *declarations* of these routines in your code

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

- At this point, the `omp.h` header (and library object file) must be available on any system you want to compile your code. It implies adding the `-fopenmp` flag to `gcc`.
- If your code must support systems and/or compilers where this cannot be guaranteed, use the `_OPENMP` macro for *conditional* compilation:

```
#ifdef _OPENMP
#include <omp.h>
#endif /* _OPENMP */
```

In the above code, the OpenMP header is only included if the `_OPENMP` macro is defined which in turn will *only* be defined when the `-fopenmp` flag is passed to the compiler. (***Note:*** different compilers may use different flags to enable OpenMP. See the compiler documentation for the correct flag.)

OpenMP LIBRARY ROUTINES

Example: setting dynamic mode in your code explicitly:

Via environment variable:

```
OMP_DYNAMIC=true
OMP_DYNAMIC=false
```

This setting will be *global* and applies to all your parallel regions in your code.

Via runtime library:

```
void omp_set_dynamic(int flag);
int omp_get_dynamic(void);
```

This setting will be *local* in your code and applies only to parallel regions that follow after you enable or disabled dynamic mode. When `flag` evaluates to `true` (different from `0`) dynamic mode will be set. See Section 3.2.6 and 3.2.7 in the specification.

Example: setting maximum levels of *nested parallelism* explicitly:

Via environment variable:

```
OMP_MAX_ACTIVE_LEVELS=2 # 2 levels
OMP_NESTED=true # deprecated
```

These settings will again be *global* and apply to all your parallel regions in your code.

Via runtime library:

```
void omp_set_max_active_levels(int max_levels);
int omp_get_max_active_levels(void);
void omp_set_nested(int flag); // deprecated
int omp_get_nested(void); // deprecated
```

This setting will be *local* in your code and applies only to parallel regions that follow after you set the maximum levels of nested parallelism. *The maximum level is implementation dependent*. See Section 3.2.15 and 3.2.16 in the specification.

USEFUL ROUTINES YOU MAY NEED FREQUENTLY

Thread control:

- Set the number of threads to be used for subsequent `parallel` regions (without using the `num_threads` clause):

```
void omp_set_num_threads(int num_threads);
```

- Get the number of threads in the *current team* that executes the parallel region:

```
int omp_get_num_threads(void);
```

- Returns the upper-bound of threads that *could* be used to form a new team of threads:

```
int omp_get_max_threads(void);
```

  Use this return value to allocate dynamic storage guaranteed to be large enough for all threads.

- Returns the thread number for the calling thread within the current team:

```
int omp_get_thread_num(void);
```

Processor level:

- Get the number of processors available on the current device:

```
int omp_get_num_procs(void);
```

Check if in parallel region:

- Returns `true` (integer other than `0`) if called by a thread inside a parallel region:

```
int omp_in_parallel(void);
```

Time measurement:

- Returns the elapsed wall clock time in seconds:

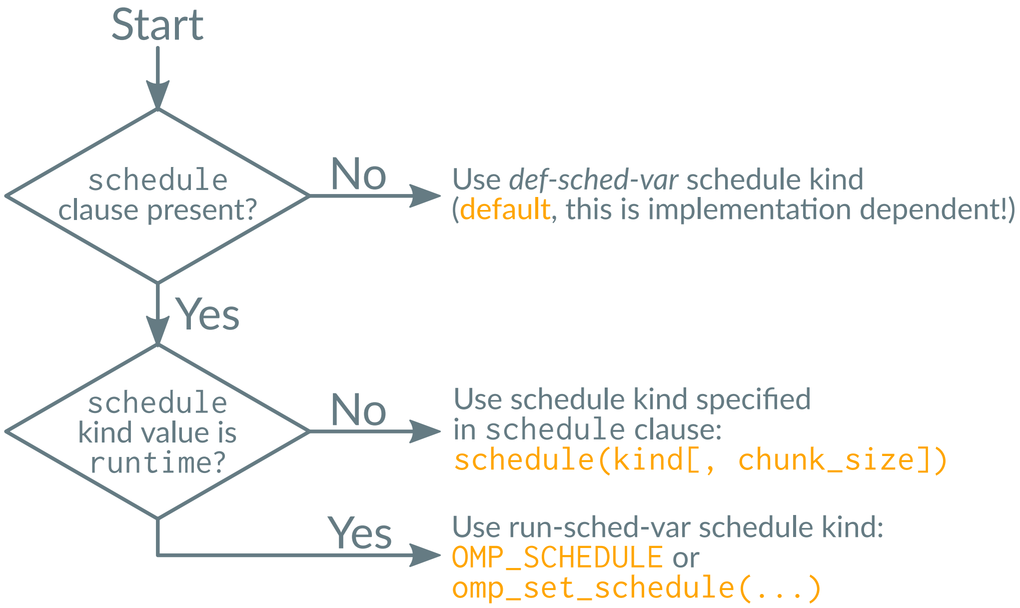
```
double omp_get_wtime(void);
```

- Returns the precision (time between two clock ticks) used by `omp_get_wtime`:

```
double omp_get_wtick(void);
```

LOOP SCHEDULING (RECALL LAST LECTURE)

You can change the scheduling policy in your code for places where you have used the `schedule(runtime)` clause (see Section 3.2.11):




```c
void omp_set_schedule(omp_sched_t kind, int chunk_size);
void omp_get_schedule(omp_sched_t *kind, int *chunk_size);
// defined in omp.h header:
typedef enum omp_sched_t {
  omp_sched_static = 0x1,
  omp_sched_dynamic = 0x2,
  omp_sched_guided = 0x3,
  omp_sched_auto = 0x4,
} omp_sched_t;
```

Example:

```cpp
#include <omp.h>
int main(void)
{
#pragma omp parallel
    {
        // set a desired schedule.  This will
        // overwrite what has been set in the
        // OMP_SCHEDULE environment variable:
        omp_set_schedule(omp_sched_dynamic, 1);

        // test it
        omp_sched_t sched;
        int chunk;
        omp_get_schedule(&sched, &chunk);
#pragma omp single nowait
        std::cout << "schedule = " << sched
                  << ", chunk = " << chunk
                  << '\n';

#pragma omp for schedule(runtime)
        for (int i = 0; i < N; ++i) {
            // do work with schedule determined
            // in software above
        }
    }
    return 0;
}
```

OpenMP LOCKS

The OpenMP runtime library provides locks for mutual exclusion. Most of the time you will not need them because the directives will make use of them *transparently*. See Section 3.9 in the specification.

- Initialize a lock to *unlocked* state. You must call this routine before you can use an OpenMP lock:

```
void omp_init_lock(omp_lock_t *lock);
```

- Ensure the allocated lock resources are released again when you no longer need it:

```
void omp_destroy_lock(omp_lock_t *lock);
```

- Acquire the lock (this will *block* the calling thread if the lock is not available):

```
void omp_set_lock(omp_lock_t *lock);
```

- Release the lock for others to make progress:

```
void omp_unset_lock(omp_lock_t *lock);
```

OpenMP LOCKS

openmp.org

Mutual exclusion the hard way:

```c
#include <omp.h>

int main(void)
{
    omp_lock_t mutex;
    omp_init_lock(&mutex);
#pragma omp parallel
    {
        omp_set_lock(&mutex);
        // critical section
        omp_unset_lock(&mutex);
    }
    omp_destroy_lock(&mutex);
    return 0;
}
```

OpenMP `critical`:

```c
int main(void)
{
#pragma omp parallel
    {
#pragma omp critical
        {
            // critical section
        }
    }
    return 0;
}
```

Both of these codes achieve the same, i.e., provide mutual exclusion for a critical section. The *directive* variant is more elegant and possibly allows for better optimizations at compile time.

(You will play with both of these variants in Lab 2)

ENVIRONMENT VARIABLES

- The most flexible (but with less control) to configure your OpenMP runtime is with *environment variables*.
- The OpenMP API manages a set of internal control variables (see Section 2.4) (ICVs) that control the behavior of an OpenMP program.
- These variables are assigned an *implementation dependent* default value.
- You can overwrite the default using *environment variables*.
- Modifying ICVs through environment variables is only possible *before your program is running*. Changing environment variables *while your program is running* has no effect. You must use the OpenMP library routines in that case (see previous slides).
- See Section 6 in the OpenMP specification.

Note: we will use a `bash` shell to discuss environment variables in the following. They behave the same for a `zsh` shell or, with different syntax, in `csh` or `ksh` shells.

MOST USEFUL ENVIRONMENT VARIABLES

Dynamic mode:

- Enable or disable *dynamic mode*. The default is *implementation dependent*. If dynamic adjustment of the number of threads is supported it will be `true`, otherwise `false`:

```
OMP_DYNAMIC=true  # enable dynamic mode
OMP_DYNAMIC=false # disable dynamic mode
```

Nested parallelism:

- Enable or disable *nested parallelism*. The default is implementation dependent. If nested parallelism is supported it will be enabled, otherwise disabled:

```
OMP_MAX_ACTIVE_LEVELS=1 # no nested parallelism
OMP_NESTED=true # deprecated
```

Thread control:

- Set the number of threads to use for `parallel` regions. The value of this environment variable is returned by calling `omp_get_max_threads()`. If *dynamic mode* is enabled, your code may run with *fewer* threads depending on available system resources (an explicit `num_threads(n)` clause takes precedence):

```
OMP_NUM_THREADS=8   # for level 1
OMP_NUM_THREADS=8,4 # for level 1 and 2
```

Scheduling:

- Set the scheduling policy for all loops with the `schedule(runtime)` clause specified. You can overwrite this value using `omp_set_schedule()` at runtime:

```
OMP_SCHEDULE='guided,4' # guided with chunk size 4
OMP_SCHEDULE='static,2' # static with chunk size 2
```

EFFICIENT USE OF ENVIRONMENT VARIABLES

- Environment variables exists because the *common practice* is to write job scripts:
  - Job scripts *document* how you ran a job
  - They allow for reproducibility
  - They help you organize your submissions
  - Environment variables are meant to be used with job configuration scripts
- Use `OMP_DISPLAY_ENV` to print a summary of your OpenMP configuration at job start. Helpful when you later review your job log!
- `SLURM_CPUS_PER_TASK` is set by SLURM and should be preferred if the job scheduler offers this environment variable (for consistency).

Lab 1, task 3: example `job.sh` script

```bash
#!/usr/bin/env bash
#SBATCH --job-name=hybrid_4nodes
#SBATCH --output=hybrid_4nodes_%j.out
#SBATCH --error=hybrid_4nodes_%j.err
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=32
#SBATCH --time=00:30:00
#SBATCH --mail-user=you@seas.harvard.edu
#SBATCH --mail-type=ALL

# you may use any bash commands in here
module load \
    gcc/12.1.0-fasrc01 \
    openmpi/4.1.3-fasrc01

# configure OpenMP runtime
export OMP_DISPLAY_ENV='verbose'
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
export OMP_SCHEDULE='static'
export OMP_DYNAMIC='false'

# here we run our program with the OpenMP and
# #SBATCH configuration above (see man srun):
srun ./hybrid_node "$@"
exit $?
```

RECAP

- Combining OpenMP constructs for convenience. *Be careful with nested parallelism.*
- The basic OpenMP data environment clauses are important to understand.
- Overview of the most important OpenMP synchronization constructs. Use them only when needed.
- OpenMP runtime environment: usage of library routines and environment variables.

Further reading:

- Sections 2.16, 2.17, 2.19, 2.21, 2.22, 3 and 6 in *OpenMP API Specification v5.1*, OpenMP, 2020 (pdf in `git` repo as well)
- Sections 5.5, 5.6, 5.7 and 5.8 in Pacheco, *An Introduction to Parallel Programming*, Morgan Kaufmann, 2011
- Sections 22, 23 and 26 in Eijkhout, *"Parallel Programming for Science and Engineering"*, free pdf
- Chapter 6 in Hager and Wellein, *"Introduction to High Performance Computing for Scientists and Engineers"*, CRC Press, 2011