CPSC 4240/5240: Parallel Programming Techniques

Lecture 5: OpenMP

Lecturer: Quanquan C. Liu

Spring 2026

Participation from Lecture Quizzes

- Take quiz within 3 days of lecture
- Lecture slides posted on Canvas, use URL to get the quiz and read the slides
- Quiz only during weeks with homework assignment

Homework 1 Clarifications

- Answer the questions as stated in the written assignment
- When turning in code; always turn in the most efficient code you can obtain using techniques from lecture
- There are no “rules” on what techniques you can use in the code, as long as it outputs the correct answer; in this case, the correct output C and F matrices

OpenMP

- You used this for Homework 1 but only the parallel for loop
- Today we'll learn more about OpenMP so you can use it in your coding projects
- OpenMP is a higher level interface for **programming with threads**
- Parallelization via **compiler directives** written as source code annotations
- Caveat: I generally do not recommend programming with threads but sometimes it is the easiest thing to try first
- But: better than explicit threads programming

Why OpenMP instead of C++ threads

Example OpenMP vs. Explicit Thread Programming

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

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Parallelize the loop using OpenMP
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    arr[i] *= multiplier;
}
```

- Takes an array of numbers arr and multiplies each element by 5
- You've seen this before in the homework
- At compile-time run the loop in parallel
- "Safe" because each thread only accesses and writes to one element in arr

Example OpenMP vs. Explicit Thread Programming

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

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Parallelize the loop using OpenMP
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    arr[i] *= multiplier;
}
```

```
#include <thread>

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
const int num_threads = 4;

// Calculate chunk size (basic static partitioning)
int chunk_size = (n + num_threads - 1) / num_threads;

// Vector to hold the threads
std::vector<std::thread> threads;
threads.reserve(num_threads);

// Spawn threads
for (int t = 0; t < num_threads; ++t) {
        threads.emplace_back([&, t]() {
        // Determine the start and end for this thread
        int start = t * chunk_size;
        int end   = std::min(start + chunk_size, n);

        for (int i = start; i < end; ++i) {
            arr[i] *= multiplier;
         }
    });
}

// Join threads
for (auto &thr : threads) {
    thr.join();
}
```

Example OpenMP vs. Explicit Thread Programming

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

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Parallelize the loop using OpenMP
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    arr[i] *= multiplier;
}
```

Needs many thread management procedures!

```
#include <thread>

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
const int num_threads = 4;

// Calculate chunk size (basic static partitioning)
int chunk_size = (n + num_threads - 1) / num_threads;

// Vector to hold the threads
std::vector<std::thread> threads;
threads.reserve(num_threads);

// Spawn threads
for (int t = 0; t < num_threads; ++t) {
      threads.emplace_back([&, t]() {
      // Determine the start and end for this thread
      int start = t * chunk_size;
      int end   = std::min(start + chunk_size, n);

      for (int i = start; i < end; ++i) {
          arr[i] *= multiplier;
       }
   });
}

// Join threads
for (auto &thr : threads) {
   thr.join();
}
```

Example OpenMP vs. Explicit Thread Programming

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

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

// Parallelize the loop using OpenMP
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    arr[i] *= multiplier;
}
```

May be required for concurrent programming but not parallel!

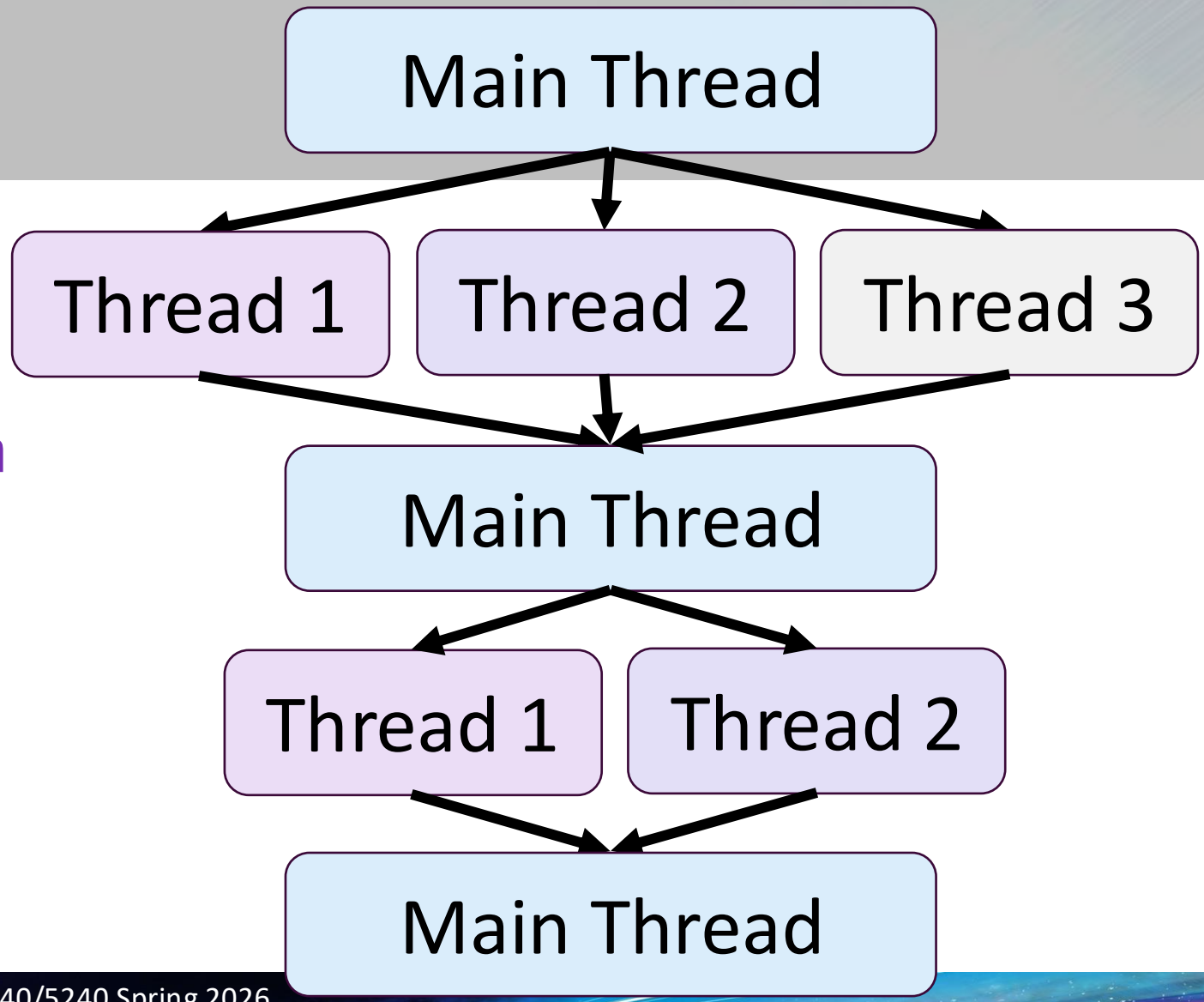
```
#include <thread>

int arr[n] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
const int num_threads = 4;

// Calculate chunk size (basic static partitioning)
int chunk_size = (n + num_threads - 1) / num_threads;

// Vector to hold the threads
std::vector<std::thread> threads;
threads.reserve(num_threads);

// Spawn threads
for (int t = 0; t < num_threads; ++t) {
      threads.emplace_back([&, t]() {
      // Determine the start and end for this thread
      int start = t * chunk_size;
      int end   = std::min(start + chunk_size, n);

      for (int i = start; i < end; ++i) {
          arr[i] *= multiplier;
       }
   });
}

// Join threads
for (auto &thr : threads) {
   thr.join();
}
```

OpenMP Fork/Join

Work-span can simulate this

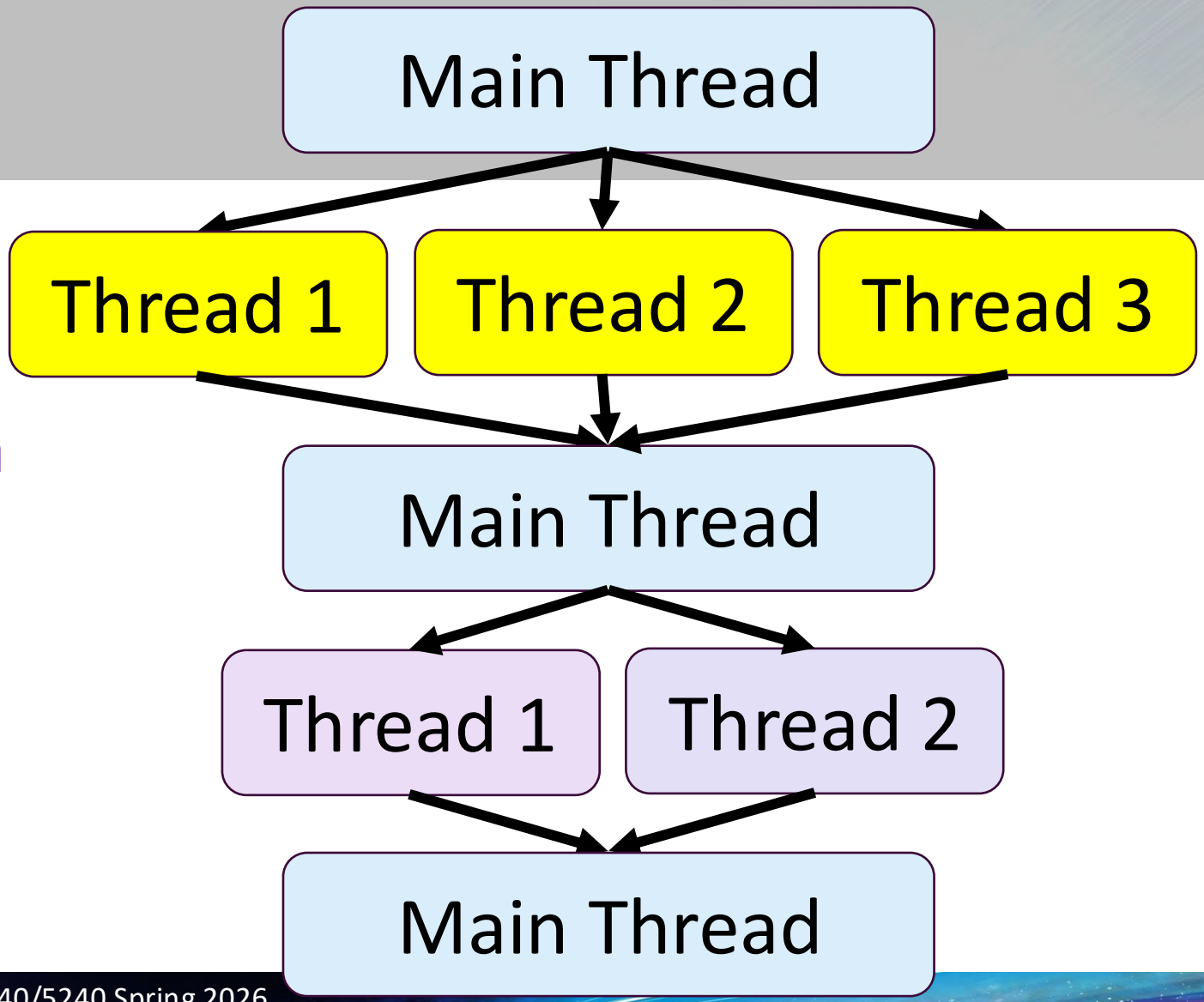
OpenMP's Fork-Join Model

- Work-span model we learned last two lectures can be mapped to this model
- A program begins as a **single thread**
- Enter a **region for parallelism** and spawns a number of other threads
- Can be called a parallelism **scope**
- Within each scope threads execute in parallel



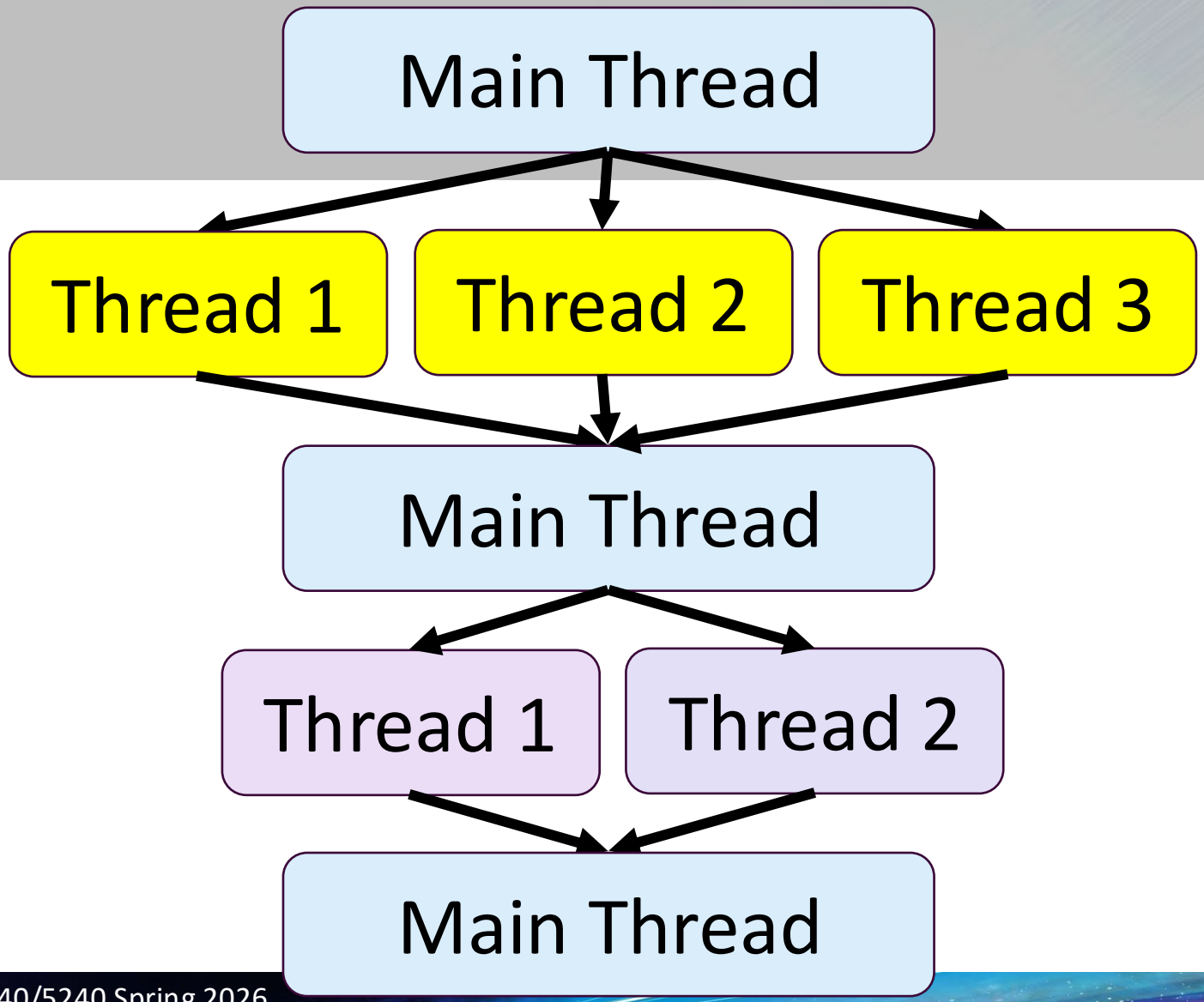
OpenMP's Fork-Join Model

- Work-span model we learned last two lectures can be mapped to this model
- A program begins as a **single thread**
- Enter a **region for parallelism** and spawns a number of other threads
- Can be called a parallelism **scope**
- Within each scope threads execute in parallel



OpenMP's Fork-Join Model

- Within each scope threads execute in parallel
- At the end of the scope:
  - Threads **synchronize at barrier** and are disbanded
- **Initial main thread continues**
- Teams of threads can be created, synchronized, and disbanded many times
  - Very costly!
  - Synchronization is costly!



OpenMP Syntax

- Compiler directives
  - #pragma omp [construct] [clause…]
  - #pragma omp
    - Signals to compiler to use omp to parallelize sections of your code
  - [construct]
    - Placeholder for specific OpenMP instructions (constructs)
      - Eg. parallel, for, sections, task, single
  - [clause…]
    - Additional modifiers or parameters: private(variable_list), shared(variable_list), reduction(operator: variable_list)…

Simple OpenMP Constructs

OpenMP Syntax Constructs

- Common OpenMP Constructs
  - #pragma omp parallel
    - Starts a **parallel region/scope**
  - #pragma omp for
    - Parallel for loop
  - #pragma omp sections
    - Divides code into separate sections, each executed by a different thread
  - #pragma omp task
    - Specifies a task that can be executed independently
  - #pragma omp single
    - Ensures a block of code is executed by **one thread**

OpenMP Syntax Constructs Examples

- #pragma omp parallel
  - Starts a **parallel region/scope**

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

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

int main() {
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        std::cout << "Hello from thread " <<
omp_get_thread_num() << std::endl;
    }
    return 0;
}
```

OpenMP Syntax Constructs Examples

- #pragma omp parallel
  - Starts a **parallel region/scope**

```
Hello from thread 0
Hello from thread 1
Hello from thread 2
Hello from thread 3
```

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        std::cout << "Hello from thread " <<
omp_get_thread_num() << std::endl;
    }
    return 0;
}
```

OpenMP Syntax Constructs Examples

- #pragma omp parallel
  - Starts a **parallel region/scope**

Set number of threads equal to either number of **physical cores** or number of **logical cores (number of hyperthreads)**

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        std::cout << "Hello from thread " <<
omp_get_thread_num() << std::endl;
    }
    return 0;
}
```

OpenMP Syntax Constructs Examples

- #pragma omp parallel for
  - Two equivalent definitions

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

int main() {
    #pragma omp parallel for
    for (int i = 1; I < N – 1; i++)
            a[i] = b[i-1] + b[i+1];
    return 0;
}
```

OpenMP Syntax Constructs Examples

- #pragma omp parallel for

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

int main() {
    #pragma omp parallel {
    #pragma omp for
    for (int i = 1; I < N – 1; i++)
            a[i] = b[i-1] + b[i+1];
    return 0;
}
```

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

int main() {
    #pragma omp parallel for
    for (int i = 1; I < N – 1; i++)
            a[i] = b[i-1] + b[i+1];
    return 0;
}
```

OpenMP Syntax Constructs Examples

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

int main() {
   #pragma omp parallel sections
   {
      #pragma omp section
      {
         std::cout << "Thread " << omp_get_thread_num() << " executing section 1" << std::endl;
      }
      #pragma omp section
      {
         std::cout << "Thread " << omp_get_thread_num() << " executing section 2" << std::endl;
      }
   }
   return 0;
}
```

OpenMP Syntax Constructs Examples

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

int main() {
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            std::cout << "Thread " << omp_get_thread_num() << " executing section 1" << std::endl;
        }
        #pragma omp section
        {
            std::cout << "Thread " << omp_get_thread_num() << " executing section 2" << std::endl;
        }
    }
    return 0;
}
```

```
Thread 0 executing section 1
Thread 1 executing section 2
```

OpenMP Syntax Constructs Examples

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

int main() {
   #pragma omp parallel
   {
      #pragma omp single
      {
         #pragma omp task
         std::cout << "Task 1 executed by thread " << omp_get_thread_num() << std::endl;

         #pragma omp task
         std::cout << "Task 2 executed by thread " << omp_get_thread_num() << std::endl;
      }
   }
   return 0;
}
```

OpenMP Syntax Constructs Examples

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task
            std::cout << "Task 1 executed by thread " << omp_get_thread_num() << std::endl;

            #pragma omp task
            std::cout << "Task 2 executed by thread " << omp_get_thread_num() << std::endl;
        }
    }
    return 0;
}
```

```
Task 1 executed by thread 0
Task 2 executed by thread 1
```

OpenMP Syntax Constructs Examples

Sections vs. Tasks: Static parallelism versus dynamic

Sections fix threads, tasks go into task pool, picked up by any free thread

OpenMP Syntax Constructs Examples

```
int main() {
   #pragma omp parallel
   {
      // Only one thread will create tasks, but ALL threads can execute them
      #pragma omp single
      {
         for (int i = 0; i < num_tasks; i++) {
            #pragma omp task
            {
               // Do some tasks
            }
         }
      } // implicit barrier for the single region
   } // implicit barrier for the parallel region

   return 0;
}
```

```
Task 0 started by thread 0
Task 1 started by thread 1
Task 2 started by thread 0
Task 1 finished by thread 1
Task 3 started by thread 1
Task 0 finished by thread 0
Task 2 finished by thread 0
Task 4 started by thread 0
Task 3 finished by thread 1
Task 4 finished by thread 0
```

OpenMP Syntax Constructs Examples

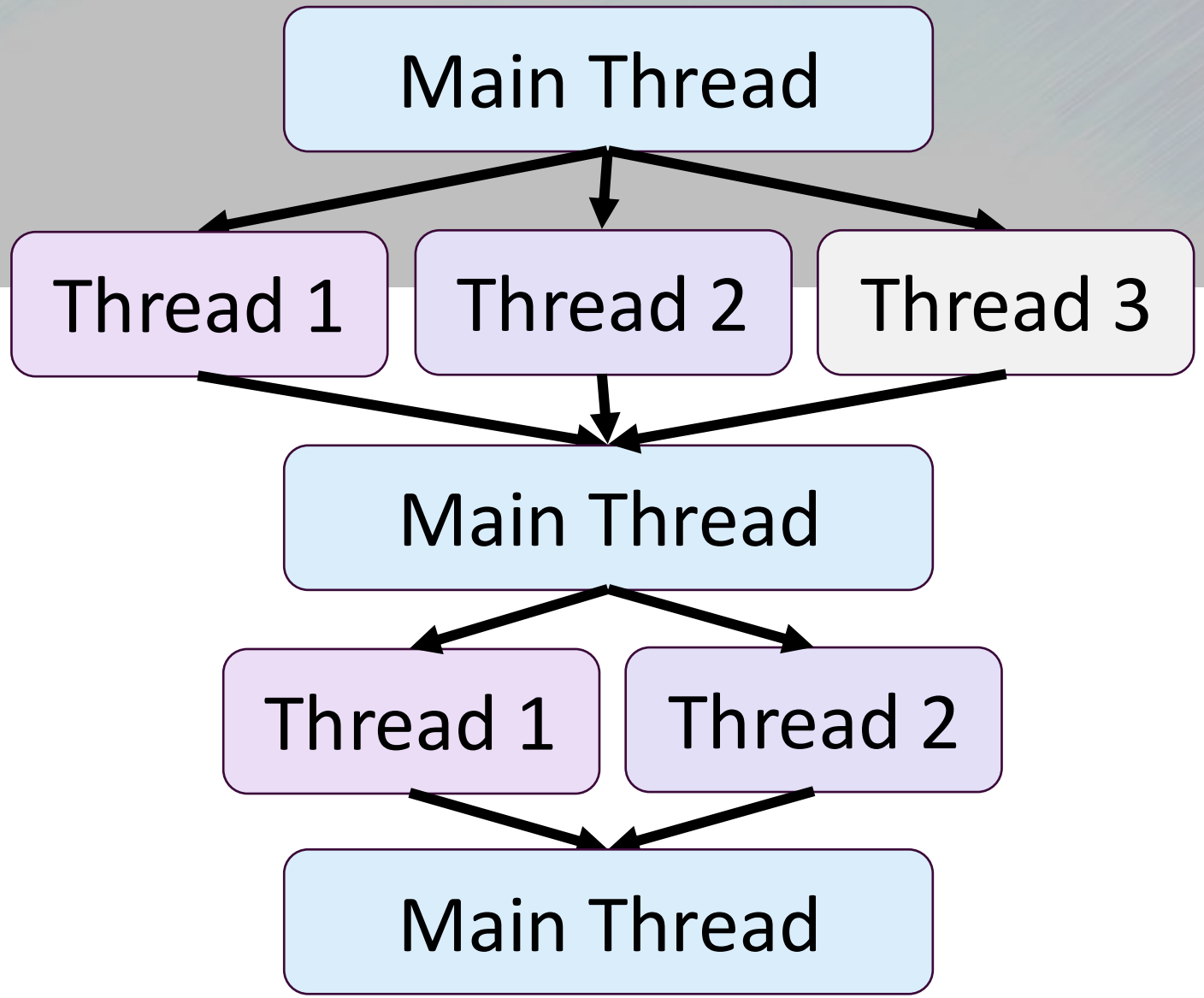
```
#include <omp.h>
#include <iostream>

int main() {
   #pragma omp parallel
   {
      #pragma omp single
      {
         std::cout << "This block is executed by thread "
                   << omp_get_thread_num() << std::endl;
      }
   }
   return 0;
}
```

OpenMP Syntax Constructs Examples

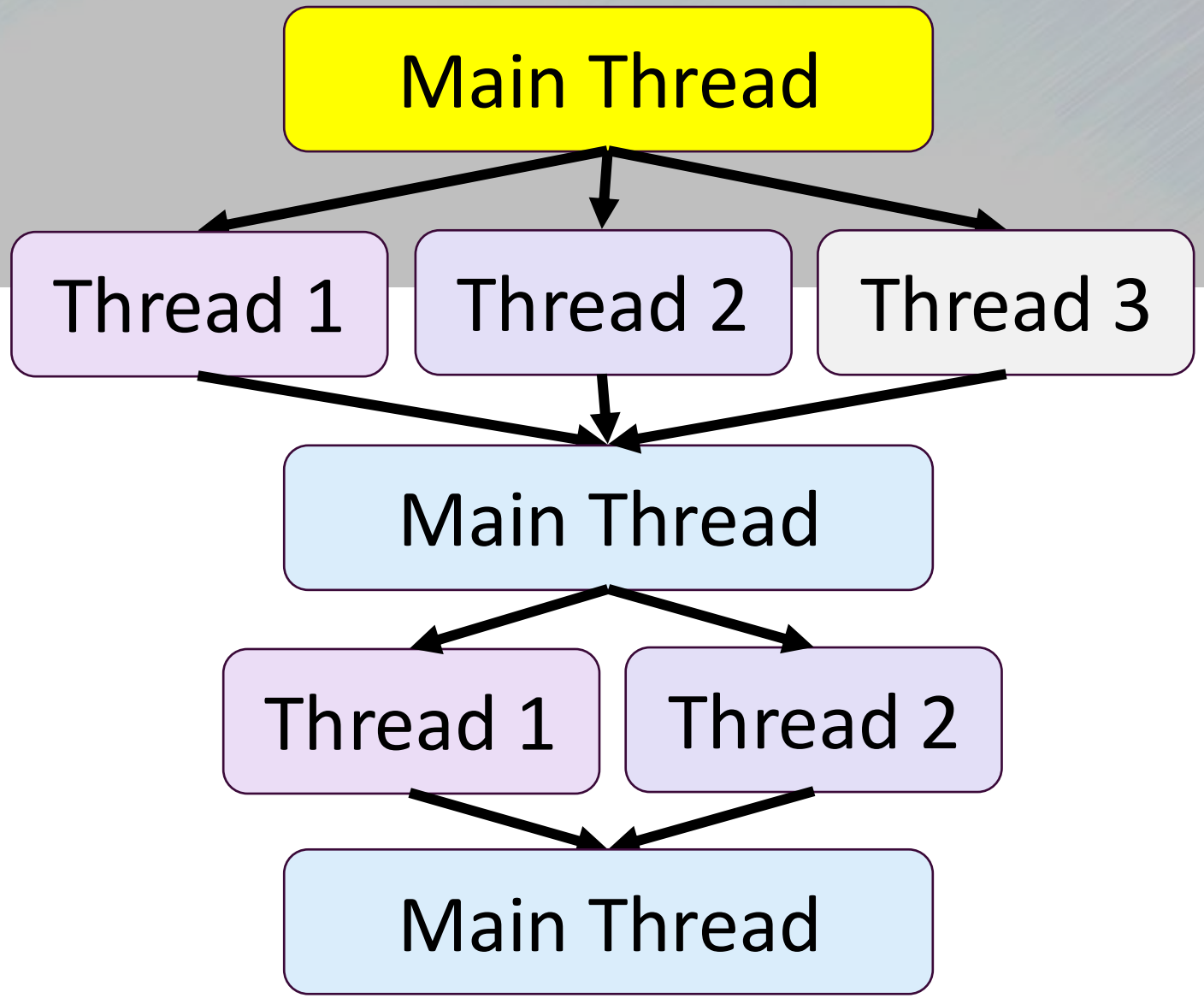
```
#include <omp.h>
#include <iostream>

int main() {
   #pragma omp parallel
   {
      #pragma omp single
      {
         std::cout << "This block is executed by thread "
                   << omp_get_thread_num() << std::endl;
      }
   }
   return 0;
}
```

This block is executed by thread 0

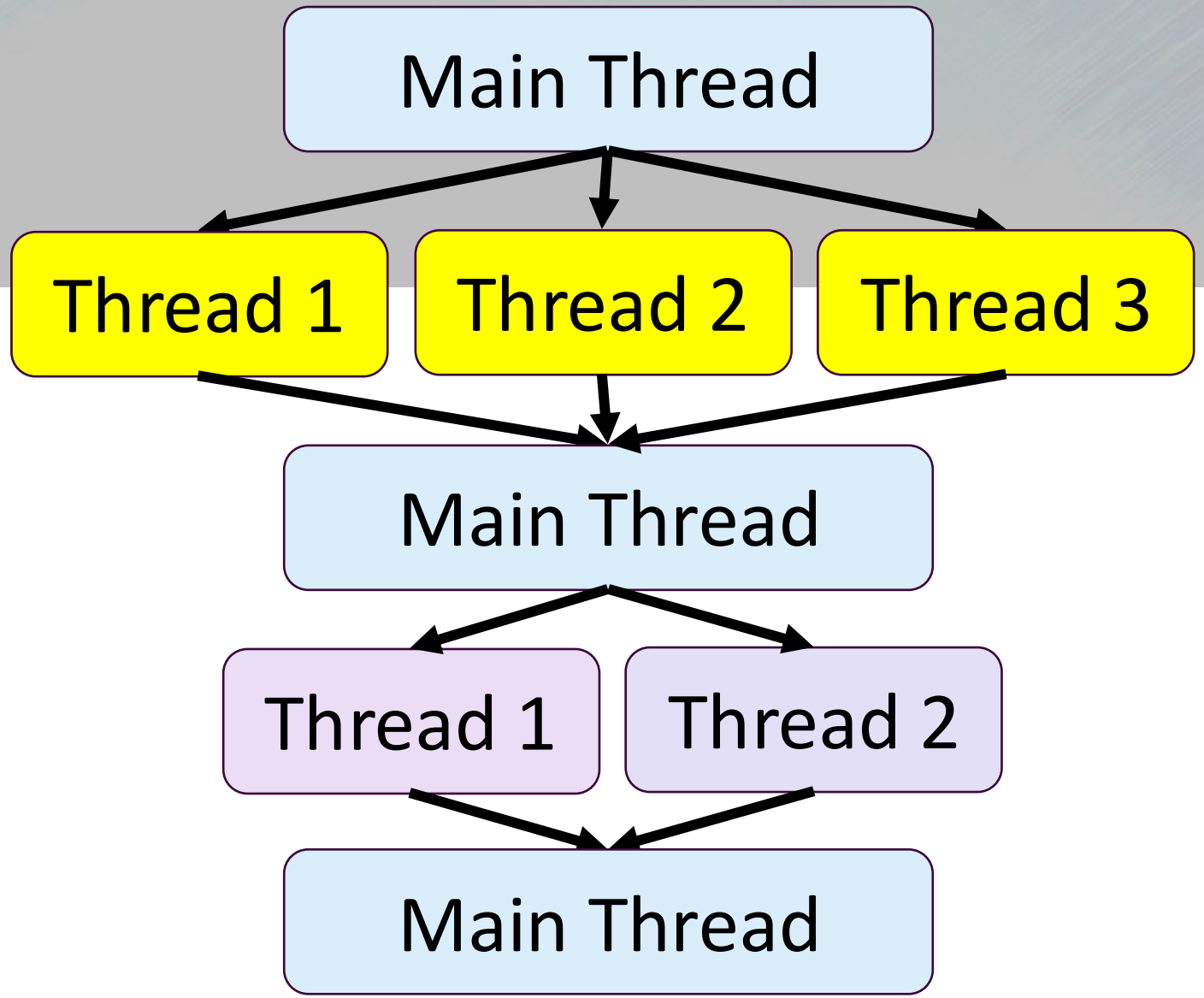
```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



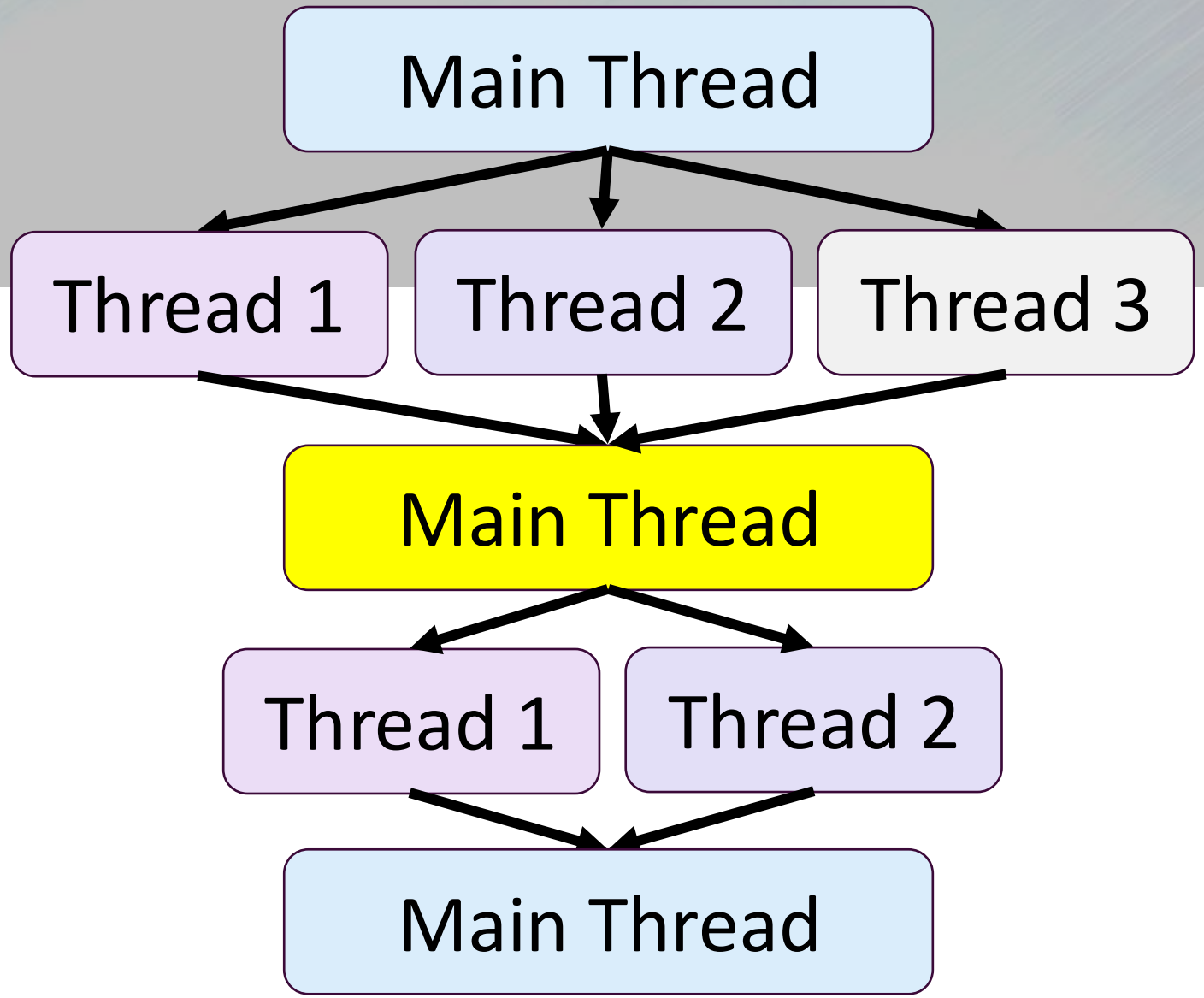
```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



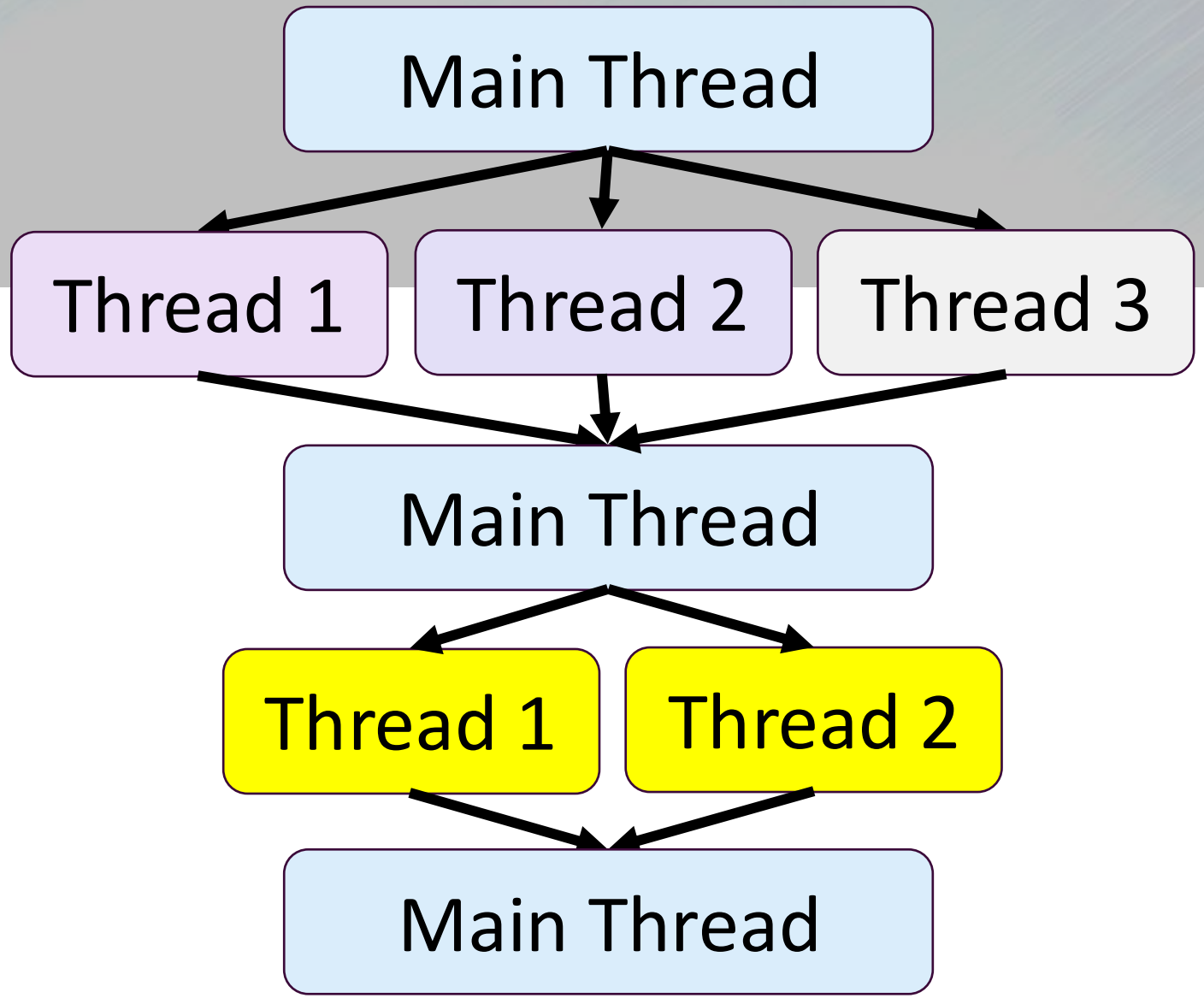
```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



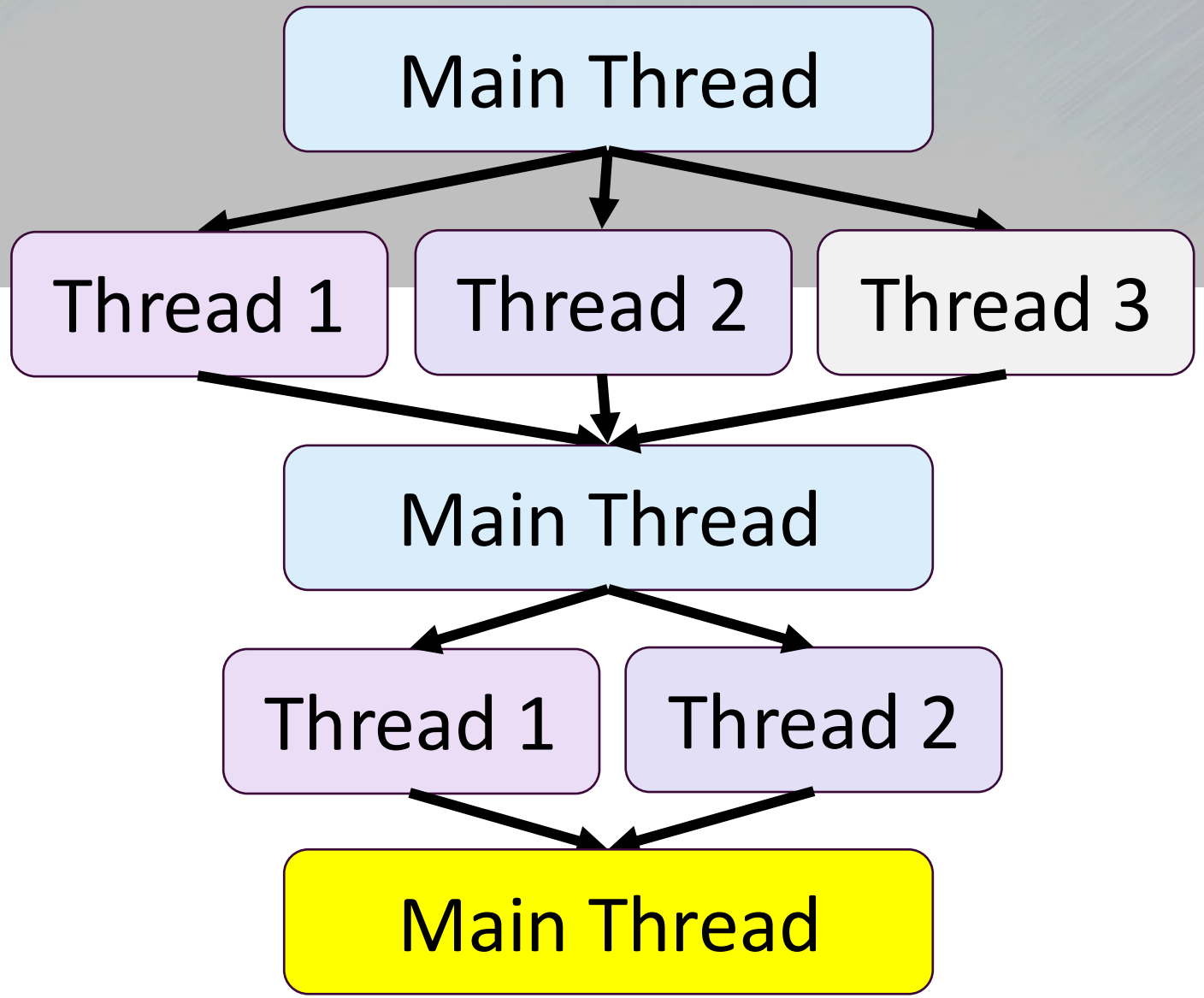
```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



```
cout << “Serial\n”;
N = 1000;
#pragma omp parallel{

#pragma omp for
for (i=0; i < N; i++)
        A[i] = B[i] + C[i];

#pragma omp single
M = A[N/2]

#pragma omp for
for (j = 0; j < M; j++)
        P[j] = Q[j] – R[j];
}
cout << “Finish\n”;
```



A Few More Details

- Parallel for:
  - Translator automatically generates appropriate threads
- Parallel
  - Inserts needed **barriers**
  - **Barriers** synchronize threads
  - {CODE}: {} braces responsible for initializing threads and creating synchronization barriers
  - Without this, we need to construct barriers manually, a pain to debug!

Variable Scoping

When to share variables between threads

Variable Scoping

- Any variables declared **outside parallel region shared by all threads** by default
- Variables declared **inside parallel region** are **private to that region**
- **Shared** and **private** clause declarations override defaults
- May want to explicit declare for documentation, readability, or bookkeeping purposes

Variable Scoping

- Why do we want variable scoping?
  - **Avoiding data races**
    - Multiple threads write to the same variable
    - Making private ensures each thread gets own copy
  - **Ensuring correct results**
    - Logically, some variables shouldn't be shared between multiple threads
    - Others should be shared
  - **Performance optimization**
    - Under the hood synchronization overhead from shared variables etc.

Variable Scoping Example

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

int main() {
    const int N = 5;
    std::vector<int> data(N, 0);
    int i;
    #pragma omp parallel for shared(data) private(i)
    for (i = 0; i < N; i++) {
        data[i] = i;  // Each thread writes to a distinct element of data
    }
    return 0;
}
```

Variable Scoping Example

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

int main() {
    const int N = 5;
    std::vector<int> data(N, 0);
    int i;
    #pragma omp parallel for shared(data) private(i)
    for (i = 0; i < N; i++) {
        data[i] = i;  // Each thread writes to a distinct element of data
    }
    return 0;
}
```

data vector is **shared among all threads** and the **index i** is **private to each thread—makes own copy**

Variable Scoping Example

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

int main() {
    const int N = 5;
    std::vector<int> data(N, 0);
    int i;
    #pragma omp parallel for shared(data) private(i)
    for (i = 0; i < N; i++) {
        data[i] = i;  // Each thread writes to a distinct element of data
    }
    return 0;
}
```

data vector is **shared among all threads** and the **index i** is **private to each thread—makes own copy**

Why is it necessary here? Very subtle!

Variable Scoping Example

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

int main() {
    const int N = 5;
    std::vector<int> data(N, 0);
    int i;
    #pragma omp parallel for
    for (i = 0; i < N; i++) {
        data[i] = i;  // Each thread writes to a distinct element of data
    }
    return 0;
}
```

data vector is **shared among all threads** and the **index i** is **private to each thread—makes own copy**

Why is it necessary here? Very subtle!

int i is a shared variable!

Dealing with Dependencies in Loops

- OpenMP charges ahead with parallelizing a loop when you tell it to even if doing so breaks correctness

```
std::vector<int> data(N, 0);
   int i;
   #pragma omp parallel for
   for (i = 0; i < N; i++) {
      data[i] = i;  // Each thread writes to a distinct element of data
   }
```

- Restructure the algorithm to make it correct
- Sometimes there's a warning but **not always! Only catches simple things!**

Why are Dependencies So Bad?

- Two threads writing to the same memory location causes **race conditions**
- Threads are “independent” within a parallel region and can **execute in any order**
- When threads are racing to modify the same memory address, we call this a **race condition**
- Depends on which threads execute first, can get **different results**
- Generally not what we want in programming (getting different results)!

Simple OpenMP Race Condition Example

What could happen here?

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

int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        // Race condition: no synchronization!
        sum += 1;
    }

    std::cout << "Final sum = " << sum << std::endl;
    return 0;
}
```

Simple OpenMP Race Condition Example

sum is a shared variable

Two threads can add to sum simultaneously

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

int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        // Race condition: no synchronization!
        sum += 1;
    }

    std::cout << "Final sum = " << sum << std::endl;
    return 0;
}
```

Simple OpenMP Race Condition Example

Two threads can add to sum simultaneously

Won't give 10000 with high likelihood

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

int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        // Race condition: no synchronization!
        sum += 1;
    }

    std::cout << "Final sum = " << sum << std::endl;
    return 0;
}
```

Simple OpenMP Race Condition Example

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

int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 10000; i++) {
        // Race condition: no synchronization!
        sum += 1;
    }

    std::cout << "Final sum = " << sum << std::endl;
    return 0;
}
```

Final sum = 9992
Final sum = 9975
Final sum = 9941
Final sum = 10000
Final sum = 9987
Final sum = 9823

How do we fix it?

- Use a **reduction clause**
  - #pragma omp parallel for reduction(+: sum)
  - reduction([operation]: [variable])
  - **Performs operation on variable safely**
  - Under the hood: uses a **parallel reduce primitive**
    - Parallel primitive algorithm like those we discussed last week
    - Calculates partial sums and aggregate them

How do we fix it?

- Use a **reduction clause**
  - #pragma omp parallel for reduction(+: sum)

```
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < 10000; i++) {
    sum += 1;
}
```

What is allowed in reduction?

Arithmetic operators: +, -, *

Bitwise operators: &, |, ^

Logical operators: &&, ||

min, max

How do we fix it?

- Use a **reduction clause**
  - #pragma omp parallel for reduction(+: sum)

```
#pragma omp parallel for reduction(+: sum)
for (int i = 0; i < 10000; i++) {
    sum += 1;
}
```

What variables are allowed in reduction?

scalar variables: int, float, double, bool

Custom reductions (OpenMP 4.0+)

Subranges (OpenMP 4.5+)

Another Way to Fix

- Use an **atomic variable**
  - #pragma omp atomic

```
#pragma omp parallel for
for (int i = 0; i < 10000; i++) {
    #pragma omp atomic
    sum += 1;
}
```

Another Way to Fix: Atomic Variables

- What are **atomic variables**?
- Atomic variables are variables that can be accessed (read, written modified) without interference from multiple threads
  - Prevents race conditions
  - Operations on atomic variables happen as a single, individual (atomic) operation
- Under the hood: generate a hardware atomic instruction (eg. LOCK XADD) or internal (software) lock if hardware lock isn't available
- Locks are expensive! Can lead to serialization due to contention

Another Way to Fix: Atomic Variables

- What are **atomic variables**?
- Atomic variables are variables that ca
  modified) without interference from m
  - Prevents race conditions
  - Operations on atomic variables happen as a single, individual (atomic) operation
- Under the hood: generate a hardware atomic instruction (eg. LOCK XADD) or internal (software) lock if hardware lock isn't available
- Locks are expensive! Can lead to serialization due to contention

Use to (quickly) ensure safety when you have infrequent updates to the same variable!

Lecture 5 Quiz