CSCI 5451: Introduction to Parallel Computing

Lecture 8: OpenMP in Depth

computer science
& engineering

UNIVERSITY OF MINNESOTA
Driven to Discover®

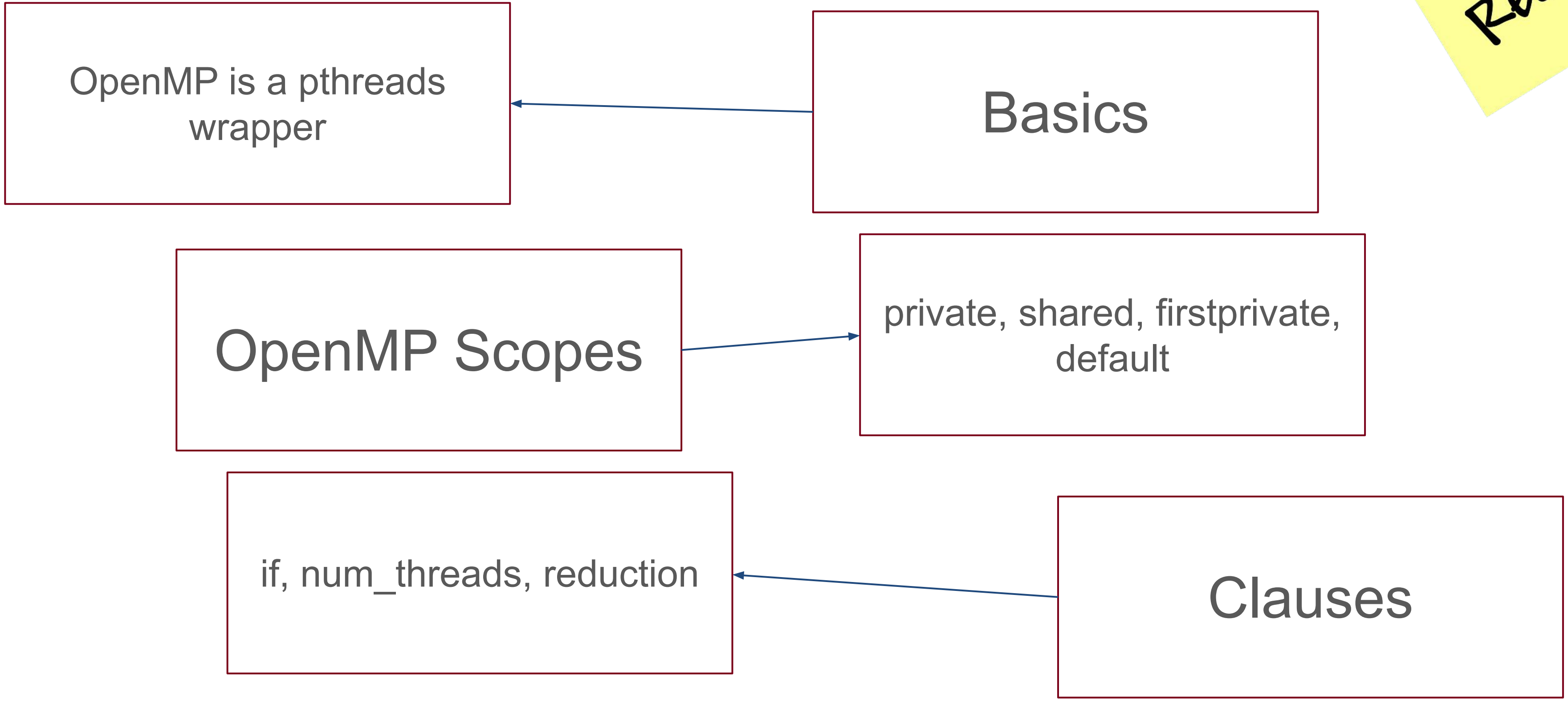
Announcements (9/29)

- HW1
  - Released later today (until we have completed OpenMP in better detail for the assignment)
  - Due Oct 12
  - In-Depth Session on Wednesday covering the homework (this will not be standard for homeworks going forward)

Lecture Overview

- ❑ **Recap**
- ❑ Compiling & Running an OpenMP program
- ❑ 'parallel' Directive in Detail
  - False Sharing Examples
  - Setting the Number of Threads
  - Additional Directives
  - threadprivate scope
- ❑ 'for' directive in Detail
  - Scheduling
  - Nesting

Recap (Threads)


OpenMP is a pthreads wrapper
Basics
OpenMP Scopes
private, shared, firstprivate, default
if, num_threads, reduction
Clauses

Recap (Threads)

```
/*  ****************************************************
   An OpenMP version of a threaded program to compute PI.
    **************************************************** */

    #pragma omp parallel default(private) shared (npoints) \
                         reduction(+: sum) num_threads(8)
    {
      num_threads = omp_get_num_threads();
      sample_points_per_thread = npoints / num_threads;
     sum = 0;
      for (i = 0; i < sample_points_per_thread; i++) {
         rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
         rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
        if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
             (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
             sum ++;
      }
    }
```

Lecture Overview

- ❑ Recap
- ❑ **Compiling & Running an OpenMP program**
- ❑ 'parallel' Directive in Detail
    - o False Sharing Examples
    - o Setting the Number of Threads
    - o Additional Directives
    - o threadprivate scope
- ❑ 'for' directive in Detail
    - o Scheduling
    - o Nesting

Compiling & Running an OpenMP program

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        printf("Hello from thread %d out of %d\n", thread_id, total_threads);
    }
    return 0;
}
```

```
>> gcc -O3 -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 out of 4\n
Hello from thread 2 out of 4\n
Hello from thread 3 out of 4\n
Hello from thread 1 out of 4\n
```

Compiling & Running an OpenMP program

Make sure to include omp header

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        printf("Hello from thread %d out of %d\n", thread_id, total_threads);
    }
    return 0;
}
```

```
>> gcc -O3 -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 out of 4\n
Hello from thread 2 out of 4\n
Hello from thread 3 out of 4\n
Hello from thread 1 out of 4\n
```

Compiling & Running an OpenMP program

Make sure to include omp header

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        printf("Hello from thread %d out of %d\n", thread_id, total_threads);
    }
    return 0;
}
```

Use optimization flag (we care about speedups)

```
>> gcc -O3 -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 out of 4\n
Hello from thread 2 out of 4\n
Hello from thread 3 out of 4\n
Hello from thread 1 out of 4\n
```

Compiling & Running an OpenMP program

Make sure to include omp header

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

int main() {
    #pragma omp parallel num_threads(4)
    {
        int thread_id = omp_get_thread_num();
        int total_threads = omp_get_num_threads();
        printf("Hello from thread %d out of %d\n", thread_id, total_threads);
    }
    return 0;
}
```

Use optimization flag (we care about speedups)

```
>> gcc -O3 -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 out of 4\n
Hello from thread 2 out of 4\n
Hello from thread 3 out of 4\n
Hello from thread 1 out of 4\n
```

Printing not guaranteed to be in order

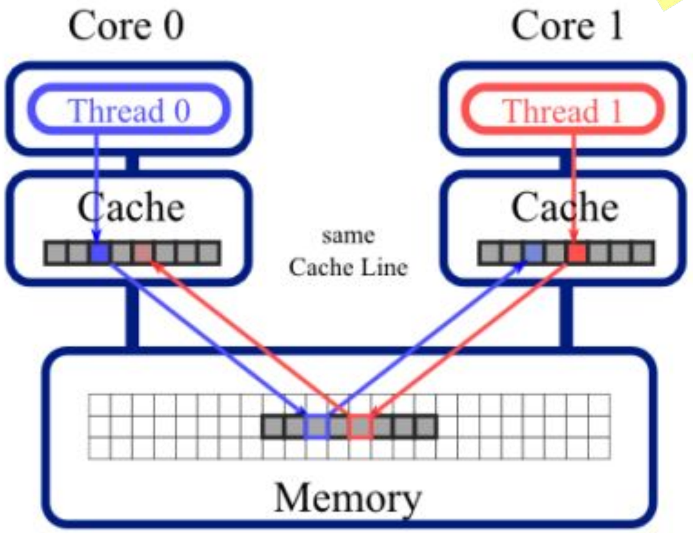
Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ **‘parallel’ Directive in Detail**
  - **False Sharing Examples**
  - Setting the Number of Threads
  - Additional Directives
  - threadprivate scope
- ❑ ‘for’ directive in Detail
  - Scheduling
  - Nesting

False Sharing

Recap

When multiple objects are very close together in memory & updated at around the same time by different processes → The program may run more slowly due to frequent cache invalidations



Program with False Sharing

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

#define N_THREADS 10
#define ITER 100000000

int main() {
    volatile int counter[N_THREADS];  // <--- All counters on the same cache line!

    double start = omp_get_wtime();

    #pragma omp parallel num_threads(N_THREADS)
    {
        int id = omp_get_thread_num();
        for (int i = 0; i < ITER; i++) {
            counter[id]++;  // <--- Each thread writes a different element
        }
    }

    double end = omp_get_wtime();

    printf("Time: %f seconds\n", end - start);
    return 0;
}
```

Program with False Sharing

Volatile is used here for demonstrating speedups (discussed further in later slide)

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

#define N_THREADS 10
#define ITER 100000000

int main() {
    volatile int counter[N_THREADS];  // <--- All counters on the same cache line!

    double start = omp_get_wtime();

    #pragma omp parallel num_threads(N_THREADS)
    {
        int id = omp_get_thread_num();
        for (int i = 0; i < ITER; i++) {
            counter[id]++;  // <--- Each thread writes a different element
        }
    }

    double end = omp_get_wtime();

    printf("Time: %f seconds\n", end - start);
    return 0;
}
```

Fix False Sharing with Padding

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

#define N_THREADS 10
#define ITER 100000000
#define CACHE_LINE_SIZE 128
#define PAD (CACHE_LINE_SIZE / sizeof(int))

int main() {
    volatile int counter[N_THREADS][PAD];  // <--- Each counter gets its own cache line

    double start = omp_get_wtime();

    #pragma omp parallel num_threads(N_THREADS)
    {
        int id = omp_get_thread_num();
        for (int i = 0; i < ITER; i++) {
            counter[id][0]++;  // <--- Each thread updates a separate cache line
        }
    }

    double end = omp_get_wtime();

    printf("Time: %f seconds\n", end - start);
    return 0;
}
```

Fix False Sharing with a Local Counter

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

#define N_THREADS 10
#define ITER 100000000

int main() {
   volatile int results[N_THREADS]; // shared results array

   double start = omp_get_wtime();

   #pragma omp parallel num_threads(N_THREADS)
   {
      int id = omp_get_thread_num();
      int local_count = 0;  // private counter

      for (int i = 0; i < ITER; i++) {
         local_count++;  // no contention here
      }

      results[id] = local_count;
   }

   double end = omp_get_wtime();
   printf("Time: %f seconds\n", end - start);

   return 0;
}
```

Runtimes on Mac with 10 Logical Cores

```bash
#!/bin/bash

# Array of source files
files=("false_sharing.c" "local_counter.c" "pad.c")

echo "Running without optimization"
# Loop through each file
for src in "${files[@]}"; do
   # Extract the base name without extension
   prog="${src%.c}"

   echo "Compiling $src..."
   gcc-15 -fopenmp "$src" -o "$prog"

   echo "Running $prog and printing timing information:"
   ./"$prog"

   echo "-----------------------------------------"
done
```

```
Running without optimization
Compiling false_sharing.c...
Running false_sharing and printing timing information:
Time: 1.144064 seconds
-----------------------------------------
Compiling local_counter.c...
Running local_counter and printing timing information:
Time: 0.126871 seconds
-----------------------------------------
Compiling pad.c...
Running pad and printing timing information:
Time: 0.159792 seconds
-----------------------------------------
```

Runtimes on Mac with 10 Logical Cores

We compile without optimization flags (-O3) and using the ***volatile*** keyword earlier as the compiler will make optimizations that 'fix' our setup to resolve false sharing problems. This occurs in our simple case where the compiler can easily identify the issue, but may not generalize well to the more complex cases of false sharing.

```
    prog="${src%.c}"

    echo "Compiling $src..."
    gcc-15 -fopenmp "$src" -o "$prog"

    echo "Running $prog and printing timing information:"
    ./"$prog"

    echo "----------------------------------------"
done
```

```
Running without optimization
Compiling false_sharing.c...
Running false_sharing and printing timing information:
Time: 1.144064 seconds
----------------------------------------
Compiling local_counter.c...
Running local_counter and printing timing information:
Time: 0.126871 seconds
----------------------------------------
Compiling pad.c...
Running pad and printing timing information:
Time: 0.159792 seconds
----------------------------------------
```

Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ **‘parallel’ Directive in Detail**
  - False Sharing Examples
  - **Setting the Number of Threads**
  - Additional Directives
  - threadprivate scope
- ❑ ‘for’ directive in Detail
  - Scheduling
  - Nesting

Setting the Number of Threads

There are three primary ways to set the number of threads

- ❑ Using the ***num_threads*** clause
- ❑ Using the ***void omp_set_num_threads(int num_threads)*** function
- ❑ Using the ***OMP_NUM_THREADS*** environment variable

Setting the Number of Threads

There are three primary ways to set the number of threads

- ❑ Using the ***num_threads*** clause
- ❑ Using the ***void omp_set_num_threads(int num_threads)*** function
- ❑ Using the ***OMP_NUM_THREADS*** environment variable

```
int main() {
    #pragma omp parallel num_threads(4)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

Setting the Number of Threads

There are three primary ways to set the number of threads

- ❑ Using the ***num_threads*** clause
- ❑ Using the ***void omp_set_num_threads(int num_threads)*** function
- ❑ Using the ***OMP_NUM_THREADS*** environment variable

```
int main() {
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

Setting the Number of Threads

There are three primary ways to set the number of threads

- ❑ Using the ***num_threads*** clause
- ❑ Using the ***void omp_set_num_threads(int num_threads)*** function
- ❑ Using the ***OMP_NUM_THREADS*** environment variable

```
int main() {
    #pragma omp parallel
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=4
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
```

What if we try to use multiple of these constructs at the same time with different numbers of threads?

Number of Threads Precedence

```
int main() {
    omp_set_num_threads(4);

    #pragma omp parallel num_threads(2)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=8
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
```

Number of Threads Precedence

```
int main() {
    omp_set_num_threads(4);

    #pragma omp parallel num_threads(2)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=8
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
```

OpenMP uses the following ordering:

(1) Use the ***num_threads*** clause argument if it is present

(2) Use the ***omp_set_num_threads*** argument if it is present

(3) Use the ***OMP_NUM_THREADS*** environment variable if it has been set

(4) OpenMP usually defaults to the number of logical cores - although this could vary by implementation/architecture

Number of Threads Precedence

```
int main() {
    omp_set_num_threads(4);

    #pragma omp parallel num_threads(2)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=8
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 2
Hello from thread 1 of 2
```

OpenMP uses the following ordering:

(1) Use the ***num_threads*** clause argument if it is present

(2) Use the ***omp_set_num_threads*** argument if it is present

(3) Use the ***OMP_NUM_THREADS*** environment variable if it has been set

(4) OpenMP usually defaults to the number of logical cores - although this could vary by implementation/architecture

A Note on Logical vs Physical Cores

- Hardware cores (a.k.a. physical cores) are the physical processing units on the CPU
- Logical cores incorporate superscalar processing capabilities
  - So if we have a 2-way superscalar processor, then each hardware core has 2 logical cores
  - You will also see n-way superscalar processing called n-way Simultaneous MultiThreading (SMT) or hyperthreading

CPU Model	**Physical Cores**	**Logical Cores (with SMT)**
4-core CPU	4	4
4-core w/ 2-way SMT	4	8
8-core CPU	8	8
8-core w/ 2-way SMT	8	16

Is there ever a time where the number of threads should exceed the number of logical cores?

Compute Heavy vs. I/O heavy tasks

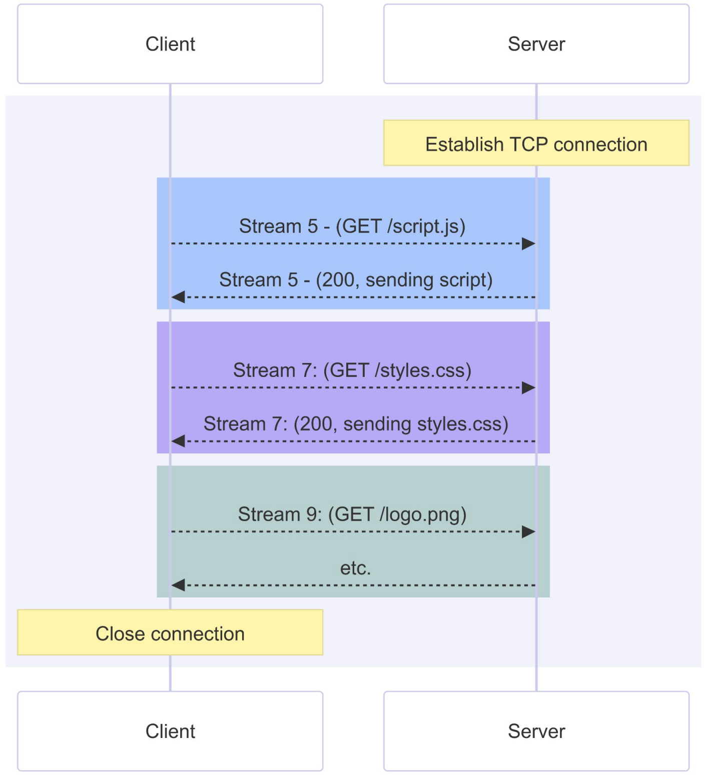
- When performing compute-heavy tasks (e.g. matrix multiplication), usually best practice is to set the number of threads equal to the number of logical cores
- On I/O-bound tasks (e.g. file loading), threads will idle while loading → it is better to use more threads than logical cores, if the system allows

```
int main() {
    extern const char *urls[NUM_URLS];  // Assume defined elsewhere

    // Oversubscribe: 64 threads
    omp_set_num_threads(64);

    #pragma omp parallel
    {
        int tid = omp_get_thread_num();
        int nthreads = omp_get_num_threads();

        // Calculate start/end indices for this thread
        int block_size = (NUM_URLS + nthreads - 1) / nthreads;  // Ceiling division
        int start = tid * block_size;
        int end = start + block_size;
        if (end > NUM_URLS) end = NUM_URLS;

        // Manually iterate over this thread's block
        for (int i = start; i < end; i++) {
            printf("Thread %d fetching: %s\n", tid, urls[i]);

            // Simulate a slow I/O operation
            for (volatile long j = 0; j < 100000000; j++);
        }
    }
}
```

I/O Thread Creation Limitations

Adding more threads than logical cores is usually helpful when computation is I/O bound (i.e. your program has a lot of data accesses). However this has some limitations

- ❑ I/O generally involves sending requests to some external location for data
- ❑ This includes Network/Disk/RAM access
- ❑ We are bound by
    - Maximum number of outgoing connections
    - External Policies (in networks, servers may choose to ignore or refuse concurrent requests if there are too many from one IP address)
    - Link Bandwidth
    - Queue depth at the receiving end
    - ...others (OS limitations, hardware limitations, contention limits, etc.)
- ❑ A usual good rule-of-thumb is to oversubscribe by ~10x the number of logical cores (although this can vary widely in practice)


Client
Server
Establish TCP connection
Stream 5 - (GET /script.js)
Stream 5 - (200, sending script)
Stream 7: (GET /styles.css)
Stream 7: (200, sending styles.css)
Stream 9: (GET /logo.png)
etc.
Close connection
Client
Server

Sometimes a program uses a different number of threads than we set - why?

Dynamic Threads

```
int main() {
    omp_set_num_threads(32);

    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=16
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 2
Hello from thread 1 of 2
```

Why does OpenMP only dispatch two threads?

Dynamic Threads

```
int main() {
    omp_set_num_threads(32);

    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_NUM_THREADS=16
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 2
Hello from thread 1 of 2
```

Why does OpenMP only dispatch two threads?

OpenMP can ***dynamically*** choose a different number of threads based on the system

Dynamic Threads

```
int main() {
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_DYNAMIC=TRUE
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 10
Hello from thread 1 of 10
…
```

How can we turn this ***dynamic*** capability on/off?

Dynamic Threads

```
int main() {
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_DYNAMIC=TRUE
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 10
Hello from thread 1 of 10
…
```

How can we turn this ***dynamic*** capability on/off?

Using environment variables or the ***omp_set_dynamic*** function. OpenMP will give higher precedence to the function, then look for the environment variable.

Dynamic Threads

```
int main() {
    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_DYNAMIC=FALSE
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 64
Hello from thread 1 of 64
…
```

How can we turn this ***dynamic*** capability on/off?

Using environment variables or the ***omp_set_dynamic*** function. OpenMP will give higher precedence to the function, then look for the environment variable.

Dynamic Threads

```
int main() {
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 64
Hello from thread 1 of 64
…
```

How can we turn this ***dynamic*** capability on/off?

Using environment variables or the ***omp_set_dynamic*** function. OpenMP will give higher precedence to the function, then look for the environment variable.

Dynamic Threads

```
int main() {
    #pragma omp parallel num_threads(64)
    {
        printf("Hello from thread %d of %d\n",
omp_get_thread_num(), omp_get_num_threads());
    }

    return 0;
}
```

```
>> export OMP_DYNAMIC=TRUE
>> gcc -fopenmp hello_omp.c -o hello_omp
>> ./hello_omp
Hello from thread 0 of 10
Hello from thread 1 of 10
…
```

Typically, OpenMP sets the number of threads equal to the number of logical cores when ***OMP_DYNAMIC*** is ***TRUE***, but may also choose a smaller number is there are many other threads from other programs executing

Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ **'parallel' Directive in Detail**
  - False Sharing Examples
  - Setting the Number of Threads
  - **Additional Directives**
  - threadprivate scope
- ❑ 'for' directive in Detail
  - Scheduling
  - Nesting

Additional Directives

- Specifying Concurrency
  - Specific directives
    - ✓ ***for***
    - ✓ ***sections***
  - Merging Directives
- Synchronization Directives
  - ***barrier***
  - ***critical***
  - ***atomic***
  - others (***single***, ***master***, ***ordered***, ***nowait***)

Additional Directives

Automatically breaks up for-loops on our behalf - we do not need to control the logic of which thread gets which set of iterations

- ❑ Specifying Concurrency
  - Specific directives
    - ✔ ***for***
    - ✔ ***sections***
  - Merging Directives
- ❑ Synchronization Directives
  - ***barrier***
  - ***critical***
  - ***atomic***
  - others (***single***, ***master***, ***ordered***, ***nowait***)

```
int main() {
    int sum = 0;

    #pragma omp parallel num_threads(8)
    {
        #pragma omp for reduction(+:sum)
        for (int i = 0; i < 100; i++) {
            sum += i;
        }
    }

    printf("Sum = %d\n", sum);
}
```

Additional Directives

Automatically splits independent code blocks into separate tasks — each section runs on a different thread without us needing to manage the distribution.

- ❑ Specifying Concurrency
  - Specific directives
    - ✔ *for*
    - ✔ ***sections***
  - Merging Directives
- ❑ Synchronization Directives
  - ***barrier***
  - ***critical***
  - ***atomic***
  - others (***single***, ***master***, ***ordered***, ***nowait***)

```
int main() {
    #pragma omp parallel
    {
        #pragma omp sections
        {
            #pragma omp section
            {
                printf("Section 1 running on thread %d\n",
omp_get_thread_num());
            }

            #pragma omp section
            {
                printf("Section 2 running on thread %d\n",
omp_get_thread_num());
            }
        }
    }
}
```

Additional Directives

- ❑ Specifying Concurrency
  - ○ Specific directives
    - ✔ *for*
    - ✔ ***sections***
  - ○ Merging Directives
- ❑ Synchronization Directives
  - **o** ***barrier***
  - **o** ***critical***
  - **o** ***atomic***
  - ○ others (***single***, ***master***, ***ordered***, ***nowait***)

We can combine the ***parallel*** directive with either the ***for*** or ***sections*** directives to create a merged directive that is easier to program

```
int main() {
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 100; i++) {
        sum += i;
    }

    printf("Sum = %d\n", sum);
}
```

Additional Directives

Forces all threads in a team to wait until every thread reaches the barrier before continuing

- ❑ Specifying Concurrency
  - ○ Specific directives
    - ✔ ***for***
    - ✔ ***sections***
  - ○ Merging Directives
- ❑ Synchronization Directives
  - **o** $\boxed{\textbf{\textit{barrier}}}$
  - **o** ***critical***
  - **o** ***atomic***
  - ○ others (***single***, ***master***, ***ordered***, ***nowait***)

```
int main() {
    #pragma omp parallel num_threads(4)
    {
        int tid = omp_get_thread_num();
        printf("Thread %d before barrier\n", tid);

        #pragma omp barrier  // all threads wait here

        printf("Thread %d after barrier\n", tid);
    }
}
```

Additional Directives

- Specifying Concurrency
  - Specific directives
    - ✔ ***for***
    - ✔ ***sections***
  - Merging Directives
- Synchronization Directives
  - ***barrier***
  - ***critical***
  - ***atomic***
  - others (***single***, ***master***, ***ordered***, ***nowait***)

Ensures that only one thread at a time executes the code inside the ***critical*** section to prevent race conditions.

```
int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        #pragma omp critical
        {
            sum += i;  // only one thread at a time can execute this
        }
    }

    printf("Sum = %d\n", sum);
}
```

Additional Directives

- ❑ Specifying Concurrency
  - ○ Specific directives
    - ✔ ***for***
    - ✔ ***sections***
  - ○ Merging Directives
- ❑ Synchronization Directives
  - ***o barrier***
  - ***o critical***
  - ***o*** $\boxed{\textbf{\textit{atomic}}}$
  - ○ others (***single***, ***master***, ***ordered***, ***nowait***)

Ensures that a specific memory update (like +=) is performed **atomically** so no two threads can interfere with each other. Like a 1-line version of ***critical***

```
int main() {
    int sum = 0;

    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        #pragma omp atomic
        sum += i;  // increment happens atomically
    }

    printf("Sum = %d\n", sum);
}
```

Additional Directives

- ❑ Specifying Concurrency
  - Specific directives
    - ✓ *for*
    - ✓ ***sections***
  - Merging Directives
- ❑ Synchronization Directives
  - ***barrier***
  - ***critical***
  - ***atomic***
  - others (***single***, ***master***, ***ordered***, ***nowait***)

Most directives in OpenMP only change program execution when placed inside of a ***parallel*** block. For example, the below atomic directive will have no impact as it is placed inside of the serial portion of the program.

```
int main() {
    #pragma omp atomic
    sum += 1

    #pragma omp parallel for
    for (int i = 0; i < 100; i++) {
        // Do parallel work here
    }
}
```

Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ **'parallel' Directive in Detail**
  - False Sharing Examples
  - Setting the Number of Threads
  - Additional Directives
  - **threadprivate scope**
- ❑ 'for' directive in Detail
  - Scheduling
  - Nesting

‘threadprivate’ scope [OpenMP directive]

```
// headers omitted
int counter = 0;              // A global variable
#pragma omp threadprivate(counter)  // ‘counter’ per-thread private

int main() {
    // Initialize the threadprivate variable differently in each thread
    #pragma omp parallel
    {
        counter = omp_get_thread_num() * 10;
        printf("Thread %d sets counter = %d\n",
                 omp_get_thread_num(), counter);
    }
    printf("\nAfter first parallel region:\n");
    // Access the variable again in a new parallel region
    #pragma omp parallel
    {
        printf("Thread %d sees counter = %d\n",
                  omp_get_thread_num(), counter);
    }
    return 0;
}
```

Unlike other scopes (***private, shared, firstprivate***), this is a directive, not a clause

‘threadprivate’ scope [OpenMP directive]

```
// headers omitted
int counter = 0;              // A global variable
#pragma omp threadprivate(counter)  // ‘counter’ per-thread private

int main() {
    // Initialize the threadprivate variable differently in each thread
    #pragma omp parallel
    {
        counter = omp_get_thread_num() * 10;
        printf("Thread %d sets counter = %d\n",
                  omp_get_thread_num(), counter);
    }
    printf("\nAfter first parallel region:\n");
    // Access the variable again in a new parallel region
    #pragma omp parallel
    {
        printf("Thread %d sees counter = %d\n",
                   omp_get_thread_num(), counter);
    }
    return 0;
}
```

Allows you to set per-thread variables which persist throughout the entirety of the program execution.

‘threadprivate’ scope [OpenMP directive]

```
// headers omitted
int counter = 0;              // A global variable
#pragma omp threadprivate(counter)  // ‘counter’ per-thread private

int main() {
    // Initialize the threadprivate variable differently in each thread
    #pragma omp parallel
    {
        counter = omp_get_thread_num() * 10;
        printf("Thread %d sets counter = %d\n",
                 omp_get_thread_num(), counter);
    }
    printf("\nAfter first parallel region:\n");
    // Access the variable again in a new parallel region
    #pragma omp parallel
    {
        printf("Thread %d sees counter = %d\n",
                  omp_get_thread_num(), counter);
    }
    return 0;
}
```

As a requirement - the number of threads must be constant throughout the entirety of program execution - we cannot use a different number of threads in each ***parallel*** block

‘threadprivate’ scope [OpenMP directive]

```
// headers omitted
int counter = 0;               // A global variable
#pragma omp threadprivate(counter)  // ‘counter’ per-thread private

int main() {
    // Initialize the threadprivate variable differently in each thread
    #pragma omp parallel
    {
        counter = omp_get_thread_num() * 10;
        printf("Thread %d sets counter = %d\n",
                 omp_get_thread_num(), counter);
    }
    printf("\nAfter first parallel region:\n");
    // Access the variable again in a new parallel region
    #pragma omp parallel
    {
        printf("Thread %d sees counter = %d\n",
                  omp_get_thread_num(), counter);
    }
    return 0;
}
```

Output

```
Thread 0 sets counter = 0
Thread 1 sets counter = 10
Thread 2 sets counter = 20
Thread 3 sets counter = 30

After first parallel region:
Thread 0 sees counter = 0
Thread 1 sees counter = 10
Thread 2 sees counter = 20
Thread 3 sees counter = 30
```

Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ ‘parallel’ Directive in Detail
    - False Sharing Examples
    - Setting the Number of Threads
    - Additional Directives
    - threadprivate scope
- ❑ **‘for’ directive in Detail**
    - **Scheduling**
    - Nesting

‘For’ directive

- For-loops give a standardized means of expressing computation
- Because of this standardization, we can have greater control of computation using the ***for*** directive
- Most of the things we will be parallelizing will involve for-loops $\longrightarrow$ The ‘for’ directive is thus very powerful and will be used in most of our examples

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

OpenMP first splits the array loops into ***num_threads*** separate chunks of size ***num_iterations/num_threads***. This is the default when no ***schedule*** clause is used.

Thread 0: i = 0, 1, 2
Thread 1: i = 3, 4, 5
Thread 2: i = 6, 7, 8

```
int main() {
    // Vector initialization omitted for brevity
    // Assume each vector is of length six

    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(static)
    for (int i = 0; i < 9; i++) {
        C[i] = A[i] + B[i];
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

We can also use cyclic task decomposition by adding an additional argument to the ***schedule*** clause. This argument details the size of the chunk to split the iterations into.

Size = 1

Thread 0: i = 0, 3, 6
Thread 1: i = 1, 4, 7
Thread 2: i = 2, 5, 8

```
int main() {
    // Vector initialization omitted for brevity
    // Assume each vector is of length six

    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(static, 1)
    for (int i = 0; i < 9; i++) {
        C[i] = A[i] + B[i];
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

Useful for load-balancing

We can also use cyclic task decomposition by adding an additional argument to the ***schedule*** clause. This argument details the size of the chunk to split the iterations into.

Size = 1

Thread 0: i = 0, 3, 6
Thread 1: i = 1, 4, 7
Thread 2: i = 2, 5, 8

```
int main() {
    // Vector initialization omitted for brevity
    // Assume each vector is of length six

    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(static, 1)
    for (int i = 0; i < 9; i++) {
        C[i] = A[i] + B[i];
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

We can also use cyclic task decomposition by adding an additional argument to the ***schedule*** clause. This argument details the size of the chunk to split the iterations into.

Size = 2

Thread 0: i = 0, 1, 6, 7
Thread 1: i = 2, 3, 8
Thread 2: i = 4, 5

```
int main() {
    // Vector initialization omitted for brevity
    // Assume each vector is of length six

    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(static, 2)
    for (int i = 0; i < 9; i++) {
        C[i] = A[i] + B[i];
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

Be careful to supply a reasonable chunk size argument which ***actually*** balances the load

We can also use cyclic task decomposition by adding an additional argument to the ***schedule*** clause. This argument details the size of the chunk to split the iterations into.

Size = 2

Thread 0: i = 0, 1, 6, 7
Thread 1: i = 2, 3, 8
Thread 2: i = 4, 5

```
int main() {
    // Vector initialization omitted for brevity
    // Assume each vector is of length six

    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(static, 2)
    for (int i = 0; i < 9; i++) {
        C[i] = A[i] + B[i];
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ [✓ Dynamic]
- ❑ Guided
- ❑ Runtime

Dynamic scheduling in OpenMP assigns loop iterations to threads at runtime, giving each thread a chunk of work as soon as it becomes available. It is useful when iterations have uneven or unpredictable workloads, because it helps balance the work across threads and reduces idle time.

We cannot know which threads get which iterations in advance. We only know they will get them once they finish their chunk.

```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(dynamic)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

Dynamic scheduling in OpenMP assigns loop iterations to threads at runtime, giving each thread a chunk of work as soon as it becomes available. It is useful when iterations have uneven or unpredictable workloads, because it helps balance the work across threads and reduces idle time.

The default chunk size is 1 if we do not pass an argument. If we pass an argument, the chunk size is set to that argument.

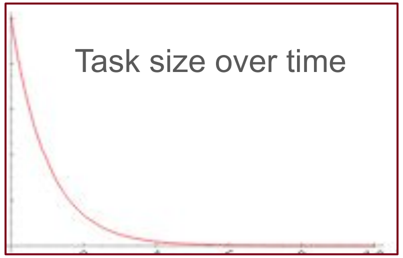
```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(dynamic, 2)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

Guided scheduling in OpenMP assigns loop iterations to threads in progressively smaller chunks, starting with large chunks and decreasing over time. It is useful for loops with uneven workloads because it reduces scheduling overhead at the start while dynamically balancing the remaining work as threads finish their chunks.



```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(guided)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ $\boxed{\text{Guided}}$
- ❑ Runtime

Guided scheduling adopts an approach similar to dynamic scheduling in that we do not know precisely how loop iterations will be provisioned at the beginning of program execution. The formula for the size of each chunk is:

$$\text{chunk\_size} = \left\lceil \frac{\text{remaining iterations}}{\text{number of threads}} \right\rceil$$

```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(guided)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Scheduling

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

We can use an argument that determines the minimum chunk size we want to provision to each thread when they request new tasks. No chunks are provisioned which are less than this value. The default used when no chunk size is specified is ‘1’.

```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(guided, 1)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Scheduling

Runtime scheduling in OpenMP defers the choice of scheduling policy (static, dynamic, guided, etc.) to runtime, allowing the program to use whatever schedule is set via the environment variable OMP_SCHEDULE. This is useful when you want to tune scheduling behavior without recompiling the code.

We can choose which threads are assigned to which iterations of the for loop using the ***scheduling*** clause:

- ❑ Static
- ❑ Dynamic
- ❑ Guided
- ❑ Runtime

```
int main() {
    // Parallel vector addition with static scheduling
    #pragma omp parallel for num_threads(3) schedule(runtime)
    for (int i = 0; i < N; i++) {
        // Perform some amount of unequal work here
    }

    // End of function calls omitted for brevity
}
```

Examples of how to set this variable

```
>> export OMP_SCHEDULE="dynamic,2"
>> export OMP_SCHEDULE="static"
>> export OMP_SCHEDULE="guided"
```

Lecture Overview

- ❑ Recap
- ❑ Compiling & Running an OpenMP program
- ❑ ‘parallel’ Directive in Detail
    - False Sharing Examples
    - Setting the Number of Threads
    - Additional Directives
    - threadprivate scope
- ❑ **‘for’ directive in Detail**
    - Scheduling
    - **Nesting**

Matrix Multiplication

- Up until now, we have only considered parallelizing one for-loop at a time
- However, many operations we will be examining are nested in nature
- If we have more logical cores than iterations of the outer for loop - we are not taking full advantage of all logical cores

```
int main() {
    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Nested Matrix Multiplication

- We can nest ***parallel for*** directives in order to launch new threads inside of each for loop
- ***Only*** useful when there are more logical cores than number of outer iterations or when operations are more I/O-heavy

```
int main() {
    omp_set_nested(1);

    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for private(k)
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            #pragma omp parallel for reduction(+:sum)
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Nested Matrix Multiplication

- We can nest ***parallel for*** directives in order to launch new threads inside of each for loop
- ***Only*** useful when there are more logical cores than number of outer iterations or when operations are more I/O-heavy

If we do not perform this, then nesting is disabled and only the outer for-loop will parallelize

```
int main() {
   omp_set_nested(1);

   #pragma omp parallel for private(j, k)
   for (i = 0; i < N; i++) {
      #pragma omp parallel for private(k)
      for (j = 0; j < N; j++) {
         double sum = 0.0;
         #pragma omp parallel for reduction(+:sum)
         for (k = 0; k < N; k++) {
            sum += A[i][k] * B[k][j];
         }
         C[i][j] = sum;
      }
   }
}
```

Nested Matrix Multiplication

We can also set this using (if we set the following env variable, *omp_set_nested(1)* is unnecessary) :

>> export OMP_NESTED=TRUE
>> ./your_program

- We can nest ***parallel for*** directives in order to launch new threads inside of each for loop
- ***Only*** useful when there are more logical cores than number of outer iterations or when operations are more I/O-heavy

```
int main() {
   omp_set_nested(1);

   #pragma omp parallel for private(j, k)
   for (i = 0; i < N; i++) {
      #pragma omp parallel for private(k)
      for (j = 0; j < N; j++) {
         double sum = 0.0;
         #pragma omp parallel for reduction(+:sum)
         for (k = 0; k < N; k++) {
            sum += A[i][k] * B[k][j];
         }
         C[i][j] = sum;
      }
   }
}
```

Nested Matrix Multiplication

- We can nest ***parallel for*** directives in order to launch new threads inside of each for loop
- ***Only*** useful when there are more logical cores than number of outer iterations or when operations are more I/O-heavy

Why make these variables private?

```
int main() {
    omp_set_nested(1);

    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for private(k)
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            #pragma omp parallel for reduction(+:sum)
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Nested Matrix Multiplication

Why use a local variable & not reduce directly into C[i][j] with ***reduction(+:C[i][j])***?

- We can nest ***parallel for*** directives in order to launch new threads inside of each for loop
- ***Only*** useful when there are more logical cores than number of outer iterations or when operations are more I/O-heavy

```
int main() {
    omp_set_nested(1);

    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for private(k)
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            #pragma omp parallel for reduction(+:sum)
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

Collapse Clause

You can replace nested calls using the ***collapse*** clause. This clause just combines for loops into one larger for loop.

```
int main() {
    omp_set_nested(1);

    #pragma omp parallel for private(j, k)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for private(k)
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```

```
int main() {
    #pragma omp parallel for collapse(2) private(k)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            double sum = 0.0;
            for (k = 0; k < N; k++) {
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}
```