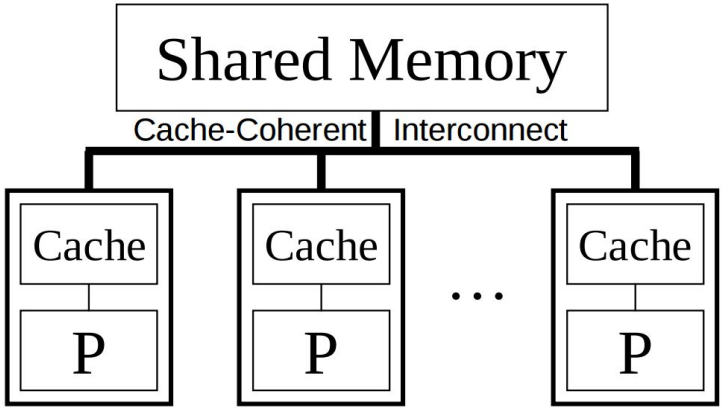
Introduction to OpenMP

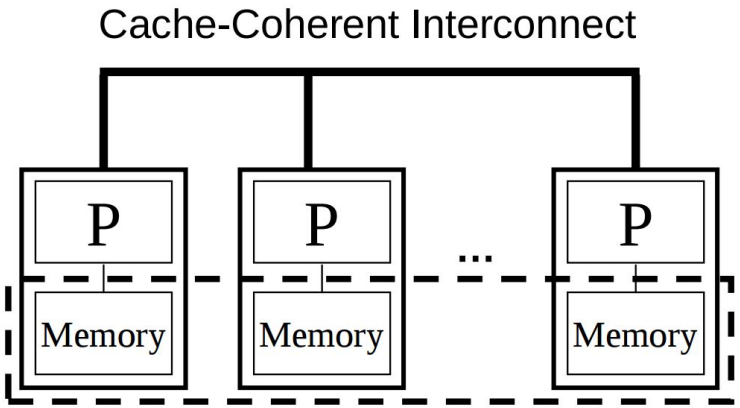
OpenMP (Open Multi-Processing)

OpenMP is a shared-memory application programming interface (API).

- programs are executed on one or more processors that share some or all of the available memory.
- programs are typically executed by multiple independent threads that share data but may also have some additional private data.
- provides a means for starting up threads, assigning work to them, and coordinating their accesses to shared data.



a shared-memory parallel computer



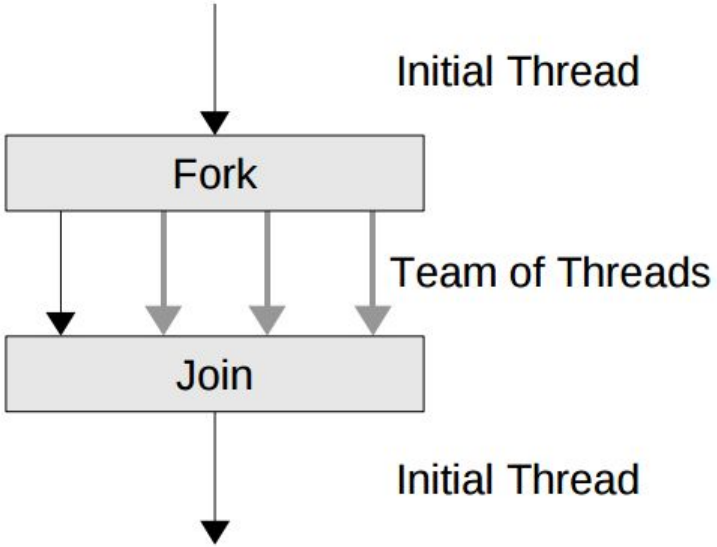
a distributed shared-memory platform

Runs efficiently

- OpenMP consists of a set of compiler #pragmas that can be added to a sequential program in Fortran, C, or C++ to control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield the correct behavior, but without any parallelism.

- By the C++ standard, if the compiler encounters a #pragma that it does not support, it will ignore it (remember #pragma twice from Quiz2). So adding the OMP statements can be done safely without breaking compatibility in case of compilers that don’t support OpenMP.

The Fork-Join Programming Model



The program starts as a single thread of execution, the initial thread. A team of threads is forked at the beginning of a parallel region and joined at the end.

Parallel Construct

The parallel construct

- The parallel construct starts a parallel block.

- It creates a team of N threads (N is determined at runtime, usually from the number of CPU cores) all of which execute the statements in the block. After the statements, the threads join back into one.

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

// compile using: g++ helloWorld.cpp -fopenmp -o helloWorld

int main () {

    #pragma omp parallel num_threads(4)
    {
        // code inside this region runs in parallel
        std::cout << "Thread id: " << omp_get_thread_num() << std::endl;
        std::cout << "Hello World!";
    }
    std::cout << std::endl;

    return 0;
}
```

Program Output:

```
Thread id: 0
Thread id: Hello World!1
Hello World!Thread id: 2
Hello World!Thread id: 3
Hello World!
```

Sharing the Work among Threads

The for-loop construct: #pragma omp for

- The for loop construct splits the for loop so that each thread in the current team handles a different portion of the loop.

- **#pragma omp for** only assigns portions of the loop for different threads in the current team.

- To create a new team of threads, parallel keyword needs to be specified.

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

// compile using: g++ forLoop.cpp -fopenmp -o forLoop

int main () {

    // set the number of threads to maximum possible
    omp_set_num_threads( omp_get_max_threads() );

    #pragma omp parallel
    {
        #pragma omp for
        for(int n = 0; n < 10; n++)
        {
            std::cout << n << " ";
        }
    }
    std::cout << std::endl;

    return 0;
}
```

Program Output: 1 0 3 5 7 8 6 4 9 2

- The parallel construct and the loop construct could be combined.

```
#pragma omp parallel for
for(int n = 0; n < 10; n++)
{
    std::cout << n << " ";
}
```

A ***team*** is a group of threads that execute currently. At the program beginning, the team consists of a single thread.

A ***parallel*** construct splits the current thread into a ***new team*** of threads for the duration of the next block/statement, after which the team merges back into one.

For divides the work of the for loop among the threads of the ***current team***. It does not create threads, it only divides the work amongst the threads of the currently executing team.

Parallel for is a shorthand for two commands at once: ***parallel*** and ***for***. Parallel creates a new team, and ***for*** splits that team to handle different portions of the loop.

If the program never contains a ***parallel*** construct, there is never more than one thread; the master thread that starts the program and runs it, as in non-threading programs.

Clauses to Control Parallel and Work-Sharing Constructs

Scheduling

- The scheduling of the for loop can be explicitly controlled.

- There are five scheduling types: ***static***, ***dynamic***, ***guided***, ***auto*** and ***runtime***.

Syntax:

#pragma omp parallel for schedule(kind [,chunk size])

Static schedules

- Divides the loop into equal-sized chunks or as equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size.

- By default, chunk size is ***loop_count*** / ***number_of_threads***. Set chunk to 1 to interleave the iterations.

- It is not needed to specify static scheduling in the parallel for directive as it is the default (unless a chunk size different than 1 is needed).

```
#pragma omp parallel for schedule(static)
for(int n = 0; n < 10; n++)
{
    std::cout << n << " ";
}
```

- A static schedule can be non-optimal, however. This is the case when the different iterations take different amounts of time.

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ staticInefficientSchedule.cpp -fopenmp -o staticInefficientSchedule

int main () {

    int i;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(static) num_threads(4)
    for(i=0; i < 16; i++)
    {
        // wait for i seconds
        sleep(i);
        std::cout << "Thread " << omp_get_thread_num() << " has completed iteration " << i << std::endl;
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Thread 0 has completed iteration 0
Thread 0 has completed iteration 1
Thread 0 has completed iteration 2
Thread 1 has completed iteration 4
Thread 0 has completed iteration 3
Thread 2 has completed iteration 8
Thread 1 has completed iteration 5
Thread 3 has completed iteration 12
Thread 1 has completed iteration 6
Thread 2 has completed iteration 9
Thread 1 has completed iteration 7
Thread 3 has completed iteration 13
Thread 2 has completed iteration 10
Thread 2 has completed iteration 11
Thread 3 has completed iteration 14
Thread 3 has completed iteration 15
All done!
Total execution time: 54.0007 seconds
```

- The serial program would take 120 seconds to run (0+1+2..+15 = 120 seconds)
- The static schedule took ~54 seconds to run

Dynamic Schedules

- Uses the internal work queue to give a chunk-sized block of loop iterations to each thread. By default, the chunk size is 1.

- When a thread is finished, it retrieves the next block of loop iterations from the top of the work queue.

- Be careful when using this scheduling type because of the extra overhead involved.

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ dynamicEfficientSchedule.cpp -fopenmp -o dynamicEfficientSchedule

int main () {

    int i;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(dynamic) num_threads(4)
    for(i=0; i < 16; i++)
    {
        // wait for i seconds
        sleep(i);
        std::cout << "Thread " << omp_get_thread_num() << " has completed iteration " << i << std::endl;
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Thread 1 has completed iteration 0
Thread 0 has completed iteration 1
Thread 2 has completed iteration 2
Thread 3 has completed iteration 3
Thread 1 has completed iteration 4
Thread 0 has completed iteration 5
Thread 2 has completed iteration 6
Thread 3 has completed iteration 7
Thread 1 has completed iteration 8
Thread 0 has completed iteration 9
Thread 2 has completed iteration 10
Thread 3 has completed iteration 11
Thread 1 has completed iteration 12
Thread 0 has completed iteration 13
Thread 2 has completed iteration 14
Thread 3 has completed iteration 15
All done!
Total execution time: 36.0006 seconds
```

- The serial program would take 120 seconds to run (0+1+2..+15 = 120 seconds)
- The static schedule took ~54 seconds to run
- The dynamic schedule took ~36 seconds to run

- Dynamic scheduling is better when the iterations may take very different amounts of time. However, there is some overhead to dynamic scheduling.

- The following program demonstrates this overhead:

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ dynamicInefficientSchedule.cpp -fopenmp -o dynamicInefficientSchedule

#define THREADS 16
#define N 100000000

int main () {

    int i;

    std::cout << "Running " << N << " iterations on " << THREADS << " threads dynamically" << std::endl;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(dynamic) num_threads(THREADS)
    for(i=0; i < N; i++)
    {
        // a loop that does not take very long
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Running 100000000 iterations on 16 threads dynamically
All done!
Total execution time: 13.0064 seconds
```

- The serial program would take 0.23 seconds to run!
- The dynamic schedule took ~13 seconds to run

- If we specify static scheduling, the program will run faster:

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ staticEfficientSchedule.cpp -fopenmp -o staticEfficientSchedule

#define THREADS 16
#define N 100000000

int main () {

    int i;

    std::cout << "Running " << N << " iterations on " << THREADS << " threads statically" << std::endl;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(static) num_threads(THREADS)
    for(i=0; i < N; i++)
    {
        // a loop that does not take very long
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Running 100000000 iterations on 16 threads statically
All done!
Total execution time: 0.0247741 seconds
```

- The serial program would take 0.23 seconds to run!
- The dynamic schedule took ~13 seconds to run
- The static schedule took ~0.025 seconds to run

- In dynamic scheduling it is costly to call the runtime library! The chunk size can be specified to lessen the number of calls to the runtime library:

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ dynamicEfficientSchedule2.cpp -fopenmp -o dynamicEfficientSchedule2

#define THREADS 16
#define N 100000000
#define CHUNK 100

int main () {

    int i;

    std::cout << "Running " << N << " iterations on " << THREADS << " threads dynamically" << std::endl;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(dynamic, CHUNK) num_threads(THREADS)
    for(i=0; i < N; i++)
    {
        // a loop that does not take very long
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Running 100000000 iterations on 16 threads dynamically
All done!
Total execution time: 0.0958957 seconds
```

- The serial program would take 0.23 seconds to run!
- The dynamic schedule took ~13 seconds to run
- The static schedule took ~0.025 seconds to run
- The dynamic schedule with chunk size different than 1 took ~0.096 seconds to run

- **Increasing the chunk size in dynamic scheduling makes the scheduling more static, and decreasing it makes it more dynamic.**

Guided Schedules

- The scheduling policy is similar to a dynamic schedule, except that the chunk size changes as the program runs. It begins with big chunks, but then adjusts to smaller chunk sizes if the workload is imbalanced.

- The optional chunk parameter specifies the minimum chunk size to use. By default the chunk size is approximately ***loop_count / number_of_threads***

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ guidedSchedule1.cpp -fopenmp -o guidedSchedule1

#define THREADS 16
#define N 100000000

int main () {

    int i;

    std::cout << "Running " << N << " iterations on " << THREADS << " threads guided" << std::endl;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(guided) num_threads(THREADS)
    for(i=0; i < N; i++)
    {
        // a loop that does not take very long
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Running 100000000 iterations on 16 threads guided
All done!
Total execution time: 0.0184597 seconds
```

- The serial program would take 0.23 seconds to run!
- The dynamic schedule took ~13 seconds to run
- The static schedule took ~0.025 seconds to run
- The dynamic schedule with chunk size different than 1 took ~0.096 seconds to run
- The guided schedule took ~0.018 seconds to run

```cpp
#include <omp.h>
#include <unistd.h>
#include <iostream>
#include <chrono>

// compile using: g++ guidedSchedule2.cpp -fopenmp -o guidedSchedule2

int main () {

    int i;

    auto t1 = std::chrono::high_resolution_clock::now();

    #pragma omp parallel for schedule(guided) num_threads(4)
    for(i=0; i < 16; i++)
    {
        // wait for i seconds
        sleep(i);
        std::cout << "Thread " << omp_get_thread_num() << " has completed iteration " << i << std::endl;
    }

    auto t2 = std::chrono::high_resolution_clock::now();

    // measure execution time
    std::chrono::duration<double> executionTime = t2 - t1;

    // all threads done
    std::cout << "All done!" << std::endl;
    std::cout << "Total execution time: " << executionTime.count() << " seconds" << std::endl;
    return 0;
}
```

```
Thread 0 has completed iteration 0
Thread 0 has completed iteration 1
Thread 0 has completed iteration 2
Thread 1 has completed iteration 4
Thread 0 has completed iteration 3
Thread 2 has completed iteration 7
Thread 1 has completed iteration 5
Thread 3 has completed iteration 10
Thread 2 has completed iteration 8
Thread 1 has completed iteration 6
Thread 0 has completed iteration 12
Thread 3 has completed iteration 11
Thread 2 has completed iteration 9
Thread 1 has completed iteration 13
Thread 0 has completed iteration 14
Thread 3 has completed iteration 15
All done!
Total execution time: 36.0005 seconds
```

- The serial program would take 120 seconds to run (0+1+2..+15 = 120 seconds)
- The static schedule took ~54 seconds to run
- The dynamic schedule took ~36 seconds to run
- The guided schedule took ~36 seconds to run

Observations:

In general,

- Static schedules have little overhead and work best when each iteration takes approximately equal amount of time.
- Dynamic schedules work best when the work is uneven across the iterations. The work will be split more evenly across threads.
- Use of chunks or using a guided schedule provides a trade-off between the two schemes.

Auto Schedules

The decision regarding scheduling is delegated to the compiler. The compiler freely chooses any possible mapping of iterations to threads in the team.

Runtime Schedules

Uses the OMP_SCHEDULE environment variable to specify which one of the three loop-scheduling types should be used.

It is possible to force that certain events within the loop happen in a predicted order, using the ordered clause.

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; n++)
{
    files[n].compress();

    #pragma omp ordered
    send(files[n]);
}
```

```
#pragma omp for ordered schedule(dynamic)
for(int n=0; n<100; n++)
{
    files[n].compress();

    #pragma omp ordered
    send(files[n]);
}
```

- This loop "compresses" 100 files with some files being compressed in parallel, but ensures that the files are "sent" in a strictly sequential order.
- If the thread assigned to compress file 7 is done but the file 6 has not yet been sent, the thread will wait before sending, and before starting to compress another file.
- The ordered clause in the loop guarantees that there always exists one thread that is handling the lowest-numbered unhandled task.
- Each file is compressed and sent exactly once, but the compression may happen in parallel.

Controlling which data to share between threads

- The variables in a parallel region needs to be scoped with the ***private***, ***shared***, ***reduction***, ***firstprivate*** or ***lastprivate*** clause.

- By default, all variables are shared except those declared within the parallel block.

- The loop index variable is automatically private, and not changes to it inside the loop are allowed

It is a good practice to declare all the variables that will be used inside a parallel block outside the block and to explicitly specify their scope

```
int i;
int j;
int prime;
int n = 100;

#pragma omp parallel \
shared (n) \
private (i, j, prime) \
```

Firstprivate and lastprivate clauses

- Private copy is an uninitialized variable by the same name and same type as the original variable; it does ***not*** copy the value of the variable that was in the surrounding context.

- In the case of primitive data types (int, float, char* etc.), the private variable is uninitialized, just like any declared but not initialized local variable.

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

// compile using: g++ privateExample.cpp -fopenmp -o privateExample

int main () {

    std::string a = "x";
    std::string b = "y";
    int c = 3;

    #pragma omp parallel private(a,c) shared(b) num_threads(2)
    {
        a += "k"; // string concatenation
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")" << std::endl;
        std::cout << c << std::endl;
    }

    return 0;
}
```

Program Output:

```
A becomes (k), b is (y)A becomes (k), b is (y)

7
7
```

At the entrance of the block, “a” becomes a new instance of std::string, that is initialized with the default constructor; it is not initialized with the copy constructor.

If we actually need a copy of the original value, we need to use the ***firstprivate*** clause instead.

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

// compile using: g++ firstPrivateExample.cpp -fopenmp -o firstPrivateExample

int main () {

    std::string a = "x";
    std::string b = "y";
    int c = 3;

    #pragma omp parallel firstprivate(a,c) shared(b) num_threads(2)
    {
        a += "k"; // string concatenation
        c += 7;
        std::cout << "A becomes (" << a << "), b is (" << b << ")" << std::endl;
        std::cout << c << std::endl;
    }

    return 0;
}
```

Program Output:

```
A becomes (A becomes (xk), b is (y)xk), b is (y)
10
10
```

- The lastprivate clause defines a variable private as in firstprivate or private, but causes the value from the last task to be copied back to the original value after the end of the loop/sections construct.

- In a loop construct (for construct), the last value is the value assigned by the thread that handles the last iteration of the loop (***not*** the last operation at runtime). Values assigned during other iterations are ignored.

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

// compile using: g++ lastPrivateExample.cpp -fopenmp -o lastPrivateExample

int main () {

    int i;
    int x = 44;

    #pragma omp parallel for lastprivate(x) num_threads(4)
    for(i = 0; i <= 10; i++)
    {
        x = i;
        std::cout << "Thread number: " << omp_get_thread_num() << " x: " << x << std::endl;
    }

    std::cout << "x is " << x << std::endl;

    return 0;
}
```

```
Thread number: 0 x: 0
Thread number: 0 x: 1
Thread number: Thread number: 3 x: 9
Thread number: 3 x: 10
Thread number: 2 x: 6
Thread number: 2 x: 7
Thread number: 2 x: 8
Thread number: 0 x: 2
1 x: 3
Thread number: 1 x: 4
Thread number: 1 x: 5
x is 10
```

Note that x is set to 10 and not 5 outside the parallel block.

Reduction clause

- The reduction clause specifies that one or more variables that are private to each thread are the subject of a reduction operation at the end of the parallel region.
  - at the beginning of the parallel block, a private copy is made of the variable and pre-initialized to a certain value .
  - at the end of the parallel block, the private copy is atomically merged into the shared variable using the defined operator.

Syntax:

reduction(operation:var)

- The reduction operators and the proper initialization values are as follows:

Operator	Initialization value
+, -, |, ^, ||	0
*, &&	1
&	~0
min	largest representable number
max	smallest representable number

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

// compile using: g++ reductionExample.cpp -fopenmp -o reductionExample

int factorial (int number)
{
    int fac = 1;
    #pragma omp parallel for reduction(* : fac) num_threads(2)
    for(int n = 2; n <= number; n++)
    {
        fac *= n;
    }

    return fac;
}


int main () {

    std::cout << "10!: " << factorial(10) << std::endl;

    return 0;
}
```

Program Output:

```
10!: 3628800
```

The default clause

- The most useful purpose on the default clause is to check whether you have remembered to consider all variables for the private/shared question, using the default(none) setting.

```
int a = 0;
int b = 0;

// This code won't compile:
// It requires explicitly specifying whether "a" is shared or private.
#pragma omp parallel default(none) shared(b)
{
    b += a;
}
```

Synchronization Constructs

The critical construct

- The critical construct specifies that code is only be executed on one thread at a time.

- The critical construct may optionally contain a global name that identifies the type of the critical construct. No two threads can execute a critical construct of the same name at the same time.

```
// find the maximum using critical section
maximumValue = data[0];

auto t1 = std::chrono::high_resolution_clock::now();

#pragma omp parallel for num_threads(4)
for(int i = 1; i < SIZE; i++)
{
    if (data[i] > maximumValue)
    {
        #pragma omp critical
        {
            // compare data[i] and maximumValue again because maximumValue
            // could have been changed by another thread after the comparison
            // outside the critical section
            if (data[i] > maximumValue)
                maximumValue = data[i];
        }
    }
}

auto t2 = std::chrono::high_resolution_clock::now();

// measure execution time
executionTime = t2 - t1;

std::cout << "Maximum value (critical section): " << maximumValue << std::endl;
std::cout << "Execution time: " << executionTime.count() << " seconds" << std::endl;
```

```
// find the maximum using reduction
maximumValue = data[0];

t1 = std::chrono::high_resolution_clock::now();

#pragma omp parallel for reduction(max : maximumValue) num_threads(4)
for(int i = 1; i < SIZE; i++)
{
    if (data[i] > maximumValue)
        maximumValue = data[i];
}

t2 = std::chrono::high_resolution_clock::now();

// measure execution time
executionTime = t2 - t1;

std::cout << "Maximum value (reduction): " << maximumValue << std::endl;
std::cout << "Execution time: " << executionTime.count() << " seconds" << std::endl;
```

```
Maximum value (critical section): 2147483611
Execution time: 0.0616284 seconds
Maximum value (reduction): 2147483611
Execution time: 0.0620229 seconds
```

- In this example, similar performances are achieved.

Warnings

- STL is not thread-safe. If STL containers are used in a parallel context, concurrent access must be excluded using locks or other mechanisms. Const access is usually fine.

- Exceptions may not be thrown and caught across omp constructs.

References

The material in the slides are heavily taken from the following references:

- Guide into OpenMP: Easy multithreading programming for C++. Retrieved November 28, 2018, from https://bisqwit.iki.fi/story/howto/openmp/
- MSDN Library. Retrieved November 28, 2018, from https://msdn.microsoft.com/en-us/library
- Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. Using OpenMP: portable shared memory parallel programming. Vol. 10. MIT press, 2008.
- OpenMP Scheduling. Retrieved November 28, 2018, from http://www.inf.ufsc.br/~bosco.sobral/ensino/ine5645/OpenMP_Dynamic_Scheduling.pdf
- OpenMP Loop Scheduling. Retrieved November 28, 2018, from https://software.intel.com/en-us/articles/openmp-loop-scheduling
- OpenMP Tutorial - firstprivate and lastprivate. Retrieved November 28, 2018, from https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-firstprivate-and-lastprivate/
- C++ Examples of Parallel Programming with OpenMP. Retrieved November 28, 2018, from https://people.sc.fsu.edu/~jburkardt/cpp_src/openmp/openmp.html