Code of the Day

```
void task1() {
  // Lock mutex1 first, then mutex2
  std::lock_guard<std::mutex> lock1(mutex1);
  std::cout << "Task 1 locked mutex1\n";

  // Simulate some work
  std::this_thread::sleep_for(std::chrono::milliseconds(1));

  std::lock_guard<std::mutex> lock2(mutex2);
  std::cout << "Task 1 locked mutex2\n";
}
```

```
void task2() {
  // Lock mutex2 first, then mutex1
  std::lock_guard<std::mutex> lock2(mutex2);
  std::cout << "Task 2 locked mutex2\n";

  // Simulate some work
  std::this_thread::sleep_for(std::chrono::milliseconds(1));

  std::lock_guard<std::mutex> lock1(mutex1);
  std::cout << "Task 2 locked mutex1\n";
}
```

```
int main() {
  std::thread t1(task1);
  std::thread t2(task2);
  t1.join();
  t2.join();
  return 0;
}
```

What problem can you see from this code?

Will the problem always occur?

ECE 455

GPU Algorithm and System Design

[Fall 2025]

OpenMP Basics

09/22/2025

Before we get started…

- Quick overview, things discussed last time
  - Thread-level parallelism vs process-level parallelism
    - Process: An isolated virtual memory space that wraps the execution of your program
    - Thread: A basic execution unit living inside a process to run your program, starting with "`main`" function
  - Parallel programming using C++ thread (`std::thread`, `std::mutex`, `std::condition_variable`)

- Purpose of today's lecture:
  - Introduce the basic usage of OpenMP
  - Use OpenMP to parallelize a loop of independent work

- Miscellaneous
  - Lab Assignment #2 due on **Friday 9/26 at 23:59 PM**

Multi-core parallel computing with OpenMP

OpenMP: https://www.openmp.org/

- **OpenMP** is an Application Programming Interface (API) that supports shared-memory multithreaded programming in C, C++, and Fortran

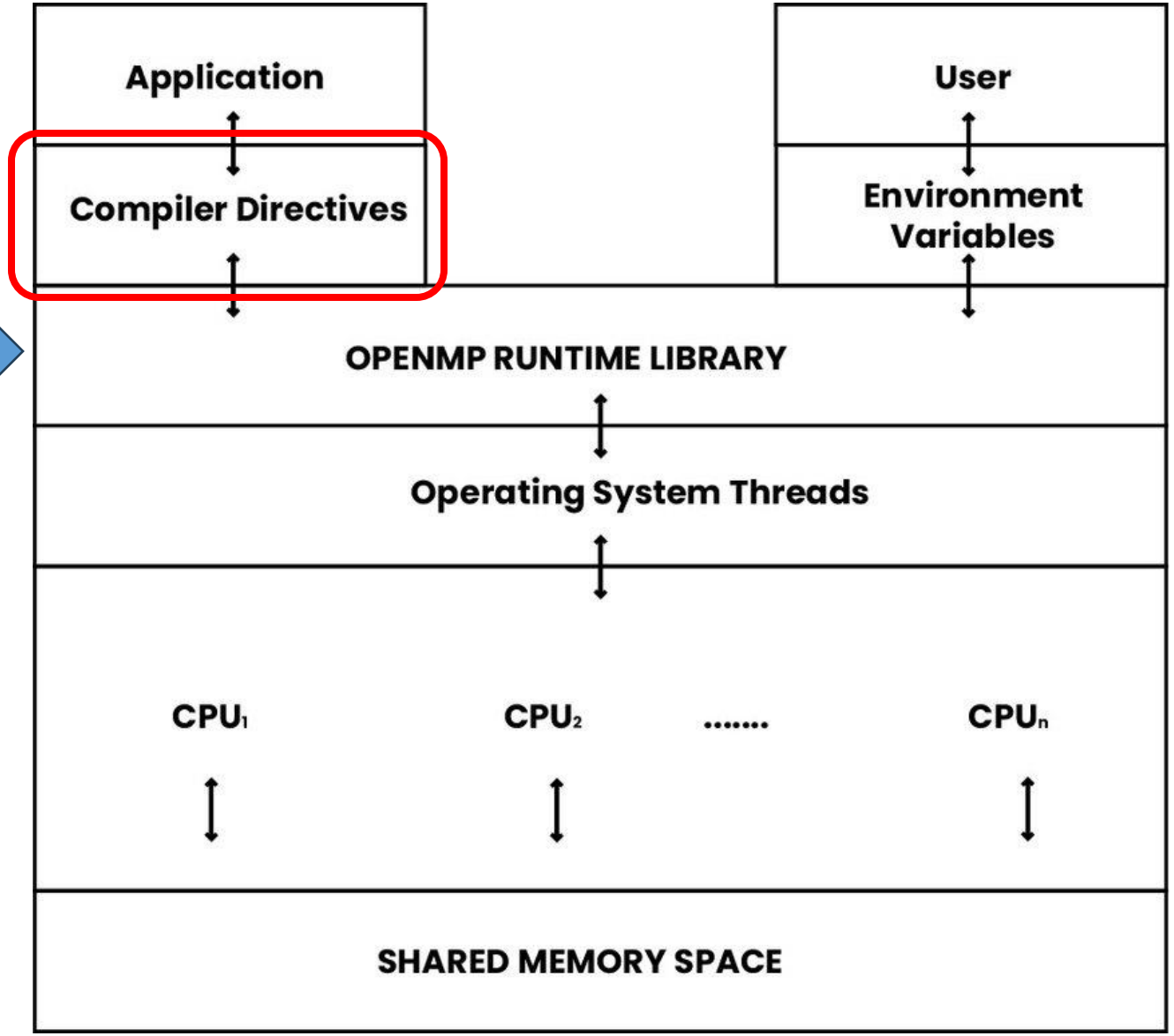
- Unlike functional-styled programming model like `std::thread`, OpenMP provides *compiler directives* that instruct the compiler to insert parallel code
  - aka *directive-based programming model* – counting on compiler to generate parallel code without explicitly managing threads which can sometimes be very massy

- OpenMP has two major advantages
  - Ease of use: Just tell compiler what and how to parallelize your code
  - Portability: It's compiler's job to insert platform-dependent parallel code!

https://www.openmp.org/

Software Architecture of OpenMP

Compiler-generated parallel code



History of OpenMP

- Misson statement: standardize directive-based high-level parallelism that is performant, productive and portable across different platforms

- Current spec is 5.2
  - Released in November 2021, http://www.openmp.org
  - More than 600 Pages

- Not all compilers are equally up-to-date with the OpenMP standard
  - GNU compiler supports up to OpenMP 4.5 , but does not fully support GPU offloading
  - LLVM is the new kind in town, good support, up to features of 5.0 (partial features)
  - Microsoft compiler lagging behind (Version 2.0 as of Jan 2021)

Example: Hello World in OpenMP

```
#include <iostream>
#include <omp.h>
int main() {
  #pragma omp parallel num_threads(4)
  {
    int me = omp_get_thread_num();
    int NT = omp_get_num_threads();

    std::cout << "Hello World. I'm thread " << me << " out of " << NT << ".\n";

    for (int i = 0; i < 2; i++)
      std::cout << "Iter:" << i << "\n";
  }
  std::cout << "All done here..." << std::endl;
}
```

All OpenMP directives start with "#pragma" telling the compiler to generate parallel code based on the following instructions

```
$ g++ test.cpp -fopenmp
$ ./a.out
Hello World. I'm thread 0 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 2 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 3 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 1 out of 4.
Iter:0
Iter:1
All done here...
```

OMP parallel region is executed by four threads

- Each thread executes the parallel region
- This type of programming approach is referred to *single-source multiple-execution* model

Example: Hello World in C++ Thread (for Comparison Purpose)

```cpp
#include <iostream>
#include <thread>
int main() {
  std::thread threads[4];

  for (int t = 0; t < 4; t++) {
    threads[t] = std::thread([me=t, NT=4]({

      std::cout << "Hello World. I'm thread " << me << " out of " << NT << ".\n";

      for (int i = 0; i < 2; i++) {
        std::cout << "Iter:" << i << "\n";
      }
    }));
  }

  for (int t = 0; t < 4; t++) {
    threads[t].join();
  }
  std::cout << "All done here..." << std::endl;
}
```

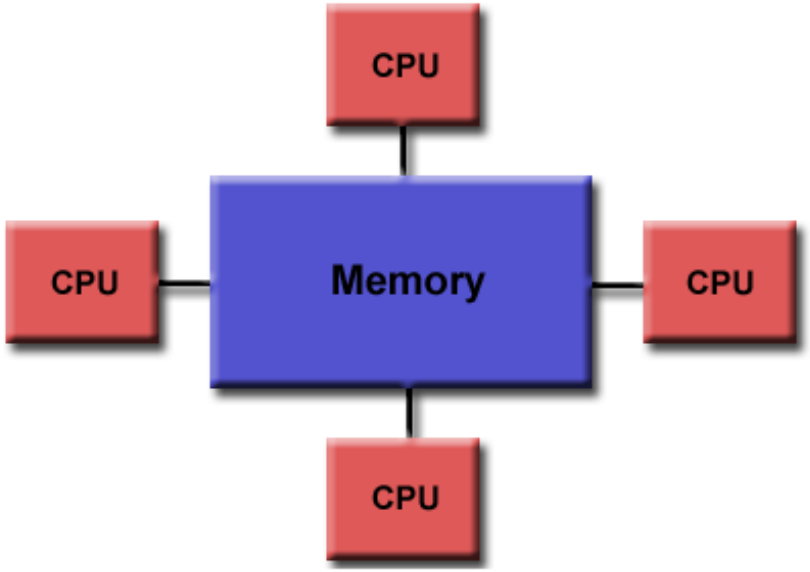
Explicit thread management, creating threads, assigning a starting function (lambda here), and joining created threads

```
$ g++ test.cpp
$ ./a.out
Hello World. I'm thread 0 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 2 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 3 out of 4.
Iter:0
Iter:1
Hello World. I'm thread 1 out of 4.
Iter:0
Iter:1
All done here...
```

When to Use OpenMP?

- OpenMP: targets parallelism on a shared-memory architecture
  - Help you program *thread-level parallelism (TLP)* on multi-core CPUs without explicitly managing threads like using `std::thread`
    - You know, programming `std::thread` can be very tedious
  - Suitable for simple, commonly used parallel patterns

Shared memory architecture



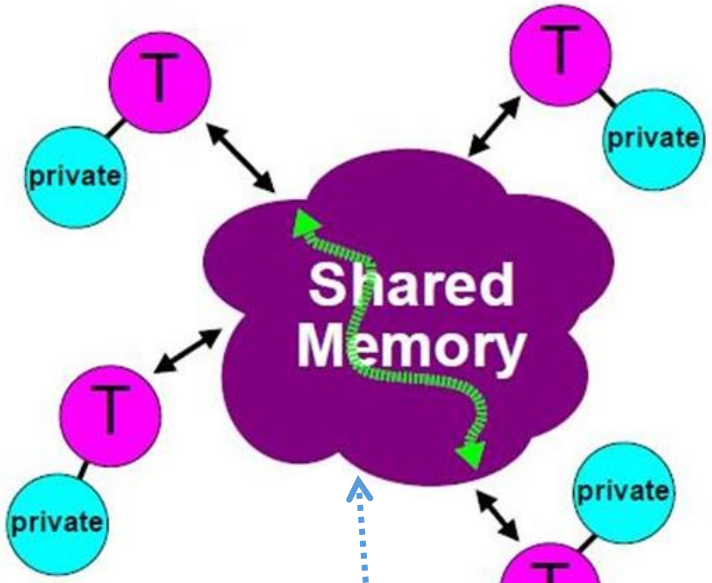
- OpenMP + CUDA: targets parallelism on the GPU
  - Modern OpenMP supports heterogeneous execution using GPU
  - Support only limited GPU-parallel primitives

- OpenMP + MPI: targets parallelism on a cluster (distributed computing)
  - MPI handles communication among machines
  - OpenMP handles thread-level parallelism on a machine

When to NOT Use OpenMP

- Fine-grained parallelism
  - For tasks with very fine-grained parallelism, the overhead of creating and managing threads might outweigh its benefits
- Complex synchronization
  - If your application requires intricate synchronization patterns, OpenMP's built-in synchronization primitives might not be sufficient (e.g., OpenMP does not have things like `std::condition_variable`)
- Dynamic parallelism pattern
  - For algorithms that depends on a lot of runtime variables to generate parallelism, OpenMP's directives may not be sufficient as the compiler has very limited information about what will happen at runtime

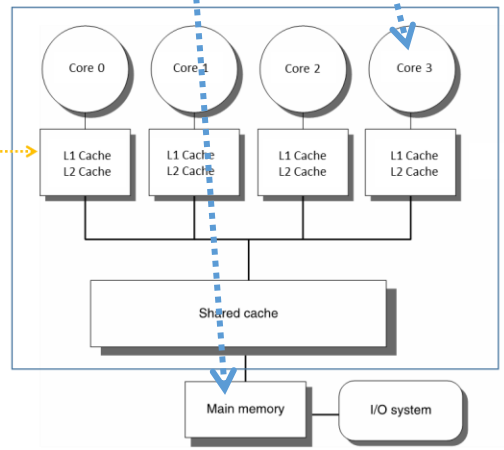
OpenMP Attributes [important slide]



This "Shared Memory" is the "Main memory"

This thread T runs here

L1 and L2 are per-core (data hazards: RAW, etc.)



- Threads execute different units of work simultaneously
- Threads have private data inaccessible by other threads
- Threads have access to a pool of memory shared by all threads
- How data is accessed by threads is transparent to programmer
- Synchronization rules are defined by OpenMP

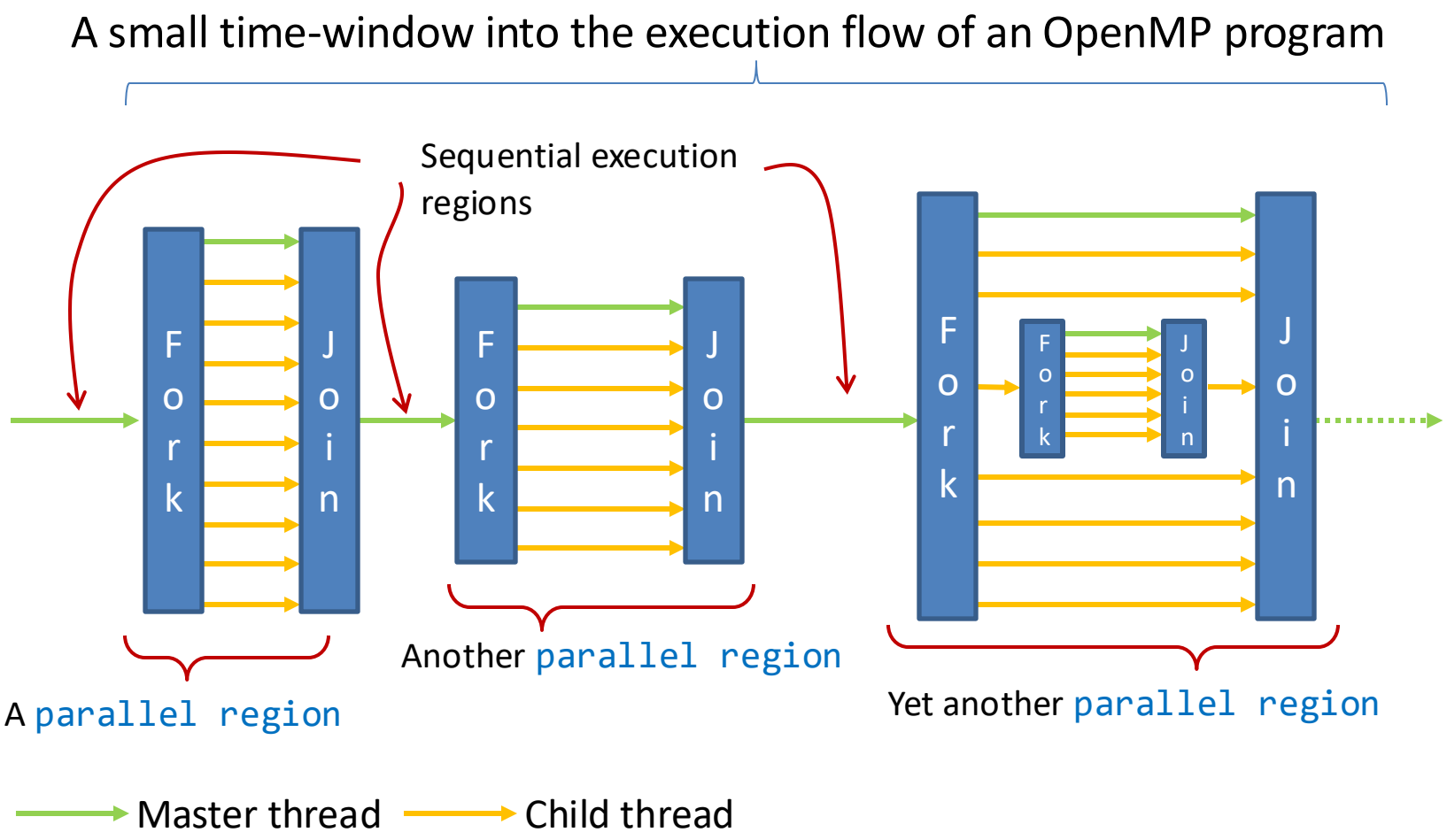
[CodeProject]→

OpenMP is Based on the Fork-Join Model

- Master thread spawns a team of child threads based on the number of threads you request
  - Master thread will be part of the team member (Note: This is different from `std::thread`)
  - When work is done, all forked threads join together automatically (no explicit `std::thread::join()`)

- You can ask for many threads in your OpenMP program, but it doesn't help you run faster if:
  - You go above the number of physical cores on your CPU
    - Ex: when you ask for more than 2x number of physical cores, typically your code slows down as the operating system gets busy servicing your many threads – *oversubscription*
  - The parallelizable portion of your program can only support limited parallelism
    - Ex: two independent tasks won't benefit from running them using six threads

OpenMP Execution Model



Compiling OpenMP Programs Using the Command Line

- Compiling OpenMP programs is easy as it has been integrated into mainstream compilers
- GCC:

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

- Clang

```
$ clang++ -o integrate_omp integrate_omp.cpp -fopenmp
```

- ICC:

```
$ icpc -o integrate_omp integrate_omp.cpp -openmp
```

- MSVC:

```
$ cl /openmp integrate_omp.cpp
```

Example: Calculate Entries in a Table Sequentially

- Version 1: Calculate the `std::sin` values at different input n from `0` to `255`

```
#include <omp.h>
constexpr auto PIE = 3.14159265358979323846;

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    for (int n = 0; n<size; ++n)
        sinTable[n] = std::sin(2 * PIE * n / size);
    // the table is now initialized
}
```

Example: Calculate Entries in a Table in Parallel w/ OpenMP

- Version 2: Calculate these 256 different `std::sin` values in parallel using OpenMP

```
#include <omp.h>
constexpr auto PIE = 3.14159265358979323846;

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    #pragma omp parallel for
    for (int n = 0; n<size; ++n)
        sinTable[n] = std::sin(2 * PIE * n / size);
    // the table is now initialized
}
```

[http://bisqwit.iki.fi/story/howto/openmp/]→

Example: Calculate Entries in a Table using SIMD w/ OpenMP

- Version 3: Uses wide registers & vector operations to enable SIMD parallelism

```
#include <omp.h>
constexpr auto PIE = 3.14159265358979323846;

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    #pragma omp simd
    for (int n = 0; n<size; ++n)
        sinTable[n] = std::sin(2 * PIE * n / size);
    // the table is now initialized
}
```

[http://bisqwit.iki.fi/story/howto/openmp/]→

Example: Calculate Entries in a Table using GPU w/ OpenMP

- Version 4: Offloading code to a GPU (not supported by all compilers)
  - The **sin** function must also exist on the target device

```
#include <omp.h>
constexpr auto PIE = 3.14159265358979323846;
int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    #pragma omp target teams distribute parallel for map(from:sinTable[0:256])
    for (int n = 0; n<size; ++n)
        sinTable[n] = std::sin(2 * PIE * n / size);
    // the table is now initialized
}
```

  - Note: Not all platforms support this feature yet although it's pretty cool

[http://bisqwit.iki.fi/story/howto/openmp/]→

A Few Syntax Details to Start with OpenMP

- Picking up the OpenMP header file

  `#include <omp.h>` (for C/C++)

- Most OpenMP constructs are compiler directives
  - C and C++: the directives start with the keyword `pragma`:

    `#pragma omp construct [clause [clause]…]`

  - Construct describes the parallelism type you want to perform
  - Clause further qualifies a directive's behavior

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

int main() {
#pragma omp parallel
    {
        int myId = omp_get_thread_num();
        int nThreads = omp_get_num_threads();

        std::cout << "Hello World. I'm thread " << myId
                  << " out of " << nThreads << ".\n";

        for( int i = 0; i < 2; i++ )
            std::cout << "Iter:" << i << "\n";
    }
    std::cout << "All done here..." << std::endl;
}
```

[IOMPP]→

Why Compiler Directives?

- One can have the same code, with no modifications, run with or without OpenMP
  - It is compiler's job to figure out how to generate the parallel code based on the provided directives
- When OpenMP is disabled, all pragma statements will be treated as comments
  - No need to maintain separate codebases between sequential and parallel code
- Directives are picked up by the compiler only if instructed to do so
  - Example: Visual Studio – you have to have the `/openmp` flag on

Commonly used Constructs in OpenMP Compiler Directives

Directive	Description
atomic	Specifies that a memory location will be updated atomically
barrier	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier
critical	Specifies that code is only executed on one thread at a time
flush	Enforces that all threads have the same view of memory for all shared objects
for	Causes the work done in a for loop inside a parallel region to be divided among threads
master	Specifies that only the master thread should execute a section of the program
ordered	Specifies that code under a parallelized for loop should be executed like a sequential loop
parallel	Defines a parallel region, which is code that will be executed by multiple threads in parallel
sections	Identifies code sections to be divided among all threads
single	Indicates that a section of code should be executed on a single thread, not necessarily the master thread
threadprivate	Specifies that a variable is private to a thread

[MSDN]→

OpenMP Beyond Directives: User-Level Runtime Routines

- In addition to directives, OpenMP provide user-level API to modify OpenMP parameters:
  - Modify/check the number of threads

```
omp_get_max_threads()          # get the max number of threads
omp_[set|get]_num_threads()    # get the number of threads
omp_get_thread_num()           # get the id of this thread
```

  - Are we in a parallel region?

```
omp_in_parallel()
```

  - How many processors in the system?

```
omp_get_num_procs()
```

  - Explicit locks

```
omp_[set|unset]_lock()
```

  - Many more…

[IOMPP]→

Example: Set/Get Number of Threads in a Parallel Region

```cpp
// omp_get_num_threads.cpp
// compile with: /openmp or -fopenmp
#include <iostream>
#include <omp.h>

int main() {
    std::cout << "Non parallel block, beginning of test: " << omp_get_num_threads() << "\n";
    omp_set_num_threads(2); // set the maximum number of threads to two
    std::cout << "Non parallel block, after omp_set_num_threads call: " << omp_get_num_threads() << "\n";

#pragma omp parallel
#pragma omp master
    {
        std::cout << "Inside a parallel block: " << omp_get_num_threads() << "\n";
    }

    std::cout << "No parallel block here: " << omp_get_num_threads() << std::endl;

    // changed the number of threads to be used inside parallel block
    // using a compiler directive
#pragma omp parallel num_threads(3)
#pragma omp master
    {
        std::cout << "Second parallel block: " << omp_get_num_threads() << "\n";
    }

    std::cout << "Outside parallel block: " << omp_get_num_threads() << std::endl;
}
```

```
$ g++ test.cpp -fopenmp
$ ./a.out
Non parallel block, beginning of test: 1
Non parallel block, after omp_set_num_threads call: 1
Inside a parallel block: 2
No parallel block here: 1
Second parallel block: 3
Outside parallel block: 1
```

Example: Get the Maximum Number of Threads

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

int main() {
    //omp_set_num_threads(5);
    std::printf("I can go w/ this many threads:%d\n", omp_get_max_threads());
#pragma omp parallel
#pragma omp master
    {
        std::printf("Here's how many threads I use in this parallel region: %d\n", omp_get_num_threads());
    }

#pragma omp parallel num_threads(3)
#pragma omp master
    {
        std::printf("Max. number of threads: %d\n", omp_get_max_threads());
        std::printf("Actual number of threads used in this other parallel region: %d\n", omp_get_num_threads());
    }

    std::printf("Here's the max number of threads at end:%d\n", omp_get_max_threads());

    return 0;
}
```

Query the maximum number of threads that can be used to form a new team when encountering a parallel region

```
$ g++ -std=c++17 test.cpp -fopenmp
$ ./a.out
I can go w/ this many threads:8
Here's how many threads I use in this parallel region: 8
Max. number of threads: 8
Actual number of threads used in this other parallel region: 3
Here's the max number of threads at end:8
```

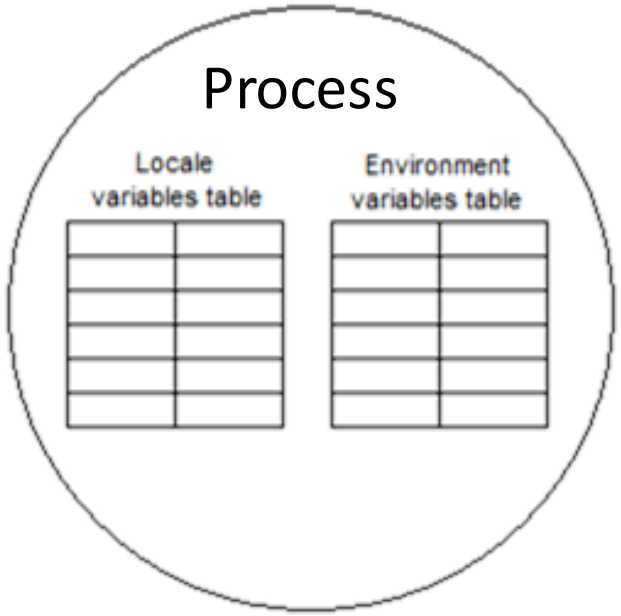
Got this on my Surface Studio laptop

```
$ g++ -std=c++17 test.cpp -fopenmp
$ ./a.out
I can go w/ this many threads:5
Here's how many threads I use in this parallel region: 5
Max. number of threads: 5
Actual number of threads used in this other parallel region: 3
Here's the max number of threads at end:5
```

Got this after uncommenting the first line in `main()` function

Modify OpenMP Parameters using Environment Variables

- Environment Variables are *dynamic named values* that can affect a running processes
  - When operating system spawns a process to run a program, it creates a list of environment variables that store process-specific information in a key-value format



Key	Description of Value
PATH	(:)-separated list of directories in which your system looks for executable files
USER	Logged-in user name
HOME	Path to the logged-in user home directory
UID	Logged-in user ID
SHELL	The current logged-in shell
…	…

  - C++ std::getenv and std::setenv allow you to control environment variables in the current process

Two Identical Ways to Modify OpenMP Runtime Parameters

- In OpenMP, you can use both function call and environment variables to modify parameters
  - Example 1: controlling the number of threads – the following two methods are equivalent
    - `omp_set_num_treads(8)`
    - For the Bash shell: **`export OMP_NUM_THREADS=8`**
  - Example 2: controlling nested parallelism – the following two methods are equivalent
    - `omp_set_nested(1);`
    - export OMP_NESTED=1
- Both are interchangeable, but OpenMP function call trumps the environment variable setting
  - `omp_set_num_treads(8)` **`export OMP_NUM_THREADS=12`**

    → The OpenMP program will use 8 threads

Commonly Used OpenMP Environment Variables

OMP_CANCELLATION:	Set whether cancellation is activated
OMP_DISPLAY_ENV:	Show OpenMP version and environment variables
OMP_DEFAULT_DEVICE:	Set the device used in target regions
OMP_DYNAMIC:	Dynamic adjustment of threads
OMP_MAX_ACTIVE_LEVELS:	Set the maximum number of nested parallel regions
OMP_MAX_TASK_PRIORITY:	Set the maximum task priority value
OMP_NESTED:	Nested parallel regions
OMP_NUM_THREADS:	Specifies the number of threads to use
OMP_PROC_BIND:	Whether theads may be moved between CPUs
OMP_PLACES:	Specifies on which CPUs the threads should be placed
OMP_STACKSIZE:	Set default thread stack size
OMP_SCHEDULE:	How threads are scheduled
OMP_THREAD_LIMIT:	Set the maximum number of threads
OMP_WAIT_POLICY:	How waiting threads are handled

OpenMP Programming is Supported by the Three Pillars

- Pillar 1: Compiler directives
  - Instruct compiler what and how to insert parallel code
  - A compiler that doesn't speak OpenMP simply ignores pragmas
    - Code runs just like before; i.e., sequentially
  - Directives have *clauses*, to further qualify a directive's behavior
- Pillar 2: User-level runtime function calls
  - Change OpenMP runtime behaviors programmatically
- Pillar 3: Environment variables
  - Another way of controlling OpenMP behavior without modifying the source code
  - Provides some of the support that the OpenMP functions provide

OpenMP: Parallel For Construct

Parallel-for is the Most Widely used Construct in OpenMP

- Revisit: Calculate 256 different `std::sin` values in parallel using OpenMP `parallel for`

```
#include <omp.h>
constexpr auto PIE = 3.14159265358979323846;

int main()
{
    const int size = 256;
    double sinTable[size]; // sin table to be initialized

    #pragma omp parallel for
    for (int n = 0; n<size; ++n) {
        sinTable[n] = std::sin(2 * PIE * n / size);
    }
}
```

[http://bisqwit.iki.fi/story/howto/openmp/]→

What's the Scope of a Directive like `#pragma omp parallel` ?

- In a nutshell, the scope of an OpenMP directive is the structured block of code that it affects
  - A structured block is the code enclosed by an opening "{" and a closing "}" – the only allowed "branches" to go out of the block are `exit`- or `return`-styled calls
  - This typically extends from the directive itself to <u>the next immediate matching pair</u> of curly braces "{}"

A "structured block"

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
more: res[id] = do_big_job (id);
  if( not_conv(res[id]) )goto more;
}
printf ("I'm outside par. region!\n");
```

Not a "structured block"

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if ( conv (res[id]) ) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

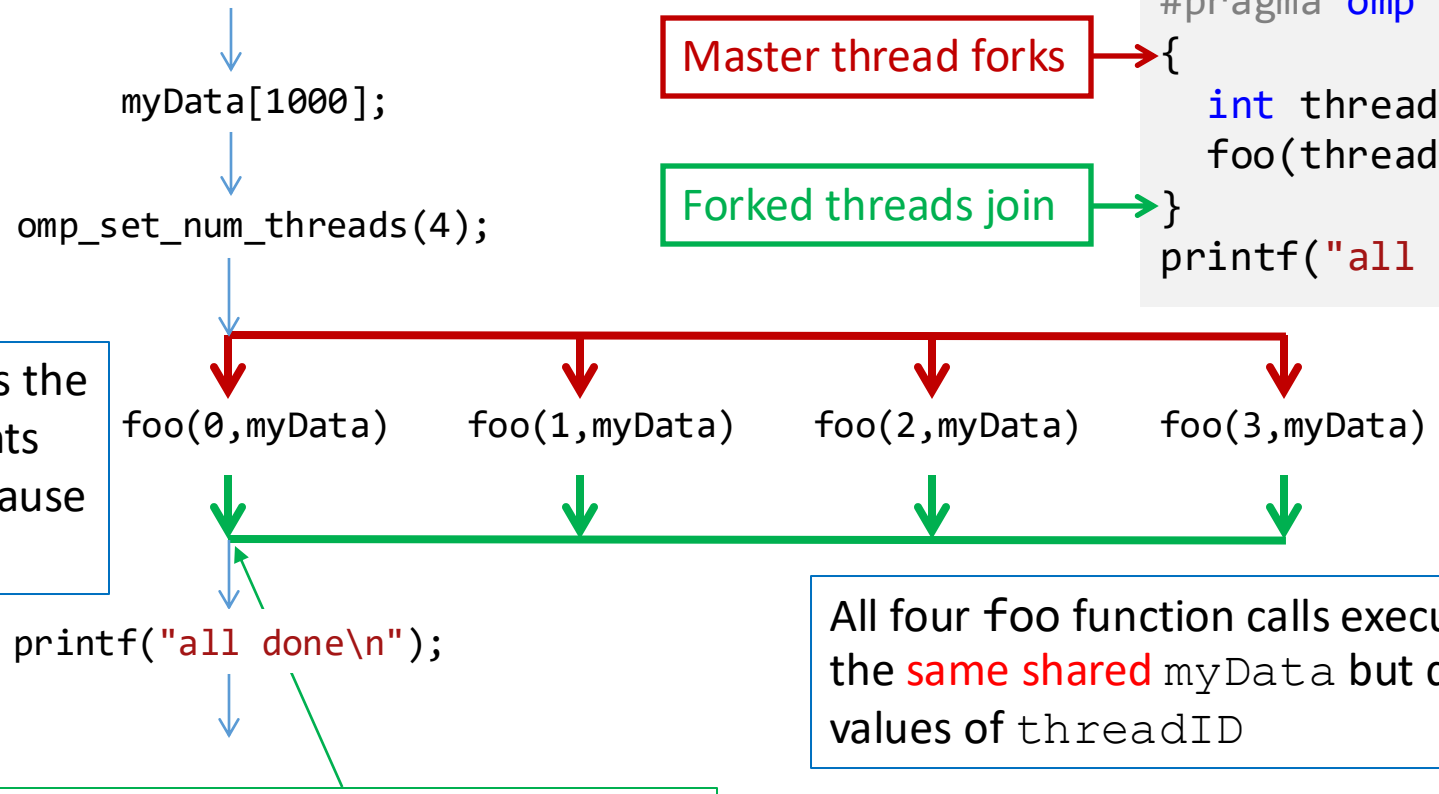
<u>IMPORTANT FACT</u>: There is an implicit barrier at the closing "}" where threads executing the block wait on each other to finish – the point at which forked threads will join.

Execution Diagram when Entering a Structured Block

```
double myData[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int threadID = omp_get_thread_num();
  foo(threadID, myData);
}
printf("all done\n");
```

Master thread forks

Forked threads join



Each thread executes the same code, arguments slightly different because of `threadID`

All four foo function calls executed with the same shared `myData` but different values of `threadID`

Implicit barrier; three of the threads are folded, after which master thread moves on

Example: Printing Messages from Multiple Threads

- The function `whatsUpQuestionMark` gets called by four parallel threads when the execution hits the `parallel` region

```
#include <omp.h>
#include <iostream>
void whatsUpQuestionMark()
{
  int me = omp_get_thread_num();
  std::cout << “What's up?, asks thread ” << me << “\n”;
}
int main()
{
  #pragma omp parallel num_threads(4)
  {
      whatsUpQuestionMark();
  }
  std::cout<< "all done...\n";
  return 0;
}
```

```
$ g++ test.cpp -fopenmp
$ ./a.out
What's up?, asks thread What's up?, asks thread What's up?, asks thread 03

1
What's up?, asks thread 2
all done...
```

- Messages get outputted in a garbled way
  - The insertion operator “<<” is a function of `std::cout`
  - Basically, it looks like the code below:

```
void whatsUpQuestionMark() {
  int me = omp_get_thread_num();
  printf("What's up?, asks thread ”);
  printf(“%d”, me);
  printf(“\n”, me);
}
```

  - Execution order of a function by different threads are non-deterministic

What Happens Under the Hood in terms of Generated Code?

```
#pragma omp parallel num_threads(4)
{
    whatsUpQuestionMark();
}
```

```
void wrapper_for_the_structured_block()
{
    whatsUpQuestionMark();
}
```

```
std::thread threads[3];

for (int i = 0; i < 3; ++i) //note that i starts at 1
  threads[i] = std::thread(wrapper_for_the_structured_block);

wrapper_for_the_structured_block();

for (int i = 0; i < 3; ++i) //three threads folded; see Remarks below
  threads[i].join();
```

- The OMP compiler generates some other code that takes care of the construct above
- When you say `num_threads(4)`, only ***three*** threads are created because the caller thread will be part of the team
- Since creating threads comes with certain cost, the OMP runtime uses a thread pool to reuse threads

Note: The threads are not always created and joined from the scratch (`std::thread` and `std::thread::join`) for a parallel region; they might be recycled back into a thread pool to reduce the cost of repetitively creating threads and joining threads in parallel regions

[Mattson, Breshears, Koniges, Kemp]→

A Short Side Trip: Timing an OpenMP Application

- `double omp_get_wtime()` – returns a value in *seconds* of the time elapsed from some timepoint
  - This timepoint is a consistent point in the past guaranteed not to change during execution of program
- `double omp_get_wtick()` – returns the number of seconds between clock ticks
  - This tick provides a higher resolution than wall time by directly measuring the duration in terms of clock ticks

```
#include <omp.h>
int main() {
    int const N = 100000;
    double dummy[N];
    double start = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        int temp = std::rand();
        dummy[i] = 2.*temp / (std::pow(temp*temp, 1.5) + 0.2);
    }
    double end = omp_get_wtime();
    std::printf("start = %.16g\n", start);
    std::printf("end   = %.16g\n", end);
    std::printf("diff  = %.16g\n", end - start);
    double wtick = omp_get_wtick();
    std::printf("wtick   = %.16g\n", wtick);
    std::printf("1/wtick = %.16g\n", 1.0 / wtick);
}
```

```
start = 1528033.489782439
end   = 1528033.510552168
diff  = 0.02076972858048975
wtick = 3.011764252346089e-07
1/wtick = 3320313
Press any key to continue . . .
```

Nested Parallelism

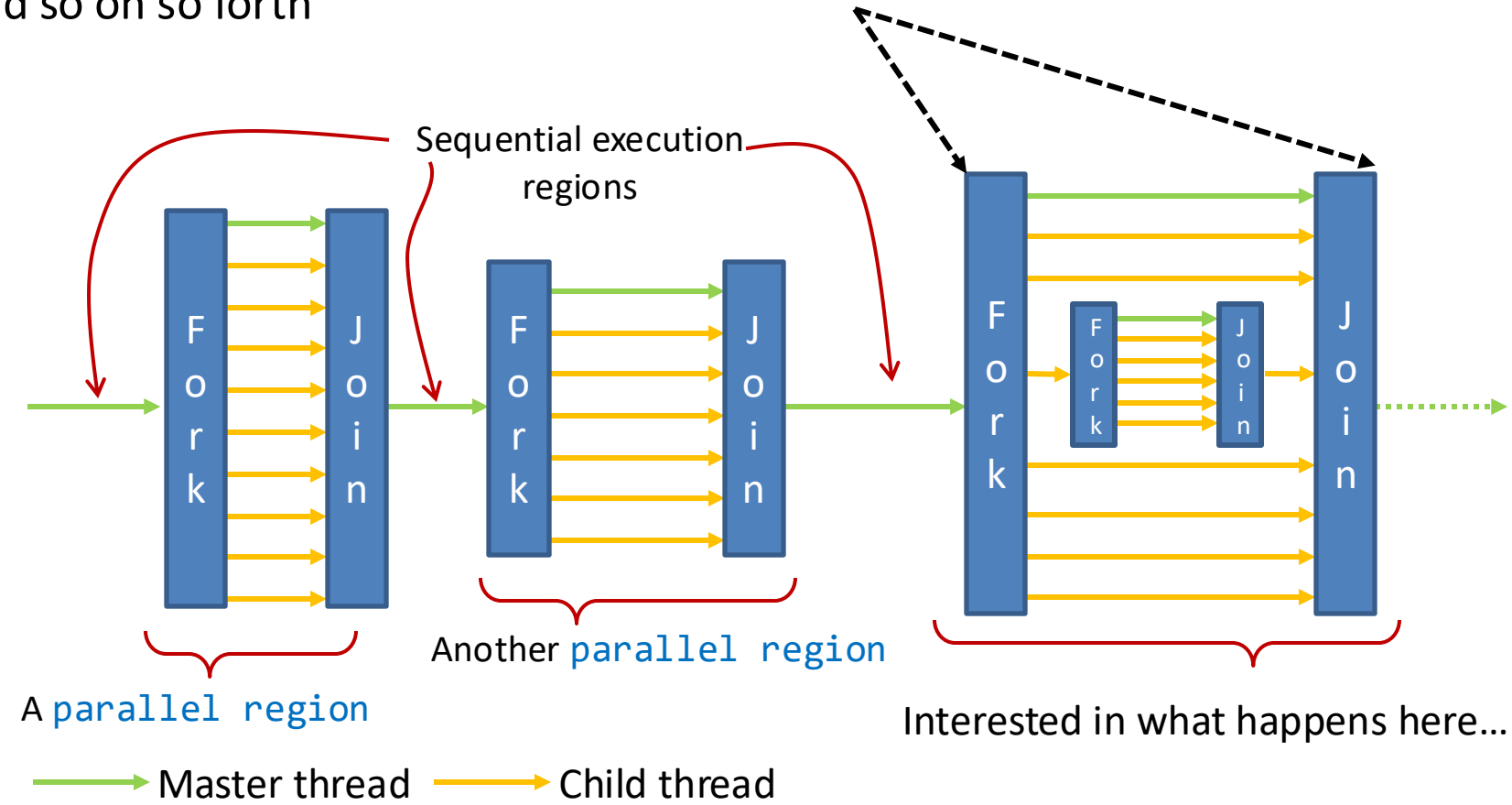
- Motivation: You have a parallel region that runs in parallel by, say four threads, and inside that parallel region, there is another code region that might also run in parallel
  - Ex: Multiplying two matrices requires looping through each row and column of the two matrices to calculate dot product

```
void MultiplyMatrices(int nCount, double **matrixA,
                      double **matrixB, double **matrixC)
{
    int i, j, k ;

    for (i = 0; i < nCount; i++)
    {
        for (j = 0; j < nCount; j++)
        {
            matrixC[i][j]=0;

            for (k = 0; k < nCount; k++)
            {
                matrixC[i][j] +=
                    matrixA[i][k]*matrixB[k][j];
            }
        }
    }
}
```

How can we parallelize nested loops using OpenMP?

OpenMP Supports Nested Parallelism

- OpenMP `parallel` construct can be defined in a nested fashion
  - When a thread *T* on a team of threads hits a parallel region, it spawns another teams of threads mastered by *T* and so on so forth



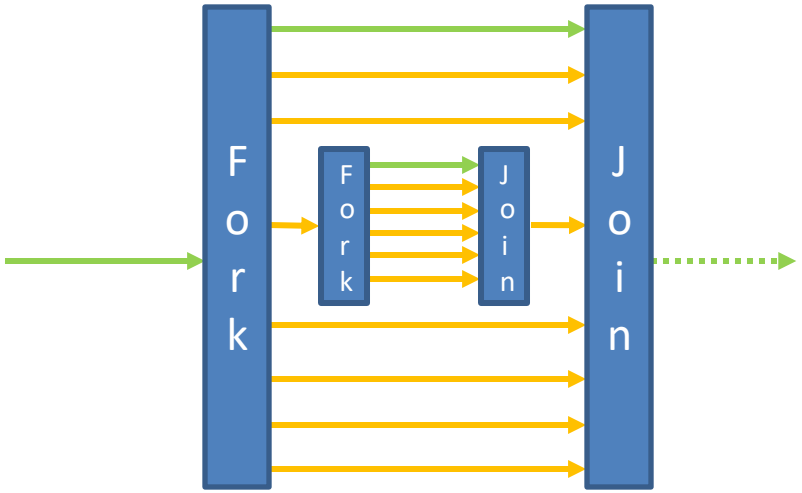
Nested Parallelism Revisited

- Nested parallelism is very easy to achieve using OpenMP – simply add another parallel-for
  - One to parallelize the first for loop of i
  - Another to parallelize the second for loop of j

```
void MultiplyMatrices(int nCount, double **matrixA,
                      double **matrixB, double **matrixC)
{
    int i, j, k ;

    for (i = 0; i < nCount; i++)
    {
        for (j = 0; j < nCount; j++)
        {
            matrixC[i][j]=0;

            for (k = 0; k < nCount; k++)
            {
                matrixC[i][j] +=
                    matrixA[i][k]*matrixB[k][j];
            }
        }
    }
}
```

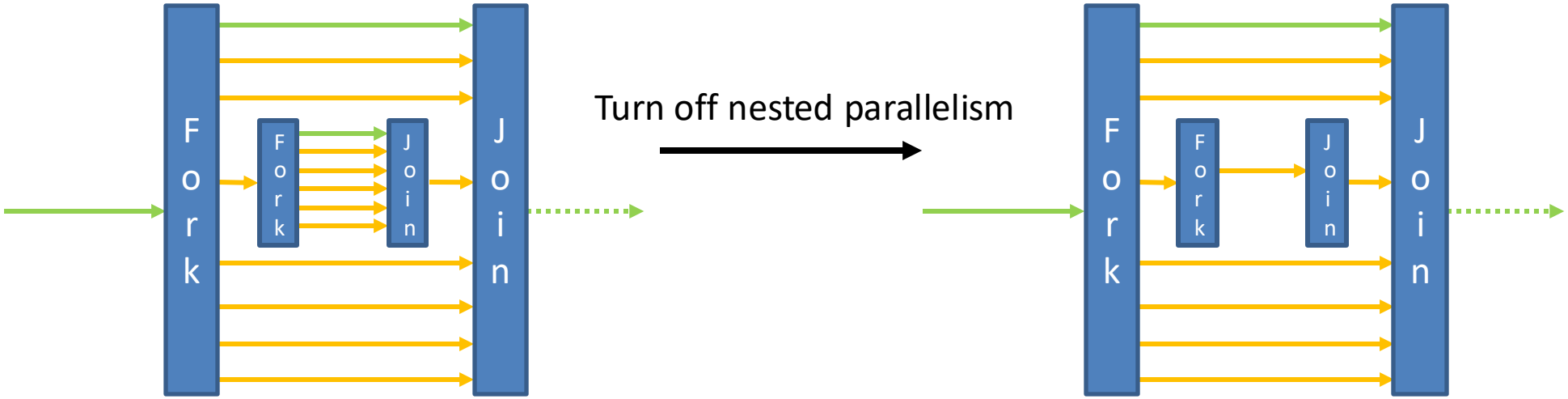
```
#pragma omp parallel
```

```
#pragma omp parallel
```



Nested Parallelism is Not a Silver Bullet …

- Nested parallelism may not always take effect as you wish
  - Why? Nested parallelism is hard to implement efficiently – depends a lot of application workloads
    - Some parts of the code with nested parallelism is ok
    - Some other parts of the code are better off without nested parallelism
- API to control nested parallelism behavior: `omp_set_nested()`
  - Alternatively, you can use the OMP_NESTED environment variable to globally turn on/off nested parallelism
  - When you turn off nested parallelism, a parallel thread encountering a new parallel region will not be able to spawn new parallel threads – that new parallel region will be executed by thread *T* only



Control the Behavior of Nested Parallelism

- Relevant API elements
  - `omp_set_nested(…)`
  - `omp_get_nested(…)`

  Enables or disables nested parallelism

  - `omp_set_dynamic(…)`
  - `omp_get_dynamic(…)`

  Control the number of threads in nested parallel regions
  - **If argument is nonzero, the runtime library can adjust the number of threads**
  - **If argument is zero, the runtime library cannot dynamically adjust the number of threads**

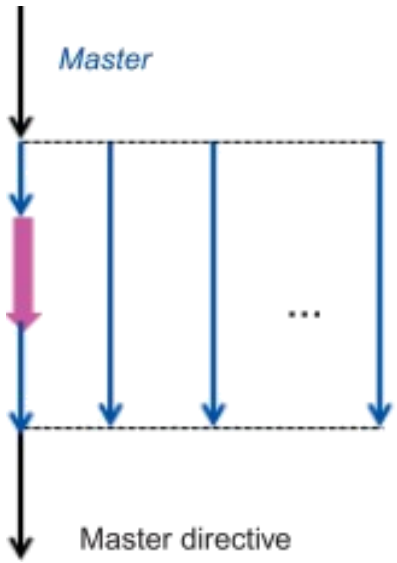
  How the runtime decides to adjust the number of threads, it's implementation specific, might change from vendor to vendor.

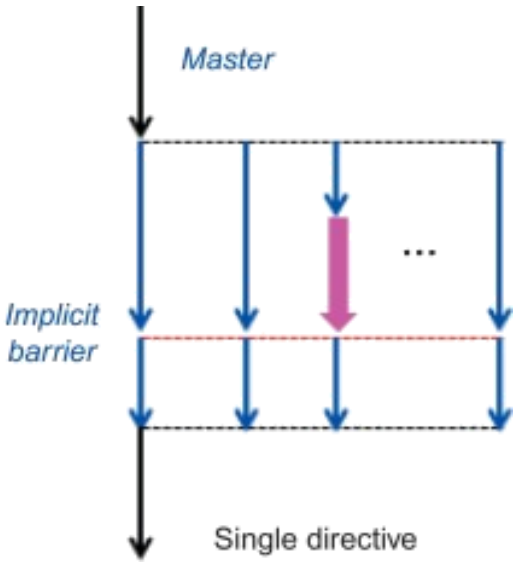
- There are equivalent environment variables doing the same thing for nested parallelism
  - Example: if you use in bash `export OMP_NUM_THREADS="4,2"`, you will get outer level parallelism to run with 4 threads, while the next level of nested parallelism to run with 2 threads. Then, you have 8 threads active: 4 X 2.

The OpenMP `single` directive

- `single` directive defines a section of code to run by only a single thread in a parallel region
  - Similar to `master` that defines a section of code to run by only the master thread, `single` allows any thread to run the section of code whichever thread reaches the region first

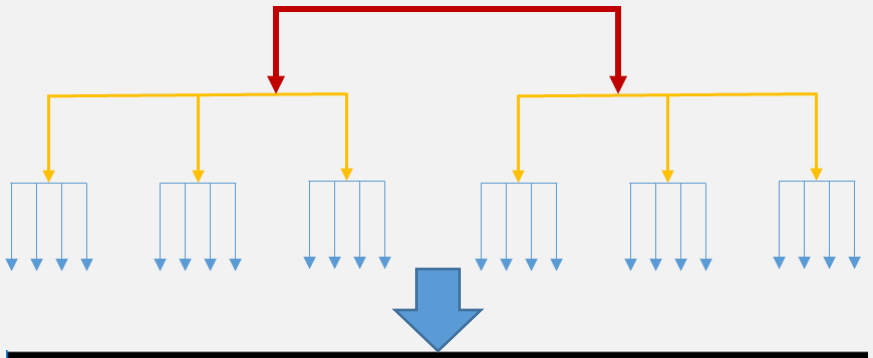
- Synchronization behavior is a bit different …
  - `single` directive has an implicit barrier upon completion of the region, where all threads wait for synchronization
  - `master` directive doesn't have the barrier. The other threads move and do their business





Example: Running a Function Once in a Nested Parallel Region

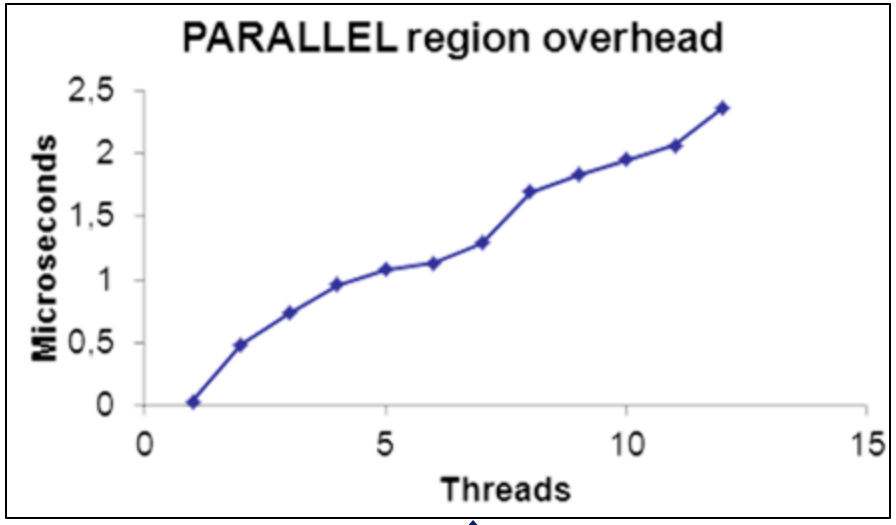
```c
#include <omp.h>

// function to print level by just one thread
void report_num_threads(int whichLevel) {
    #pragma omp single
    printf("Level %d: number of threads in the team - %d\n", whichLevel, omp_get_num_threads());
}
int main() {
    omp_set_dynamic(1);
    omp_set_nested(1);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(3)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(4)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```



```
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 3
Level 2: number of threads in the team - 3
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
Level 3: number of threads in the team - 4
```

Departing Thoughts about Nested Parallelism

- Examples of good reasons to employ nested parallelism
  - Insufficient parallelism at outer level
    - Example: an outer loop of 10 trips on a 64-core machine
  - Load balance scenario:
    - Many-core processors today can handle many threads
      - IBM POWER9 chip: handles 96 threads
    - More threads in flight can help balance the load



- In my experience, it's better to avoid nested parallelism
  - Overhead of nested parallelism often outweighs its advantages
    - Scheduling, synchronization, hierarchical concurrency, etc. in a nested region comes with non-negligible cost!
  - Better to redesign your algorithm in a flat one-dimensional loop with just one-level `omp parallel for`

[Argonne Natnl. Lab]→

Quiz: Remove Nested Parallelism

- Consider the following parallel loops with two-level nested parallelism:
  - `run_some_independent_function` is totally independent of `i` and `j`

```
#pragma omp parallel for
for (int i = 0; i < N; i++) {
  #pragma omp parallel for
  for (int j = 0; j < M; j++) {
    some_independent_func(i, j);
  }
}
```

```
#pragma omp parallel for
for (int k = 0; k < N*M; k++) {
   some_independent_func(k/M, k%M);
}
```

- Parallelize the code using just one level of `omp parallel for`

Summary: What's Reasonable to Expect from OpenMP

- If you have more than N cores in an SMP setup, it's unlikely that you get a N-times speed-up

- Some of the reasons for lack of scaling:
  - Amdahl Law states that the maximum speed-up is limited to the portion that can be parallelized
  - Cache coherence (multiple caches can have their private copies for the same data)
  - False cache sharing (data too close to each other will cause cache access to be serialized)

- Execution scales somewhat better if you have an embarrassingly-parallel application
  - No dependencies exist at all! Just throw as many threads as possible supported by your machine!
  - Unfortunately, real-world parallel applications all have different types of dependencies