CPSC 4240/5240: Parallel Programming Techniques

Lecture 6: OpenMP

Lecturer: Quanquan C. Liu

Spring 2026

Race Conditions in Parallel Instructions

```
#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;
}
```

Each << is a **separate instruction**; thread can interleave arbitrarily in between

E.g., Hello from thread Hello from thread 2 3 …

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!

Dealing with Loop Dependencies

- Last class we discussed issues with data race conditions
- Two threads writing to the same memory can cause incorrect outputs because they are competing to write
- OpenMP will likely **not warn** you about these race conditions
- Best way to resolve this is to restructure the algorithm/code to avoid race conditions
- Let's take a look at some more examples

OpenMP **nowait**

- By default, many work-sharing constructs contain a barrier at the end
  - for
  - sections
  - single
  - **Does not work for:** parallel

#pragma omp parallel for nowait

Will return an error

OpenMP **nowait**

- By default, many work-sharing constructs contain a barrier at the end
  - for
  - sections
  - single
  - **Does not work for:** parallel
- Recall that barriers act as joins, all threads complete tasks before going to the next parts of the code

OpenMP **nowait**

- OpenMP **nowait** construct modifies default synchronization behavior of OpenMP constructs
- Default OpenMP synchronization results in unnecessary waiting sometimes when not all threads need to finish before moving on to the next computation
- Let's take a look at an example

OpenMP **nowait**

Is this code correct?

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

OpenMP **nowait**

No! Why is it not correct?

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

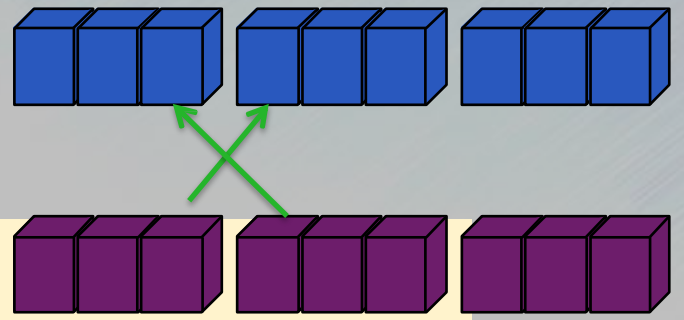
OpenMP **nowait**

Results will be incorrect because second loop **depends on first**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

OpenMP **nowait**



```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

OpenMP **nowait**

Suppose **a[i] = 0** and **b[i] = i** at initialization

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

OpenMP **nowait**

Suppose **a[i] = 0** and **b[i] = i** at initialization

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based or
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

Sample output:

Last 10 elements 'a':
a[990] = 0.00000
a[991] = 0.00000
a[992] = 0.00000
a[993] = 0.00000
a[994] = 0.00000
a[995] = 0.00000
a[996] = 0.00000
a[997] = 0.00000
a[998] = 49950.00000
a[999] = 0.00000

Last 10 elements 'b':
b[990] = 0.00000
b[991] = 0.00000
b[992] = 0.00000
b[993] = 0.00000
b[994] = 0.00000
b[995] = 0.00000
b[996] = 0.00000
b[997] = 0.00000
b[998] = 0.00000
b[999] = 999.00000

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based or
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

Is this correct?

Why did this happen?

Sample output:

Last 10 elements 'a':
a[990] = 0.00000
a[991] = 0.00000
a[992] = 0.00000
a[993] = 0.00000
a[994] = 0.00000
a[995] = 0.00000
a[996] = 0.00000
a[997] = 0.00000
a[998] = 49950.00000
a[999] = 0.00000

Last 10 elements 'b':
b[990] = 0.00000
b[991] = 0.00000
b[992] = 0.00000
b[993] = 0.00000
b[994] = 0.00000
b[995] = 0.00000
b[996] = 0.00000
b[997] = 0.00000
b[998] = 0.00000
b[999] = 999.00000

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

The b[i]'s **computed their values with the default values of a[i] before the a[i]'s computed their values**!

Sample output:

Last 10 elements 'a':
a[990] = 0.00000
a[991] = 0.00000
a[992] = 0.00000
a[993] = 0.00000
a[994] = 0.00000
a[995] = 0.00000
a[996] = 0.00000
a[997] = 0.00000
a[998] = 49950.00000
a[999] = 0.00000

Last 10 elements 'b':
b[990] = 0.00000
b[991] = 0.00000
b[992] = 0.00000
b[993] = 0.00000
b[994] = 0.00000
b[995] = 0.00000
b[996] = 0.00000
b[997] = 0.00000
b[998] = 0.00000
b[999] = 999.00000

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

Now **we remove nowait** so we allow for the default behavior with the for loop (with the barrier)

OpenMP **nowait**

```
#pragma omp parallel
  {
    // First loop: Compute a[i] based on b[i+1] and b[i-1]
    #pragma omp for
    for (int i = 1; i < N - 1; ++i) {
      a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
    }

    // Second loop: Compute b[i] based or
    #pragma omp for
    for (int i = N - 2; i > 0; --i) {
      b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
    }
  } // End of parallel region
```

Now **we remove nowait** so we allow for the default behavior with the for loop (with the barrier)

Sample output:

Last 10 elements 'a':
a[990] = 100.00000
a[991] = 100.00000
a[992] = 100.00000
a[993] = 100.00000
a[994] = 100.00000
a[995] = 100.00000
a[996] = 100.00000
a[997] = 100.00000
a[998] = 100.00000
a[999] = 0.00000

Last 10 elements 'b':
b[990] = 0.00000
b[991] = 0.00000
b[992] = 0.00000
b[993] = 0.00000
b[994] = 0.00000
b[995] = 0.00000
b[996] = 0.00000
b[997] = 0.00000
b[998] = -5000.00000
b[999] = 999.00000

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      // Second loop: Compute b[i] based or
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

This is now the correct behavior!

Questions?

Sample output:

Last 10 elements 'a':
a[990] = 100.00000
a[991] = 100.00000
a[992] = 100.00000
a[993] = 100.00000
a[994] = 100.00000
a[995] = 100.00000
a[996] = 100.00000
a[997] = 100.00000
a[998] = 100.00000
a[999] = 0.00000

Last 10 elements 'b':
b[990] = 0.00000
b[991] = 0.00000
b[992] = 0.00000
b[993] = 0.00000
b[994] = 0.00000
b[995] = 0.00000
b[996] = 0.00000
b[997] = 0.00000
b[998] = -5000.00000
b[999] = 999.00000

OpenMP **nowait**

Can also add an explicit barrier

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      #pragma omp barrier

      // Second loop: Compute b[i] based on a[i+1] and a[i-1]
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1] and b[i-1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      #pragma omp barrier

      // Second loop: Compute b[i] based on a[i
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

Can also add an explicit barrier

Explicit or implicit barrier: **no thread may pass the barrier until all threads** have arrived at the barrier

OpenMP **nowait**

```
#pragma omp parallel
   {
      // First loop: Compute a[i] based on b[i+1]
      #pragma omp for nowait
      for (int i = 1; i < N - 1; ++i) {
         a[i] = (b[i + 1] - b[i - 1]) / (2.0 * h);
      }

      #pragma omp barrier

      // Second loop: Compute b[i] based on a[i
      #pragma omp for
      for (int i = N - 2; i > 0; --i) {
         b[i] = (a[i + 1] - a[i - 1]) / (2.0 * h);
      }
   } // End of parallel region
```

In this case, no writes will occur in b until all threads have written in a

Explicit or implicit barrier: **no thread may pass the barrier until all threads** have arrived at the barrier

More Dependency Examples

- Some loops cannot be parallelized at all!
- Need to be careful about which loops you can or cannot parallelize
- Let’s look at some examples

More Dependency Examples

- Which of the following loops can OpenMP parallelize?

A)

```
for (int i = 1; i < N; i++)
    A[i] = A[i] + B[i-1];
```

B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[i] + 1;
```

C)

```
for (int i = 1; i < N; i++)
    A[i] = A[0] + A[i];
```

D)

```
for (int i = 0; i < N-1; i++) {
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

More Dependency Examples

- Which of the following loops can OpenMP parallelize?

A)

```
for (int i = 1; i < N; i++)
    A[i] = A[i] + B[i-1];
```

B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[i] + 1;
```

C)

```
for (int i = 1; i < N; i++)
    A[i] = A[0] + A[i];
```

D)

```
for (int i = 0; i < N-1; i++) {
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

More Dependency Examples

Let's go through these examples one by one!

- Which of the following loops can OpenMP parallelize?

A)

```
for (int i = 1; i < N; i++)
    A[i] = A[i] + B[i-1];
```

B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[i] + 1;
```

C)

```
for (int i = 1; i < N; i++)
    A[i] = A[0] + A[i];
```

D)

```
for (int i = 0; i < N-1; i++) {
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

More Dependency Examples

Let's go through these examples one by one!

A)

```
for (int i = 1; i < N; i++)
    A[i] = A[i] + B[i-1];
```

- Each A[i] is calculated **independently of other A[i]'s**
- Each A[i] depends only on itself and B[i-1] neither of which are modified by other threads

More Dependency Examples

Let's go through these examples one by one!

- Here, each A[i] depends on the previous value of A[i]
- What can go wrong here?
- **A[i + 1] calculated with old value of A[i]** before A[i] is calculated!

B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[i] + 1;
```

More Dependency Examples

Let's go through these examples one by one!

C)

```
for (int i = 1; i < N; i++)
    A[i] = A[0] + A[i];
```

- A[i] only depends on itself and A[0]
- Iteration starts at 1 and doesn't recompute A[0]
- Hence, A[0] is a static value and each A[i] computation is independent of other A[i]'s

More Dependency Examples

Let's go through these examples one by one!

- There are several dependencies in this computation
- What can go wrong here? Hint: **threads can interleave!**
  - Meaning **a thread can perform some computation in the middle of another thread's computation**

D)

```
for (int i = 0; i < N-1; i++) {
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

More Dependency Examples

Let's go through these examples one by one!

- There are several dependencies in this computation
- What can go wrong here? Hint: **threads can interleave!**
  - Meaning **a thread can perform some computation in the middle of another thread's computation**

- One thread can compute E[i] **after C[i+1] computed by another thread**

D)

```
for (int i = 0; i < N-1; i++) {
    A[i] = B[i];
    C[i] = A[i] + B[i];
    E[i] = C[i+1];
}
```

How would you change the algorithm to parallelize loop B?

Parallelized B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[0] + i;
```

B)

```
for (int i = 0; i < N -1; i++)
    A[i + 1] = A[i] + 1;
```

Another example for nowait

- To ensure correctness of the following code, where must we remove the nowait clause(s)?

Another example for nowait

```
#pragma omp parallel for shared(c) private(i) nowait
    for (i=0; i<N; i++)
      c[i] = (double) i

#pragma omp parallel for shared(c) private(i) nowait
    for (i=1; i<N; i+=2)
      c[i] = c[i] + c[i-1]

#pragma omp parallel for shared(c, d) private(i) nowait
    for (i=1; i<N; i+=2)
      d[i] = c[i] + c[i-1]

#pragma omp parallel for shared(c, e) private(i) nowait
    for (i=2; i<N; i+=2)
      e[i] = c[i] + c[i-1]
```

Another example for nowait

```
#pragma omp parallel for shared(c) private(i) nowait
    for (i=0; i<N; i++)
      c[i] = (double) i

#pragma omp parallel for shared(c) private(i) nowait
    for (i=1; i<N; i+=2)
      c[i] = c[i] + c[i-1]

#pragma omp parallel for shared(c, e) private(i) nowait
    for (i=1; i<N; i+=2)
      d[i] = e[i] + e[i-1]

#pragma omp parallel for shared(c, e) private(i) nowait
    for (i=2; i<N; i+=2)
      e[i] = c[i] + c[i-1]
```

Another example for nowait

```
#pragma omp parallel for shared(c) private(i) nowait
    for (i=0; i<N; i++)
      c[i] = (double) i

#pragma omp parallel for shared(c) pr
    for (i=1; i<N; i+=2)
      c[i] = c[i] + c[i-1]

#pragma omp parallel for shared(c, e)
    for (i=1; i<N; i+=2)
      d[i] = e[i] + e[i-1]

#pragma omp parallel for shared(c, e) private(i) nowait
    for (i=2; i<N; i+=2)
      e[i] = c[i] + c[i-1]
```

Now **need to remove the nowait on the third loop** because we cannot use the new value of e!

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 1 thread:	7	7	7	7	11	12	13	14

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

B[0] += B[7], B[1] += B[6], B[2] += B[5]
B[3] += B[4], B[4] += B[3], B[5] += B[2]
B[6] += B[1], B[7] += B[0]

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 10 threads:	7	7	9	7	11	7	13	7

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

B[0] += B[7], B[1] += B[6], B[2] += B[5]
B[3] += B[4], B[4] += B[3], B[5] += B[2]
B[6] += B[1], B[7] +=B[0]

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 10 threads:	**7**	**7**	**9**	**7**	**11**	**7**	**13**	**7**

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

What is happening here?

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 10 threads:	7	7	9	7	11	7	13	7

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

B[7] updated before B[0]!
B[5] updated before B[2]!
B[2] updated after B[5]!

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 10 threads:	**7**	**7**	**9**	**7**	**11**	**7**	**13**	**7**

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

How do we fix this?

Removing Data Dependencies More Examples

- How can we split the following loop into two loops so that we can parallelize each loop and the result is correct?

B initially:	0	1	2	3	4	5	6	7
B on 10 threads:	**7**	**7**	**9**	**7**	**11**	**7**	**13**	**7**

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < N; i++)
    B[i] += B[N-1-i];
```

How do we fix this? Hint: **think about splitting the loop at some point**

Removing Data Dependencies More Examples

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < int(N/2 – 0.5); i++)
    B[i] += B[N-1-i];

#pragma omp parallel for shared (N,B)
for (i = int(N/2 + 0.5); i < N; i++)
    B[i] += B[N-1-i];
```

How do we fix this? Hint: **think about splitting the loop at some point**

Removing Data Dependencies More Examples

```
#pragma omp parallel for shared (N,B)
for (i = 0; i < int(N/2 – 0.5); i++)
    B[i] += B[N-1-i];

#pragma omp parallel for shared (N,B)
for (i = int(N/2 + 0.5); i < N; i++)
    B[i] += B[N-1-i];
```

Why? Because only the second half depends on the first half

Removing Data Dependencies More Examples

Is this still correct now?

```
#pragma omp parallel shared(N, B)
{
  #pragma omp for nowait
   for(int i = 0; i <= int(N/2 - 0.5); ++i) {
      B[i] += B[N-1-i];
   }

   #pragma omp for
   for(int i = int(N/2 + 0.5); i < N; ++i) {
      B[i] += B[N-1-i];
   }
}
```

Removing Data Dependencies More Examp

No! Revert back to the previous situation

```
#pragma omp parallel shared(N, B)
{
  #pragma omp for nowait
   for(int i = 0; i <= int(N/2 - 0.5); ++i) {
      B[i] += B[N-1-i];
   }

   #pragma omp for
   for(int i = int(N/2 + 0.5); i < N; ++i) {
      B[i] += B[N-1-i];
   }
}
```

OpenMP Loop Scheduling

- OpenMP has more fine-grained control over **loop scheduling**
- **Loop scheduling** determines how iterations of a parallel loop are divided among available threads
- Helps with balancing workloads, reducing latency (idle time), maximize CPU utilization, in some cases, helps with correctness
- #pragma omp for schedule(...)

OpenMP Scheduling: A Quick Aside

- **OpenMP Default Scheduler for loops**
  - Allows compilers to define their default scheduling policy
  - Can vary between different compilers and different versions of the same compiler
  - Static scheduling is the default across most compilers: GCC, Clang, Intel's ICC
  - Loop iterations divided into equal-sized chunks, each thread assigned a specific chunk before loop begins
- Other types of schedulers: **work-stealing** (discussed in later lectures!)

OpenMP Loop Scheduling

- Types of schedules: #pragma omp for schedule(...)
- **Static scheduling**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads in round-robin fashion before loop starts
  - Hence it is static

```
#pragma omp for schedule(static)
#pragma omp for schedule(static, chunk_size)
```

OpenMP Loop Scheduling

- Types of schedules: #pragma omp for schedule(...)
- **Static scheduling**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads in round-robin fashion before loop starts
  - Hence it is static

Default chunk size when no chunk_size passed in

```
#pragma omp for schedule(static)
#pragma omp for schedule(static, chunk_size)
```

OpenMP Loop Scheduling

- Types of schedules: #pragma omp for schedule(...)
- **Static scheduling**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads in round-robin fashion before loop starts
  - Hence it is static

```
#pragma omp for schedule(s
#pragma omp for schedule(s
```

Default chunk size when no chunk_size passed in

Default chunk size is:

$$\frac{Total\ Iters}{Num\ Threads}$$

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads!

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads! Typically you don't have to assign the chunk_size yourself

OpenMP Loop Scheduling

- **Static scheduling**
  - **Pros**:
    - Predictable work distribution: each thread receives roughly equal number of iterations
    - Low overhead for runtime decision-making
    - Better cache performance
  - **Cons**:
    - Potential load imbalance for irregular workloads

Best for regular workloads! Typically you don't have to assign the chunk_size yourself

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become available
  - Hence it is dynamic

```
#pragma omp for schedule(dynamic)
#pragma omp for schedule(dynamic, chunk_size)
```

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become available
  - Hence it is dynamic

```
#pragma omp for schedule(dy
#pragma omp for schedule(dy
```

Default chunk size is determined by compiler for dynamic scheduling

Default for most compilers is 1

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - Divides loop iterations into chunks of a fixed size and assigns them to threads **on-the-fly** as the threads become availa
  - Hence

```
#pra
#pra
```

Default chunk size is determined by compiler for dynamic scheduling

Default for most compilers is 1

Why is it different? One is **extreme for uniform workloads** the other **extreme for irregular workloads**

OpenMP Loop Scheduling

- **Dynamic Scheduling:**
  - **Pros**:
    - Better load balancing for irregular workloads
    - Reduced risk of load imbalance
  - **Cons**:
    - Increased overhead
    - Nondeterministic execution order
    - Potential memory overhead for task queues and scheduling structures

Lecture 5 Quiz