



SHARED MEMORY PARALLELISM OPENMP

Kamal Beydoun
I3340 Parallel Computing
Spring 2023-2024

Compiled based on

- (1) Fundamentals of Parallelism on Intel® Architecture – Colfax Research
- (2) Parallel Programming in OpenMP and Java - Stephen A. Edwards - Columbia University

OUTLINE

Shared Memory & Multithreading

Fork-Join Concurrency Model

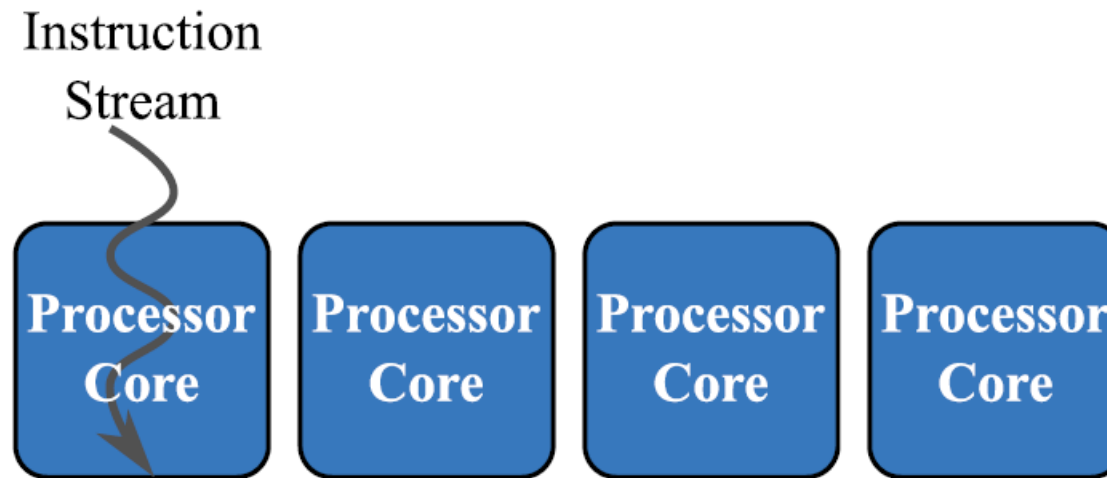
OpenMP

- Basics
- Private & Shared Variables
- Parallel For
- Critical sections
- Reduction
- Barriers

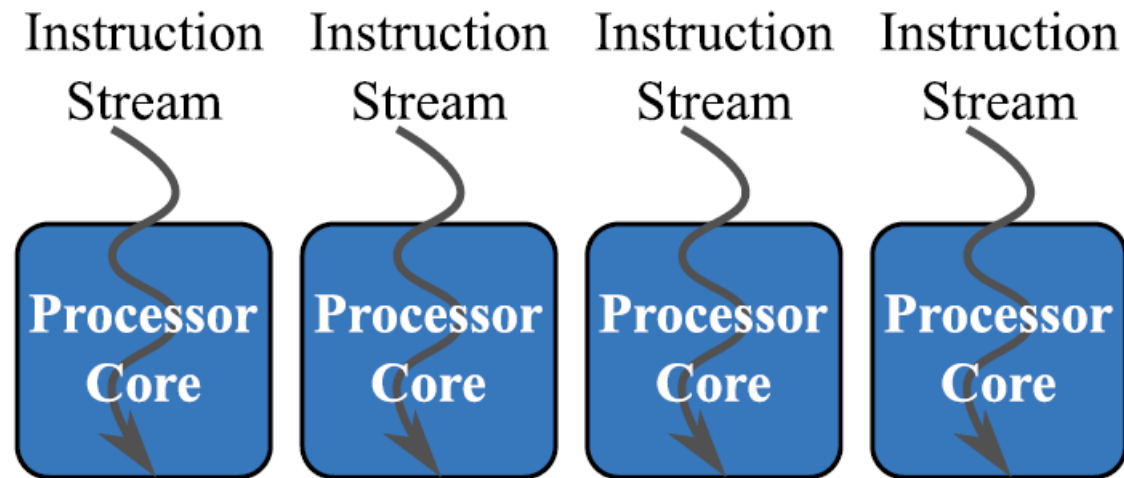
SERIAL PROCESSOR, SERIAL CODE



PARALLEL PROCESSOR, SERIAL CODE



PARALLEL PROCESSOR, PARALLEL CODE



PROCESSES VS. THREADS

Process

- Separate address space
- Explicit inter-process communication
- Synchronization part of communication
- Operating system calls: `fork()`, `wait()`

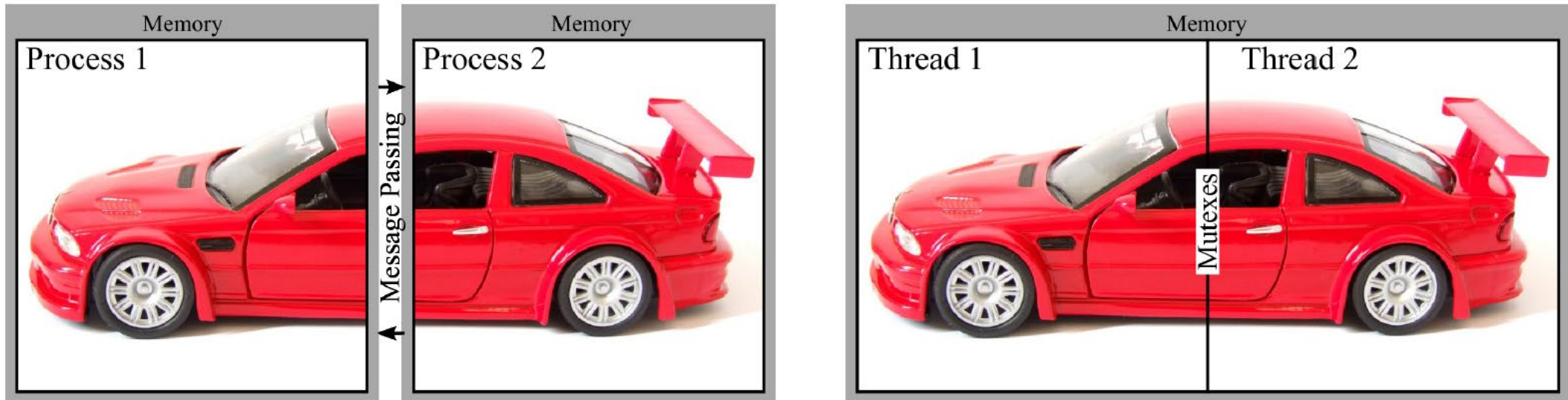
Thread

- Shared address spaces
- Separate PC and stacks
- Communication is through shared memory
- Synchronization done explicitly
- Library defined, e.g. Pthreads or Java Threads

Threads are easier to start, context-switch & terminate (but no big influence on speed)
Library threads are typically cross-platform

DATA SHARING VS. SHARING

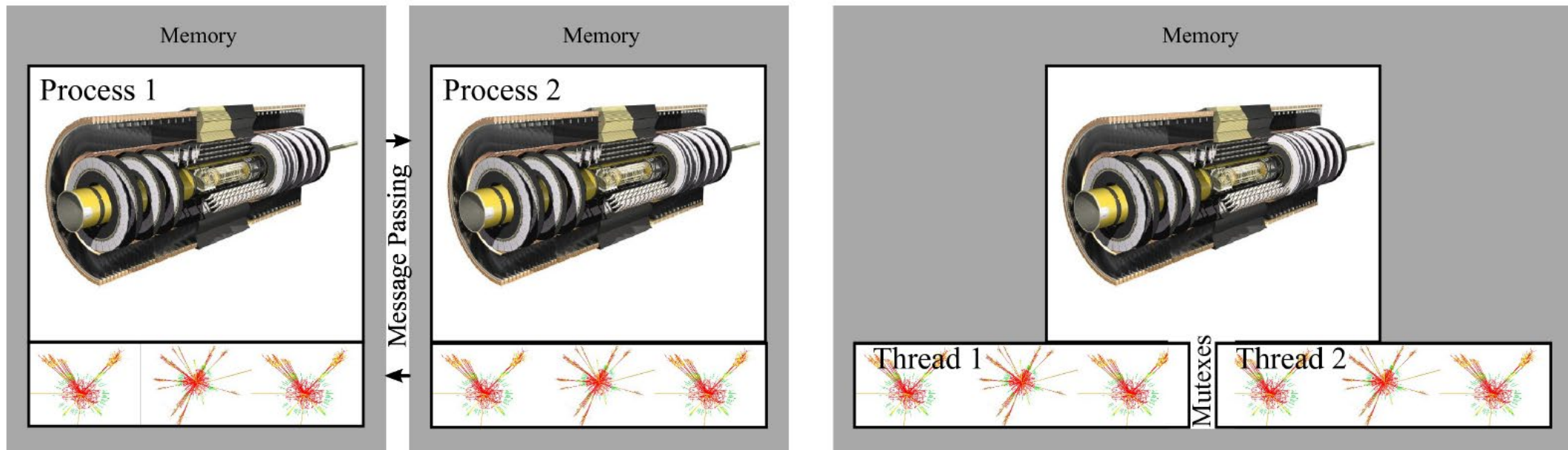
Option 1: Partitioning data set between threads/processes



Example: Image processing, computational fluid dynamics (CFD)

DATA SHARING VS. SHARING

Option 2: Sharing data set between threads/processes



Examples: particle transport simulation, machine learning (inference)

THREADING FRAMEWORKS

Framework	Functionality	Platform
C++11 Threads	Asynchronous functions	only C++
POSIX Threads (Pthreads)	Fork-Join + Synchronization	C/C++/Fortran; Linux
Cilk Plus	Async tasks, loops, reducers, load balance	C/C++
TBB	Trees of tasks, complex patterns	only C++
OpenMP	Tasks, loops, reduction, load balancing, affinity, nesting, (+SIMD, offload)	C/C++/Fortran
Java Multithreading + Concurrency Package	Fork-Join + Synchronization High-level => less granular control	Java; Cross-Platform

OpenMP

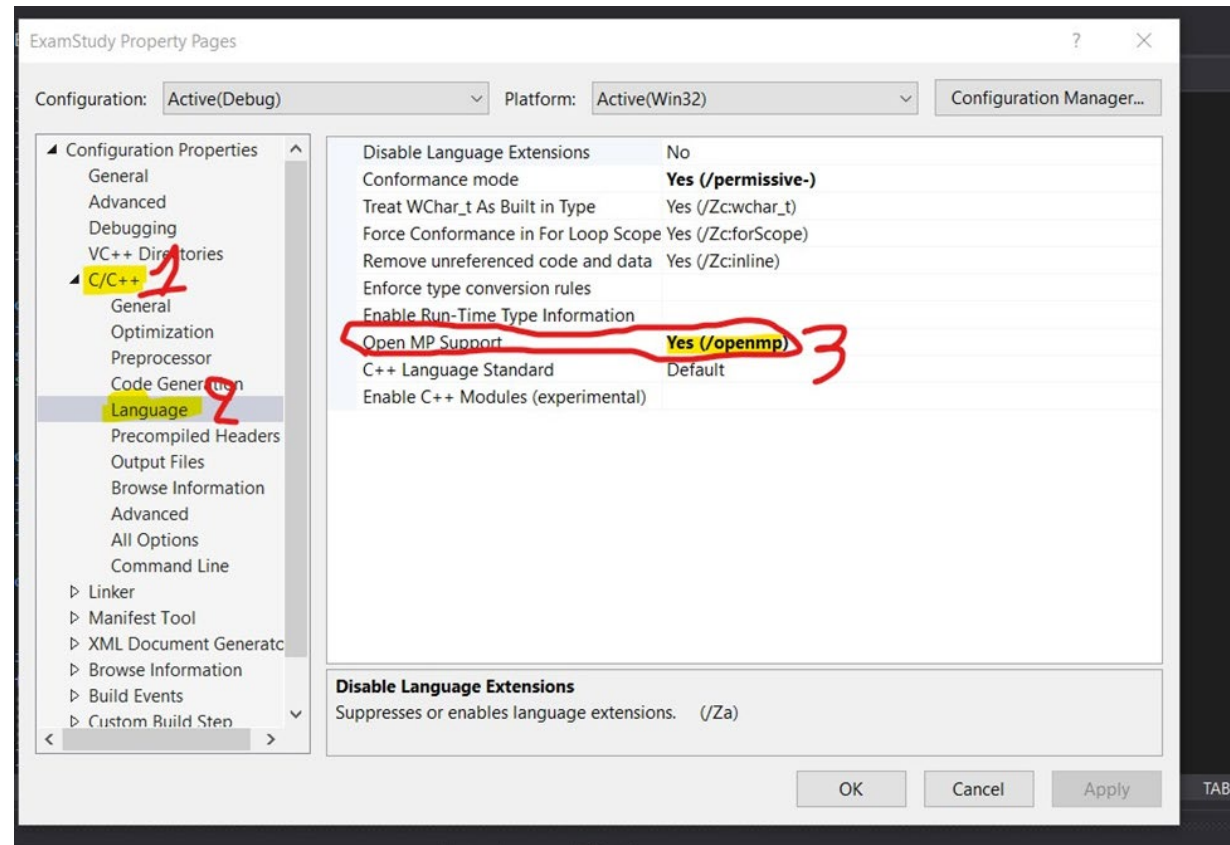


- OpenMP = “Open Multi-Processing”
- Computing-oriented **framework** for shared-memory programming
- Distribute threads across CPU cores for parallel speedup
 - Threads: streams of instructions that share memory address space
- It provides a **high-level abstraction layer** over low-level multithreading primitives
- Its programming model and interface are **portable** to different compilers and hardware architectures, **scalable** over an arbitrary number of CPU cores, and flexible in terms of writing compact and expressive source code.

OpenMP-BASICS

- The basic philosophy of OpenMP is to **augment sequential code** by using special comment-like compiler directives – so-called **pragmas** – to give hints to the compiler on how the code can be parallelized.
- Pragmas can be used to **exploit compiler-specific functionality**. If the compiler **does not support** the corresponding feature it will simply ignore the pragma.
- By just adding appropriate pragmas, OpenMP can be easily used to parallelize existing sequential code.
- Written software can be compiled and executed with a **single or multiple threads**. Whether the compiler should include support for the OpenMP API is specified by the compiler flag **-fopenmp**.
- If we compile the code without this flag, we end up with **sequential** code.

VISUAL STUDIO — PROJECT PROPERTIES



"HELLO WORLD" OpenMP PROGRAM

```
1  #include <omp.h>
2  #include <stdio.h>
3
4  int main(){
5      // This code is executed by 1 thread
6      const int nt=omp_get_max_threads();
7      printf("OpenMP with %d threads\n", nt);
8
9      #pragma omp parallel
10     { // This code is executed in parallel
11         // by multiple threads
12         printf("Hello World from thread %d\n",
13             omp_get_thread_num());
14     }
15 }
```

```
$ gcc -fopenmp hello_omp.cc -o hello_omp
```

```
$ export OMP_NUM_THREADS=5
```

```
$ ./hello_omp
```

OpenMP with 5 threads

Hello World from thread 0

Hello World from thread 3

Hello World from thread 1

Hello World from thread 2

Hello World from thread 4

OMP_NUM_THREADS controls number of OpenMP threads (default: logical CPU count)

The group of threads that is currently executing the program is called a *team*.

The **parallel** construct creates a team of threads which execute in parallel.

<https://www.perfmatrix.com/physical-cpu-and-logical-cpu/>

BASIC OPENMP FUNCTIONS

- **omp_set_num_threads(nthreads)** - **set** the number of threads used in a parallel region
- **omp_get_num_threads()** - **get** the number of threads used in a parallel region
- **omp_get_thread_num()** - **get the thread rank** in a parallel region
 - $(0 \rightarrow \text{omp_get_num_threads}() - 1)$

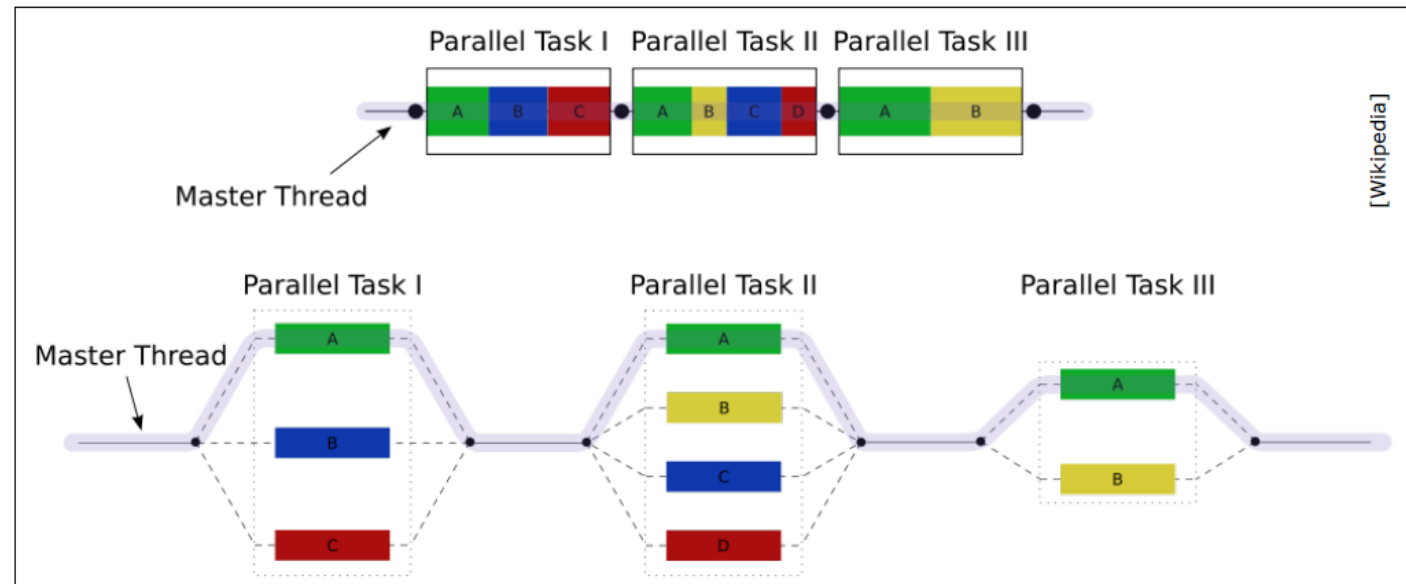
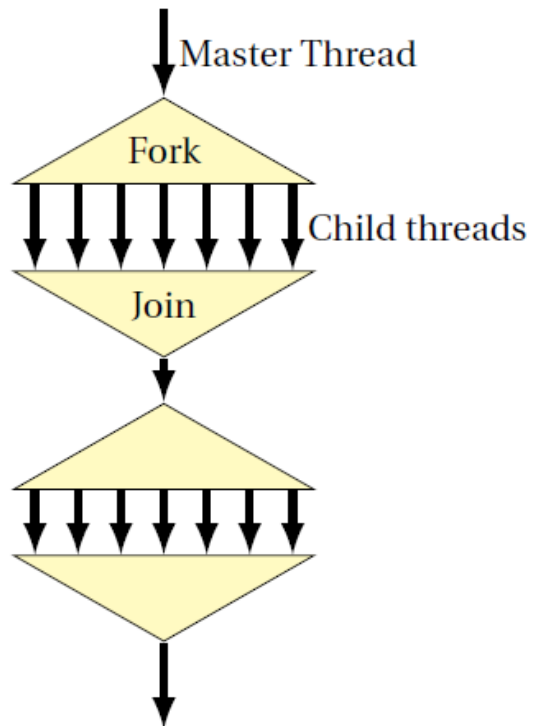
"HELLO WORLD" OpenMP PROGRAM

```
1  #include <iostream>
2  #include <omp.h>
3
4  int main() {
5      // run the block after the pragma in four threads
6      #pragma omp parallel num_threads(4)
7      {
8          int i = omp_get_thread_num();
9          int n = omp_get_num_threads();
10         std::cout << "Hello world from thread "
11                 << i << " of " << n << std::endl;
12     }
13 }
```

```
./hello_world
Hello world from thread 1 of 4
Hello world from thread 0 of 4
Hello world from thread 2 of 4
Hello world from thread 3 of 4
```

FORK-JOIN TASK MODEL

Spawn tasks in parallel, run each to completion, collect the results.



PARALLEL REGIONS

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

int main()
{
    # pragma omp parallel
    { /* Fork threads */

        printf("This is thread %d\n",
               omp_get_thread_num());

    } /* Join */

    printf("Done.\n");

    return 0;
}
```

```
$ ./parallel
This is thread 1
This is thread 3
This is thread 2
This is thread 0
Done.
$ ./parallel
This is thread 2
This is thread 1
This is thread 3
This is thread 0
Done.
$ OMP_NUM_THREADS=8 ./parallel
This is thread 0
This is thread 4
This is thread 1
This is thread 5
This is thread 3
This is thread 6
This is thread 2
This is thread 7
Done.
```

CONTROL OF VARIABLE SHARING

Method 1: using **clauses** in `pragma omp parallel` (C, C++, Fortran):

```
1 int A, B; // Variables declared at the beginning of a function
2 #pragma omp parallel private(A) shared(B)
3 {
4     // Each thread has its own copy of A, but B is shared
5 }
```

CONTROL OF VARIABLE SHARING

Method 2: using **scoping** (only C and C++):

```
1 int B; // Variable declared outside of parallel scope - shared by default
2 #pragma omp parallel
3 {
4     int A; // Variable declared inside the parallel scope - always private
5     // Each thread has its own copy of A, but B is shared
6 }
```

DATA SCOPE - SHARED

shared (*comma-separated-list-of-var-names*)

- Declares variables in its list to be **shared** among all threads **in the team**.
- A shared variable exists in only one memory location and all threads can read or write to that address.
- It is **the programmer's responsibility** to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections).

DATA SCOPE - PRIVATE

private (*comma-separated-list-of-var-names*)

- Declares variables in its list to **be private to each thread**.
- A **new object** of the same type is declared once **for each thread in the team (including main)**
- All references to the original object **are replaced** with references to the new object
- Should be assumed to be **uninitialized** for each thread

EXAMPLE

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[]) {
    int nthreads, tid;
    /* Fork a team of threads giving them their own copies of variables */
    #pragma omp parallel private(nthreads, tid)
    {
        /* Obtain thread number */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        /* Only master thread does this */
        if (tid == 0) {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        /* All threads join master thread and disband */
    }
}
```

Microsoft Visual Studio Debug Console

```
Hello World from thread = 3
Hello World from thread = 0
Number of threads = 4
Hello World from thread = 2
Hello World from thread = 1
```

EXAMPLE

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#define N 100
int main (int argc, char *argv[]) {
    int nthreads, tid, i;
    float a[N], b[N], c[N];
    /* Some initializations */
    for (i=0; i < N; i++)
        a[i] = b[i] = i;
    #pragma omp parallel shared(a,b,c,nthreads) private(i,tid)
    {
        tid = omp_get_thread_num();
        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
        printf("Thread %d starting...\n",tid);
        #pragma omp for
        for (i=0; i<N; i++)
        {
            c[i] = a[i] + b[i];
            printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
        }
    } /* end of parallel section */
}
```

DATA SCOPE - DEFAULT

default (shared | none)

- Allows the **user** to specify a default **shared** scope for all variables in the lexical extent of any parallel region.
- Specific variables can be **exempted** from the default using the **private**, **shared**, **firstprivate**, **lastprivate**, and **reduction** clauses.
- Using **none** as a default requires that the programmer **should explicitly** scope all variables.

EXAMPLE

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

/**
 * @brief Illustrates the OpenMP default clause.
 * @details An int is passed to a parallel region. Then:
 *   - 1st step: thread 0 writes "123" in the int
 *   - 2nd step: thread 1 prints the value of the int
 * The default policy, set to shared, becomes visible when the value read by thread 1 is the one written by thread 0.
 */
int main(int argc, char* argv[])
{
    // Use 2 OpenMP threads
    omp_set_num_threads(2);

    // The int that will be shared among threads
    int val = 0;

    // Variables not part of a data-sharing clause will be "shared" by default.
    #pragma omp parallel default(shared)
    {
        // Step 1: thread 0 writes the value
        if(omp_get_thread_num() == 0)
        {
            printf("Thread 0 sets the value of \"val\" to 123.\n");
            val = 123;
        }

        // Threads wait each other before progressing to step 2
        #pragma omp barrier

        // Step 2: thread 1 reads the value
        if(omp_get_thread_num() == 1)
        {
            printf("Thread 1 reads the value of \"val\": %d.\n", val);
        }
    }

    return EXIT_SUCCESS;
}
```

EXAMPLE

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

/**
 * @brief Illustrates the OpenMP none policy.
 * @details This example shows that the usage of the "none" default, by
 * comparing a version using implicit data-sharing clauses against that using
 * explicit data-sharing clauses. Both yield the same result, but one requires
 * the explicit usage of data-sharing clauses.
 */
int main(int argc, char* argv[])
{
    // Use 2 OpenMP threads
    omp_set_num_threads(2);

    // Relying on the implicit default(shared)
    int implicitlyShared = 0;
    #pragma omp parallel
    {
        #pragma omp atomic
        implicitlyShared++;
    }
    printf("Value with implicit shared: %d.\n", implicitlyShared);

    // Forcing the usage of explicit data-sharing clauses
    int explicitlyShared = 0;
    #pragma omp parallel default(none) shared(explicitlyShared)
    {
        #pragma omp atomic
        explicitlyShared++;
    }
    printf("Value with explicit shared: %d.\n", explicitlyShared);

    return EXIT_SUCCESS;
}
```

DATA SCOPE - OTHERS

○ **firstprivate**

- combines the behavior of the PRIVATE clause with **automatic initialization** of the variables in its list.
- Listed variables are **initialized according to the value of their original objects** prior to entry into the parallel or work-sharing construct.

○ **lastprivate**

- combines the behavior of the PRIVATE clause with a copy from the **last loop iteration** or **section** to the **original variable object**.
- The value copied back into the original variable object is obtained from the **last (sequentially) iteration** or **section** of the enclosing construct

EXAMPLE

```
/**
 * @brief Illustrates the OpenMP firstprivate policy.
 * @details This example shows that when a firstprivate variable is passed to a
 * parallel region, threads work on initialised copies but that whatever
 * modification is made to their copies is not reflected onto the original
 * variable.
 */
int main(int argc, char* argv[])
{
    // Use 4 OpenMP threads
    omp_set_num_threads(4);

    // Variable that will be firstprivate
    int val = 123456789;

    printf("Value of \"val\" before the OpenMP parallel region: %d.\n", val);

    #pragma omp parallel firstprivate(val)
    {
        printf("Thread %d sees \"val\" = %d, and updates it to be %d.\n", omp_get_thread_num(), val, omp_get_thread_num());
        val = omp_get_thread_num();
    }

    // Value after the parallel region; unchanged.
    printf("Value of \"val\" after the OpenMP parallel region: %d.\n", val);

    return EXIT_SUCCESS;
}
```

Microsoft Visual Studio Debug Console

```
Value of "val" before the OpenMP parallel region: 123456789.
Thread 0 sees "val" = 123456789, and updates it to be 0.
Thread 2 sees "val" = 123456789, and updates it to be 2.
Thread 3 sees "val" = 123456789, and updates it to be 3.
Thread 1 sees "val" = 123456789, and updates it to be 1.
Value of "val" after the OpenMP parallel region: 123456789.
```

EXAMPLE

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

/**
 * @brief Illustrates the OpenMP lastprivate policy.
 * @details This example shows that when a lastprivate variable is passed to a
 * parallelised for loop, threads work on uninitialised copies but, at the end
 * of the parallelised for loop, the thread in charge of the last iteration
 * sets the value of the original variable to that of its own copy.
 */
int main(int argc, char* argv[])
{
    // Use 4 OpenMP threads
    omp_set_num_threads(4);

    // Variable that will be lastprivate
    int val = 123456789;

    printf("Value of \"val\" before the OpenMP parallel region: %d.\n", val);

    #pragma omp parallel for lastprivate(val)
    for(int i = 0; i < omp_get_num_threads(); i++)
    {
        val = omp_get_thread_num();
    }

    // Value after the parallel region; unchanged.
    printf("Value of \"val\" after the OpenMP parallel region: %d. Thread %d was therefore the last one to modify it.\n", val, val);

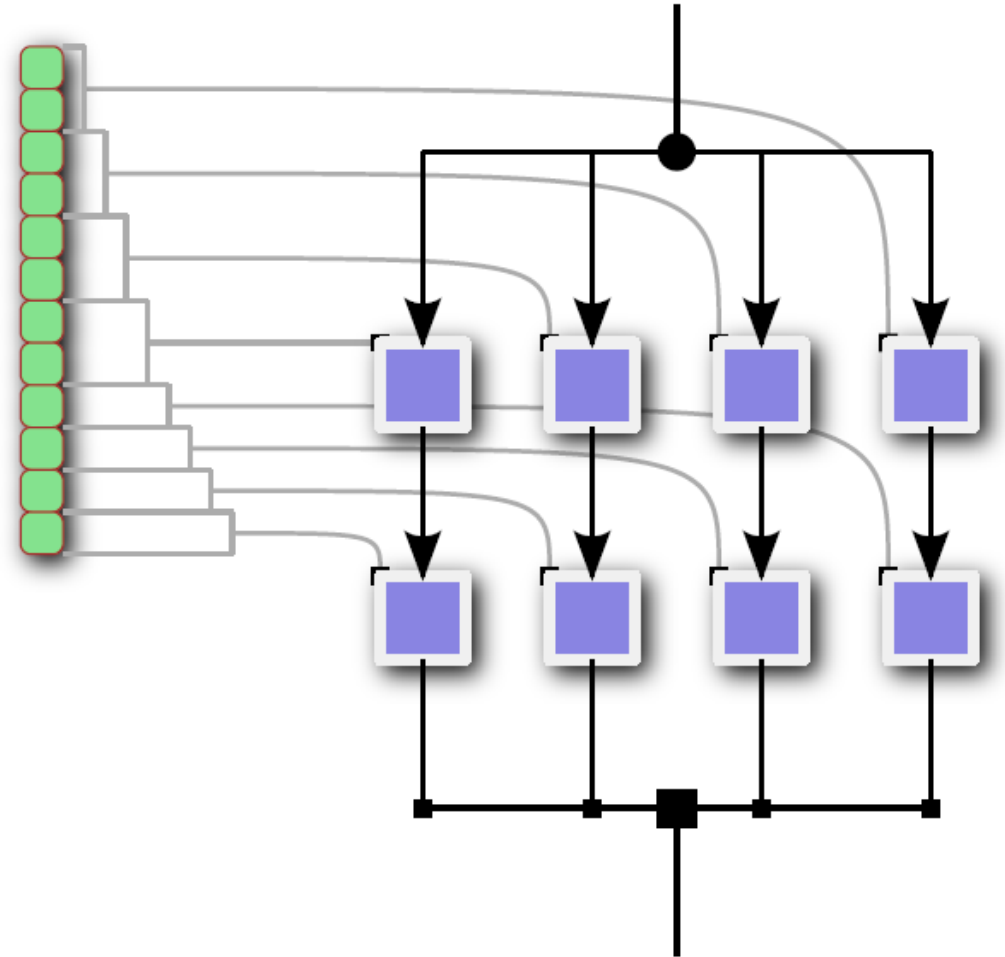
    return EXIT_SUCCESS;
}
```

Microsoft Visual Studio Debug Console

```
Value of "val" before the OpenMP parallel region: 123456789.
Value of "val" after the OpenMP parallel region: 3. Thread 3 was therefore the last one to modify it.
```

LOOP-CENTRIC PARALLELISM FOR-LOOPS IN OPENMP

- Simultaneously launch multiple threads
- Scheduler assigns loop **iterations** to threads
- Each thread processes **one** iteration at a time



LOOP-CENTRIC PARALLELISM FOR-LOOPS IN OPENMP

The OpenMP library will distribute **the iterations of the loop** following the *#pragma omp parallel for* across threads.

```
1 #pragma omp parallel for
2 for (int i = 0; i < n; i++) {
3     printf("Iteration %d is processed by thread %d\n",
4           i, omp_get_thread_num());
5     // ... iterations will be distributed across available threads...
6 }
```

LOOP-CENTRIC PARALLELISM FOR-LOOPS IN OPENMP

```
1 #pragma omp parallel
2 {
3   // Code placed here will be executed by all threads.
4
5   // Alternative way to specify private variables:
6   // declare them in the scope of pragma omp parallel
7   int private_number=0;
8
9   #pragma omp for
10  for (int i = 0; i < n; i++) {
11    // ... iterations will be distributed across available threads...
12  }
13  // ... code placed here will be executed by all threads
14 }
```


WORK SHARING — EXAMPLE 1

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

int main() {
    int i, n = 16;

    # pragma omp parallel for \
        shared(n) private(i)
    for (i = 0 ; i < n ; i++)
        printf("%d: %d\n",
            omp_get_thread_num(), i);

    return 0;
}
```

Run 1	Run 2	Run 3
1: 4	3: 12	0: 0
1: 5	3: 13	0: 1
1: 6	3: 14	0: 2
1: 7	3: 15	0: 3
2: 8	1: 4	3: 12
2: 9	1: 5	3: 13
2: 10	1: 6	3: 14
2: 11	1: 7	3: 15
0: 0	2: 8	1: 4
3: 12	2: 9	1: 5
3: 13	2: 10	1: 6
3: 14	2: 11	1: 7
3: 15	0: 0	2: 8
0: 1	0: 1	2: 9
0: 2	0: 2	2: 10
0: 3	0: 3	2: 11

WORK SHARING — EXAMPLE 1

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

int main() {
    int i, n = 16;

    # pragma omp parallel for \
        shared(n) private(i)
    for (i = 0 ; i < n ; i++)
        printf("%d: %d\n",
            omp_get_thread_num(), i);

    return 0;
}
```

“parallel for”: Split for loop iterations into multiple threads

“shared(n)”: Share variable n across threads

“private(i)”: Each thread has a separate i variable

Run 1	Run 2	Run 3
1: 4	3: 12	0: 0
1: 5	3: 13	0: 1
1: 6	3: 14	0: 2
1: 7	3: 15	0: 3
2: 8	1: 4	3: 12
2: 9	1: 5	3: 13
2: 10	1: 6	3: 14
2: 11	1: 7	3: 15
0: 0	2: 8	1: 4
3: 12	2: 9	1: 5
3: 13	2: 10	1: 6
3: 14	2: 11	1: 7
3: 15	0: 0	2: 8
0: 1	0: 1	2: 9
0: 2	0: 2	2: 10
0: 3	0: 3	2: 11

WORK SHARING — EXAMPLE 2

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

int main()
{
    int i;
    # pragma omp parallel
    {
        # pragma omp for
        for (i = 0 ; i < 8 ; i++)
            printf("A %d[%d]\n",
                omp_get_thread_num(), i);
        # pragma omp for
        for (i = 0 ; i < 8 ; i++)
            printf("B %d[%d]\n",
                omp_get_thread_num(), i);
    }

    return 0;
}
```

```
$ ./for
A 3[6]
A 3[7]
A 0[0]
A 0[1]
A 2[4]
A 2[5]
A 1[2]
A 1[3]
B 1[2]
B 1[3]
B 0[0]
B 0[1]
B 3[6]
B 3[7]
B 2[4]
B 2[5]
```

WORK SHARING — EXAMPLE 2

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

int main()
{
    int i;
    # pragma omp parallel
    {
        # pragma omp for
        for (i = 0 ; i < 8 ; i++)
            printf("A %d[%d]\n",
                omp_get_thread_num(), i);
        # pragma omp for
        for (i = 0 ; i < 8 ; i++)
            printf("B %d[%d]\n",
                omp_get_thread_num(), i);
    }

    return 0;
}
```

\$./for

A 3[6]

A 3[7]

A 0[0]

A 0[1]

A 2[4]

A 2[5]

A 1[2]

A 1[3]

B 1[2]

B 1[3]

B 0[0]

B 0[1]

B 3[6]

B 3[7]

B 2[4]

B 2[5]

Implicit barriers
at the end
of each loop

NESTED LOOPS

Outer loop's index private **by default**.

Inner loop's index **must** be set private.

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

int main()
{
    int i, j;
    # pragma omp parallel for private(j)
        for (i = 0 ; i < 4 ; i++)
            for (j = 0 ; j < 2 ; j++)
                printf("A %d[%d,%d]\n",
                    omp_get_thread_num(), i, j);

    return 0;
}
```

```
p$ ./for-nested
A 3[3,0]
A 3[3,1]
A 1[1,0]
A 1[1,1]
A 2[2,0]
A 2[2,1]
A 0[0,0]
A 0[0,1]
```

```
$ ./for-nested
A 3[3,0]
A 3[3,1]
A 1[1,0]
A 1[1,1]
A 0[0,0]
A 0[0,1]
A 2[2,0]
A 2[2,1]
```

Common Programming Errors

1. When the keyword `omp` or `parallel` of a `#pragma omp parallel for` directive is forgetfully omitted, the compiler will not give any errors, but the associated for loop will not be parallelized. Hence, the loop would be executed by a single thread. The same error occurs with the `#pragma omp parallel sections` directive. The code associated with multiple section directives would be executed by a single thread.
2. If the programmer is intended to distribute the execution of iterations of a for loop among a team of threads, he/she should use the `#pragma omp parallel for` directive. In this directive if the keyword `for` is omitted as shown in the following

code fragment, then the whole for loop would be executed separately by the two threads. Thus, the `printf()` would be executed 20 times.

```
#pragma omp parallel num_threads(2)
for(i=0;i<10;i++)
    printf("%d ", i);
```

If the `#pragma omp parallel for num_threads(2)` directive is used, the loop would be executed only once. The `printf()` would be executed 10 times. The execution of the iterations of the loop would be distributed among the two threads.

3. If the intention of the developer is to distribute the execution of for loop among multiple threads within a parallel region, he/she must use `#pragma omp for` directive rather than `#pragma omp parallel for` directive. If the `#pragma omp parallel for` is

used as shown in the following code fragment, the execution of loop iterations is not shared by the two threads. Each of the two threads execute the whole for loop separately and the func() will be executed 20 times.

```
#pragma omp parallel num_threads(2)
{
    #pragma omp parallel for
    for(int i=0; i<10; i++)
        func();
}
```

4. The number of threads cannot be changed inside a parallel region. In the following example, the parallel region consists of two threads. An attempt to change the number of threads using `omp_set_num_threads(4)` inside the parallel region

is ineffective and the number of threads will not be changed to 4.

```
#pragma omp parallel num_threads(2)
{
    omp_set_num_threads(4);
    //whatever code
}
```

5. By default, the number of threads in a parallel region will be equal to the number of cores in the processor. If the computation uses the value of number of threads, the result may vary depending on the number of cores in the machine. Hence, it is advisable to keep the behavior of the code independent of the number of threads.
6. Within the parallel region, the initial values of `threadprivate`, `private`, and `lastprivate` variables

LOOP SCHEDULING MODES IN OpenMP

- `schedule (static, [chunk])`
 - Contiguous ranges of iterations (chunks) of equal size
 - Low overhead, round robin assignment to threads, static scheduling
 - Default is one chunk per thread
- `schedule (dynamic, [chunk])`
 - Threads grab iterations resp. chunks
 - Higher overhead, but good for unbalanced work load
- `schedule (guided, [chunk])`
 - Dynamic schedule, shrinking ranges per step
 - Starts with large block, until minimum chunk size is reached
 - Good for computations with increasing iteration length (e.g. prime sieve test)

LOOP SCHEDULE — STATIC & DYNAMIC

STATIC



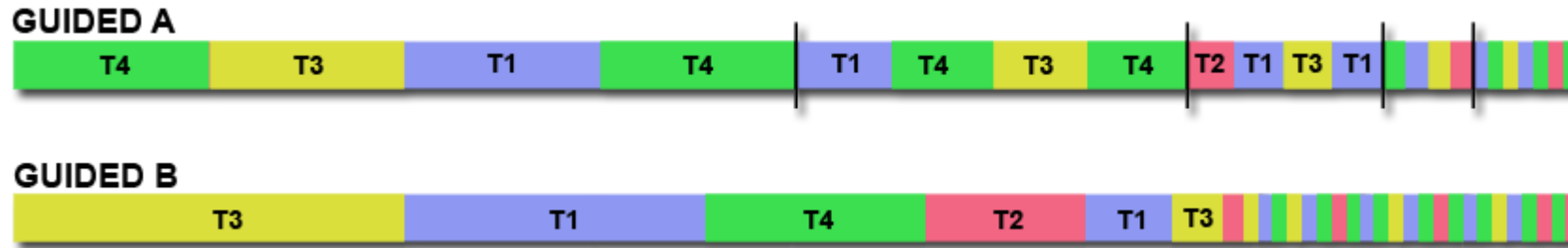
Loop iterations are **divided into pieces of size *chunk*** and then statically assigned to threads. **If *chunk* is not specified**, the iterations are evenly (if possible) divided **contiguously** among the threads.

DYNAMIC



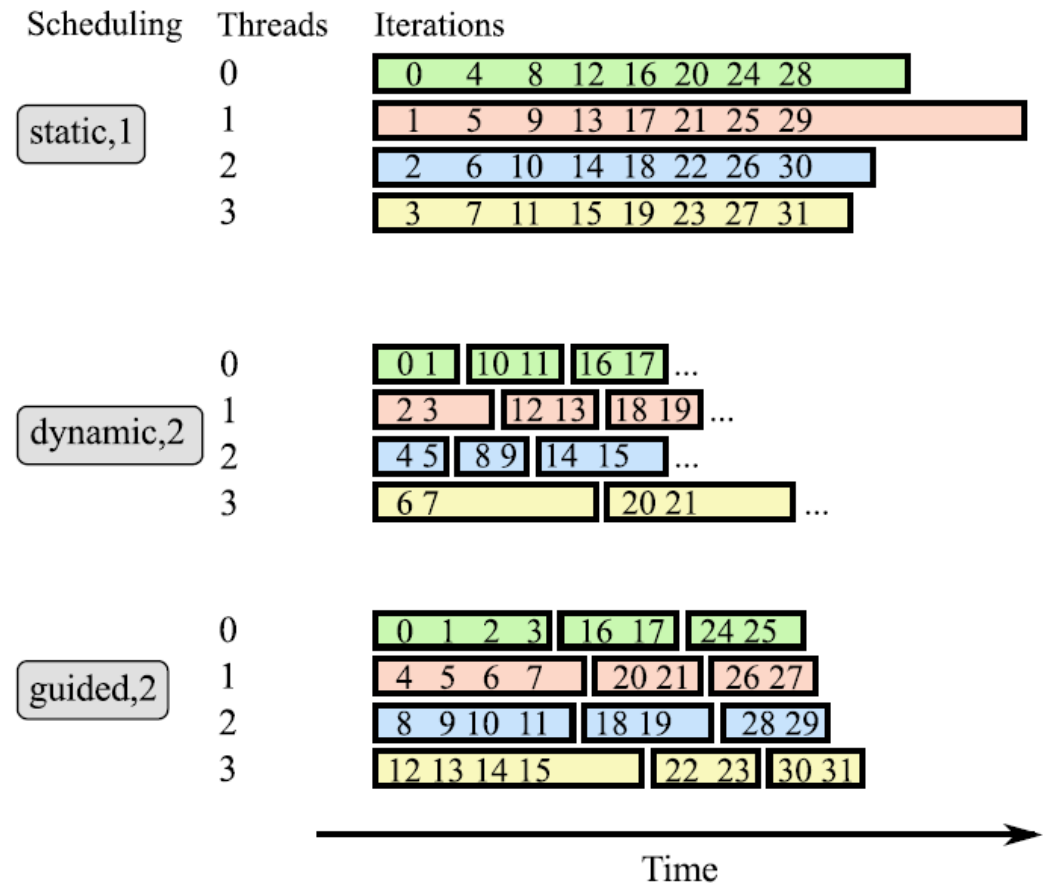
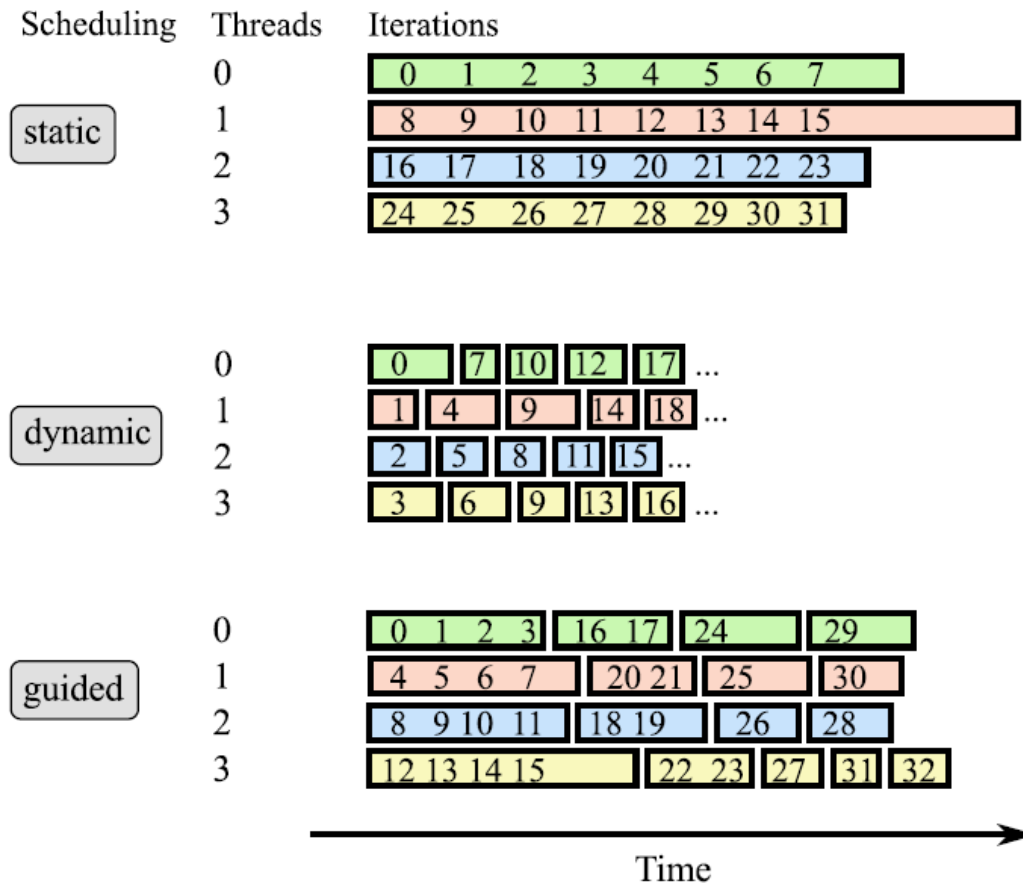
Loop iterations are **divided into pieces of size *chunk***, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

LOOP SCHEDULE — GUIDED



- Iterations are **dynamically** assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.
- The size of the initial block is proportional to: $\text{number_of_iterations} / \text{number_of_threads}$
- Subsequent blocks are proportional to $\text{number_of_iterations_remaining} / \text{number_of_threads}$
- The **chunk parameter defines the minimum block size**. The default chunk size is 1.
- Note: compilers differ in how GUIDED is implemented as shown in the "Guided A" and "Guided B" examples above.

LOOP SCHEDULING MODES IN OpenMP



RACE CONDITIONS

Occurs when 2 or more threads access the same memory address, and **at least one of these accesses is for writing**

=> Unpredictable Program Behavior

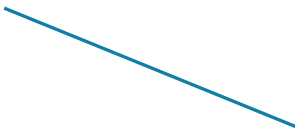
```
1 int total = 0;
2 #pragma omp parallel for
3 for (int i = 0; i < n; i++) {
4     // Race condition
5     total = total + 1;
6 }
```

RACE CONDITIONS

Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing

=> Unpredictable Program Behavior

```
1 int total = 0;
2 #pragma omp parallel for
3 for (int i = 0; i < n; i++) {
4     // Race condition
5     total = total + 1;
6 }
```



L5.1	Load	R1, @total
L5.2	Load	R2, @i
L5.3	Add	R3, R1, R2
L5.4	Store	R3, @total

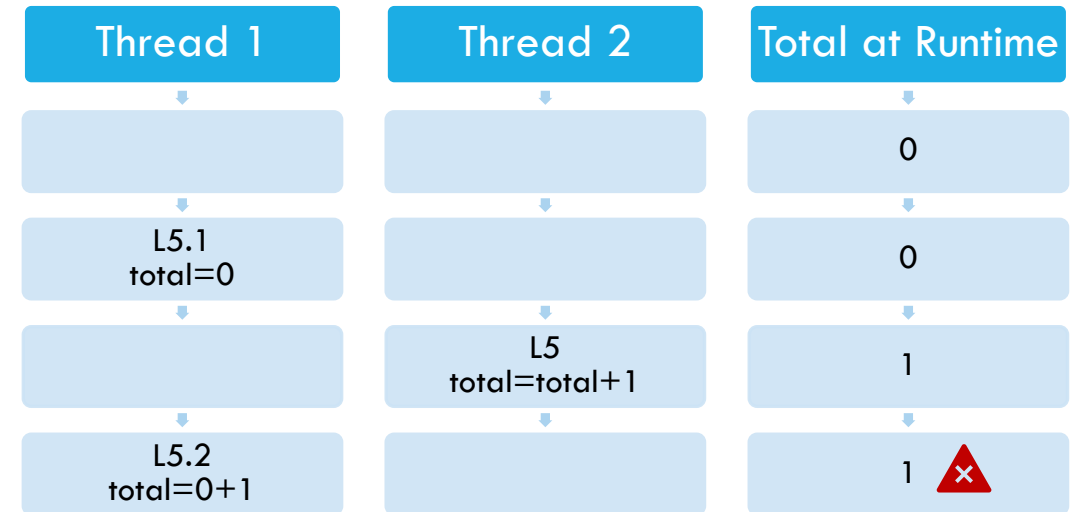
RACE CONDITIONS

Occurs when 2 or more threads access the same memory address, and at least one of these accesses is for writing

=> Unpredictable Program Behavior

```
1 int total = 0;
2 #pragma omp parallel for
3 for (int i = 0; i < n; i++) {
4     // Race condition
5     total = total + 1;
6 }
```

L5.1 Load R1, @total
L5.2 Load R2, @i
L5.3 Add R3, R1, R2
L5.4 Store R3, @total



SYNCHRONIZATION

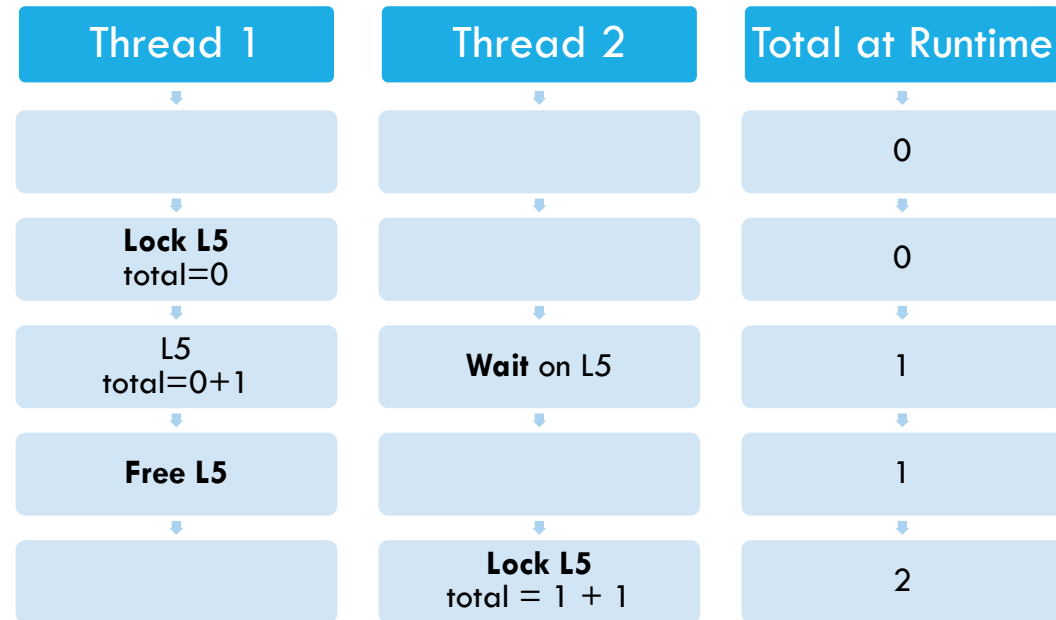
- **critical:** the enclosed code block **will be executed by only one thread at a time**, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- **Locks** provide a more flexible way than critical sections by allowing threads to **explicitly** acquire and release locks.
- **atomic:** the **memory update** (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
- **ordered:** the structured block is executed in the order in which iterations would be executed in a **sequential loop**.
- **barrier:** **each thread waits until all of the other threads** of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- **nowait:** specifies that threads completing assigned work can proceed **without waiting for all threads** in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the **work sharing construct**.

MUTEX - CRITICAL

Mutual exclusion conditions (mutexes) protect data races by serializing code.

=> **Multiple lines** (with multiple variables) declared as a *critical section*

```
1 int total = 0;
2 #pragma omp parallel for
3 for (int i = 0; i < n; i++) {
4   // Race free block
5   total = total + 1;
6 }
```



LOCKS

➤ This mechanism is closer to traditional wait/signal methods where a thread might wait for a lock to become available (signaled by another thread releasing the lock).

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

int main() {
    int counter = 0; // Shared variable among threads
    omp_lock_t lock;

    // Initialize the lock
    omp_init_lock(&lock);

    #pragma omp parallel num_threads(4) // Define the number of threads in the parallel region
    {
        // Wait until the lock is available and then acquire it
        omp_set_lock(&lock);

        // Critical section starts
        std::cout << "Thread " << omp_get_thread_num() << " is entering critical section." << std::endl;
        counter++; // Modify the shared variable
        std::cout << "Thread " << omp_get_thread_num() << " updated counter to " << counter << std::endl;
        // Critical section ends

        // Release the lock, allowing other waiting threads to enter the critical section
        omp_unset_lock(&lock);
    }

    // Destroy the lock
    omp_destroy_lock(&lock);

    std::cout << "Final counter value: " << counter << std::endl;
    return 0;
}
```

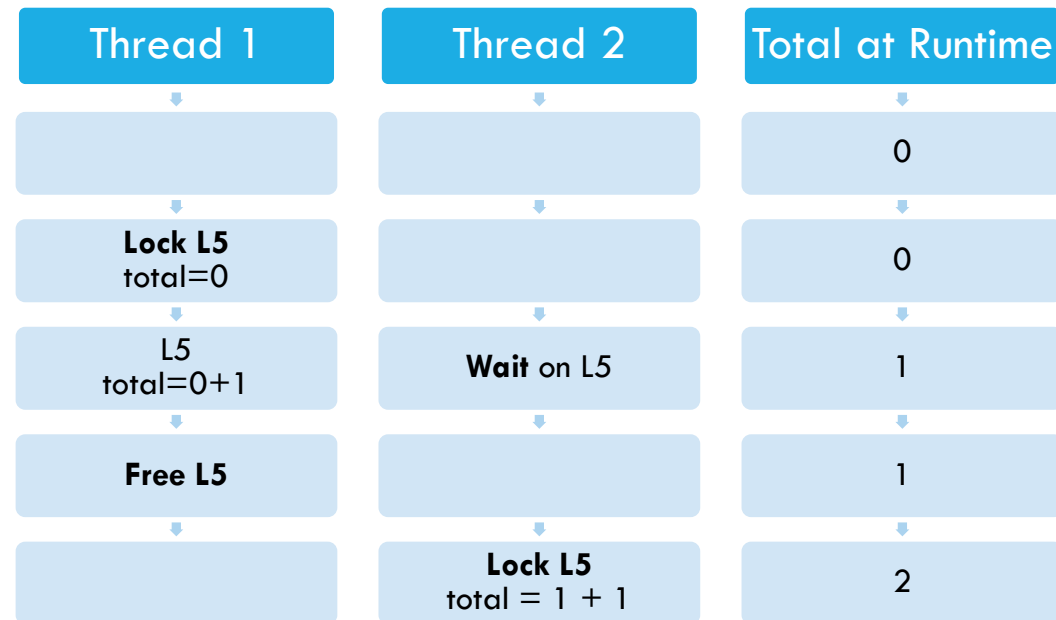
MUTEX - ATOMIC

Mutual exclusion conditions (mutexes) protect data races by serializing code.

Specific scalar instruction declared atomic

=> Good parallel codes minimize the use of mutexes

```
1 int total = 0;
2 #pragma omp parallel for
3 for (int i = 0; i < n; i++) {
4     // Race condition
5     #pragma omp atomic
6     total = total + 1;
7 }
```



MUTEX — THREAD-PRIVATE & SHARED with CRITICAL

```
#pragma omp parallel for
for (i = 0 ; i < N ; i++)
    sum += a[i]; // RACE: sum is shared
```

```
int main()
{
    double sum = 0.0, local, a[10];
    int i;
    for ( i = 0 ; i < 10 ; i++)
        a[i] = i * 3.5;
    # pragma omp parallel \
        shared(a, sum) private(local)
    {
        local = 0.0;
    #   pragma omp for
        for (i = 0 ; i < 10 ; i++)
            local += a[i];
    #   pragma omp critical
        { sum += local; }
    }
    printf("%g\n", sum);
    return 0;
}
```

Without the critical directive:

```
$ ./sum
157.5
$ ./sum
73.5
```

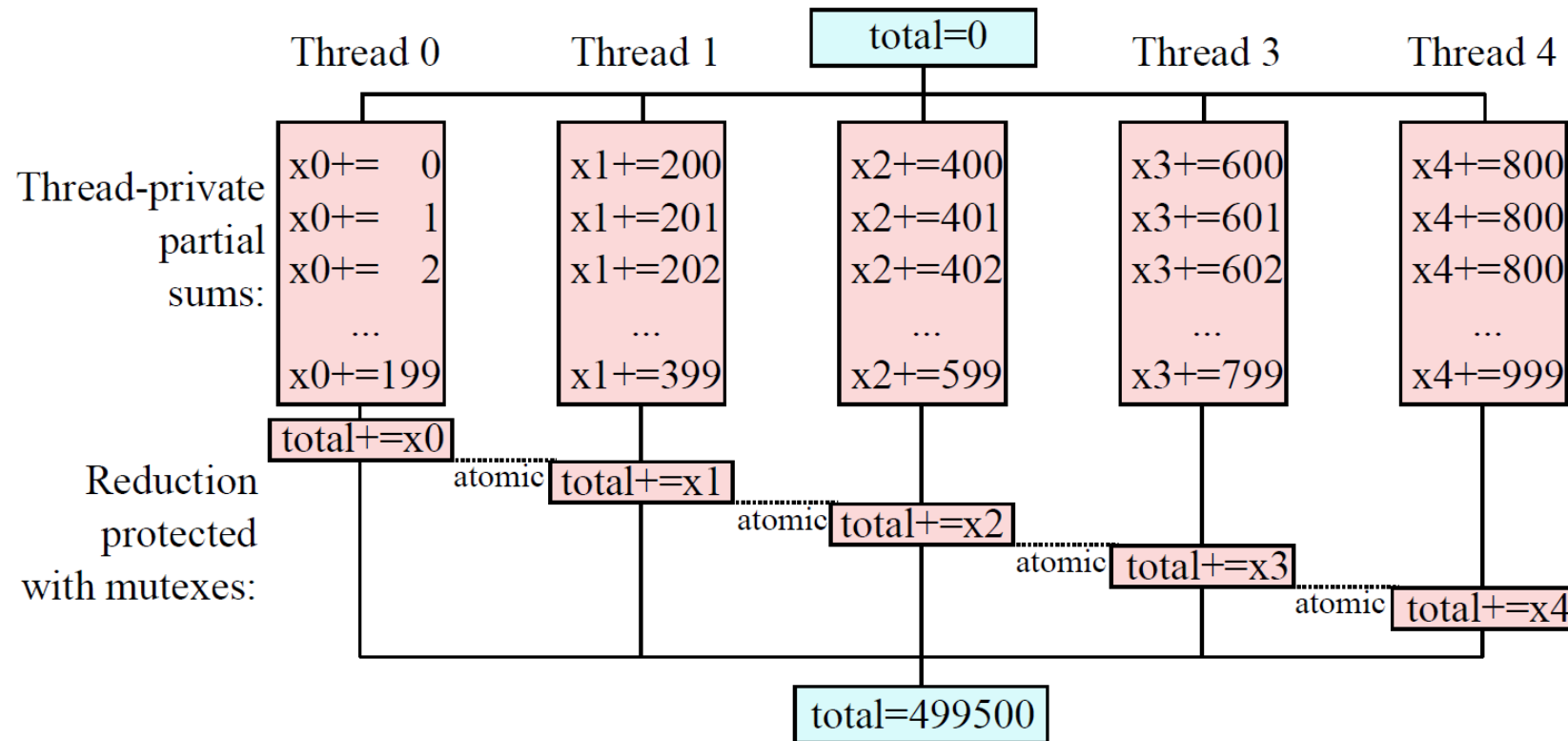
With #pragma omp critical:

```
$ ./sum
157.5
$ ./sum
157.5
```

MUTEX — THREAD-PRIVATE & SHARED with ATOMIC

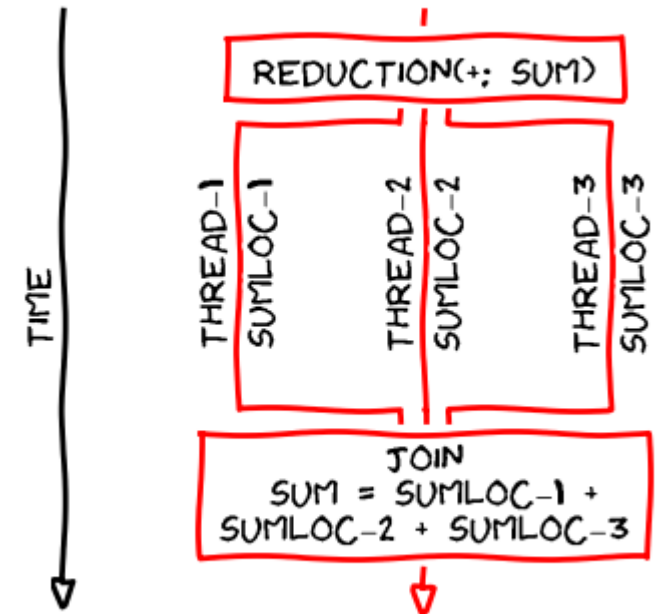
```
1  int total = 0;
2  #pragma omp parallel
3  {
4      int total_thr = 0;
5      #pragma omp for
6      for (int i=0; i<n; i++)
7          total_thr += i;
8
9      #pragma omp atomic
10     total += total_thr;
11
12 }
```

MUTEX — SHARED with REDUCTION



MUTEX — SHARED with REDUCTION

```
1 int total = 0;
2 #pragma omp parallel for reduction(+: total)
3 for (int i = 0; i < n; i++) {
4     total += 1;
5 }
```



EXAMPLE: SUMMING A VECTOR - REDUCTION

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

int main()
{
    double sum = 0.0, a[10];
    int i;
    for ( i = 0 ; i < 10 ; i++)
        a[i] = i * 3.5;

    # pragma omp parallel \
        shared(a) reduction(+:sum)
    {
        # pragma omp for
        for (i = 0 ; i < 10 ; i++)
            sum += a[i];
    }

    printf("%g\n", sum);
    return 0;
}
```

```
$ ./sum-reduction
157.5
```

EXAMPLE: DOT PRODUCT - REDUCTION

```
#include <stdio.h>

int main()
{
    double a[256], b[256], sum;
    int i, n = 256;

    for (i = 0 ; i < n ; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }

    sum = 0.0;

    for (i = 0 ; i < n ; i++ )
        sum += a[i]*b[i];

    printf("sum = %f\n", sum);
    return 0;
}
```


EXAMPLE: DOT PRODUCT - REDUCTION

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

int main()
{
    double a[256], b[256], sum;
    int i, n = 256;

    for (i = 0 ; i < n ; i++) {
        a[i] = i * 0.5;
        b[i] = i * 2.0;
    }

    sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for (i = 0 ; i < n ; i++ )
        sum += a[i]*b[i];

    printf("sum = %f\n", sum);
    return 0;
}
```

Other operators:

- Arithmetic: +, *, -, max, min
- Logical: &&, ||
- Bitwise AND, OR and XOR: &, |, ^

SINGLE REGION

- **single** is a clause that **must be used in a parallel region**; it tells OpenMP that the associated block must be executed **by one thread only**, albeit not specifying which one.
 - This block could be **executed many times** by the specified thread.
- The other threads will wait at the end of the associated block, at **an implicit barrier**, unless the single clause is accompanied with a **nowait** clause.
- The single clause must not be confused with the master or critical clause.

EXAMPLE

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

/**
 * @brief Illustrates how to use a single clause.
 * @details A parallel region is created, in which a certain part is executed by
 * every thread, and another part is executed only by a single thread.
 */
int main(int argc, char* argv[])
{
    // Use 4 threads when creating OpenMP parallel regions
    omp_set_num_threads(4);

    // Create the parallel region
    #pragma omp parallel
    {
        printf("[Thread %d] Every thread executes this printf.\n", omp_get_thread_num());

        #pragma omp barrier

        #pragma omp single
        {
            printf("[Thread %d] Only a single thread executes this printf, I happen to be the one picked.\n", omp_get_thread_num());
        }
    }

    return EXIT_SUCCESS;
}
```

SINGLE VS CRITICAL

- **single** specifies that a section of code should **be executed by single thread** (not necessarily the master thread)
- **critical** specifies that code is executed by **one thread at a time**.

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
    a++;
    #pragma omp critical
    b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

will print

```
single: 1 -- critical: 4
```

EXPLICIT BARRIER

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

int main()
{
    int x = 2;
    #pragma omp parallel shared(x)
    {
        int tid = omp_get_thread_num();
        if (tid == 0)
            x = 5;
        else
            printf("A: th %d: x=%d\n",
                  tid, x);
    }
    #pragma omp barrier

    printf("B: th %d: x=%d\n",
          tid, x);
}
return 0;
}
```

\$./barrier

A: th 2: x=2

A: th 3: x=2

A: th 1: x=2

B: th 1: x=5

B: th 0: x=5

B: th 2: x=5

B: th 3: x=5

Old value of x

\$./barrier

A: th 2: x=2

A: th 3: x=5

A: th 1: x=2

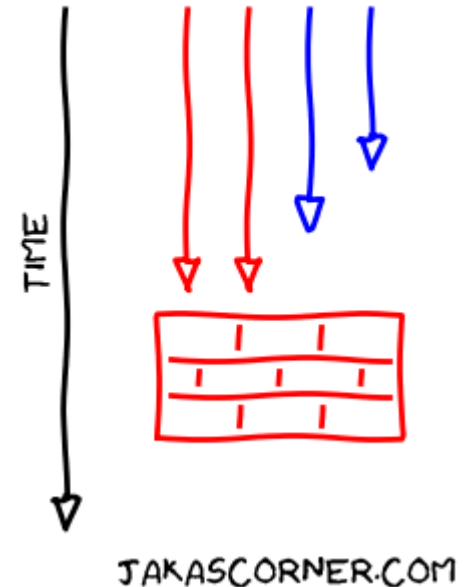
B: th 1: x=5

B: th 2: x=5

B: th 3: x=5

B: th 0: x=5

New value of x



It is a point in the execution of a program where threads wait for each other

SYNCRONIZATION - ORDERED

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

int main() {
    int i;

    #pragma omp parallel for ordered // ordered is not mandatory here,
                                   //but for performance in scheduling it is recommended to write it
    for(i = 0; i < 10; i++) {
        #pragma omp ordered
        {
            printf("Iteration %d is processed by thread %d\n", i, omp_get_thread_num());
        }
    }

    return 0;
}
```

```
Iteration 0 is processed by thread 0
Iteration 1 is processed by thread 0
Iteration 2 is processed by thread 0
Iteration 3 is processed by thread 1
Iteration 4 is processed by thread 1
Iteration 5 is processed by thread 1
Iteration 6 is processed by thread 2
Iteration 7 is processed by thread 2
Iteration 8 is processed by thread 3
Iteration 9 is processed by thread 3
```

SYNCRONAIZATION - NOWAIT

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

int main() {
    int i;
    float a[100], b[100];

    // Initialize arrays
    for (i = 0; i < 100; i++) {
        a[i] = i * 1.0f;
        b[i] = i * 2.0f;
    }

    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i = 0; i < 100; i++) {
            a[i] = a[i] * 2.0f;
        }

        // No implicit barrier here because of the nowait clause
        #pragma omp for
        for (i = 0; i < 100; i++) {
            b[i] = b[i] + a[i];
        }
    }

    // Print a few results
    for (i = 98; i < 100; i++) {
        printf("a[%d] = %f, b[%d] = %f\n", i, a[i], i, b[i]);
    }

    return 0;
}
```

```
a[98] = 196.000000, b[98] = 392.000000
a[99] = 198.000000, b[99] = 396.000000
```

In OpenMP, the `nowait` clause is not valid on a `parallel for` directive alone. Instead, **nowait is typically used with combined constructs like `#pragma omp for` (without the `parallel`) or with `sections`** where separate work-sharing constructs may immediately follow each other without needing synchronization.

SECTIONS - SECTION

The **section** construct is one way to distribute different tasks to different threads.

- The **sections** construct distributes the blocks/tasks between existing threads.
- The requirement is that **each block must be independent of the other blocks**.
- Then each thread executes one block at a time.
- Each block is executed only once by one thread.

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            // structured block 1
        }

        #pragma omp section
        {
            // structured block 2
        }

        #pragma omp section
        {
            // structured block 3
        }

        ...
    }
}
```


SECTIONS - SECTION

If there is only one `sections` construct inside the `parallel` construct

```
#pragma omp parallel
{
    #pragma omp sections
    {
        ...
    }
}
```

we can simplify both constructs into the combined construct

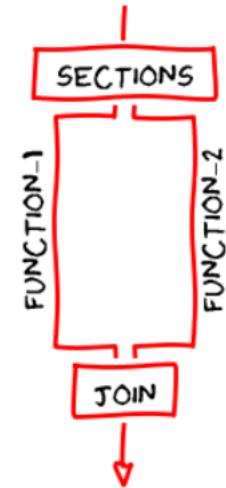
```
#pragma omp parallel sections
{
    ...
}
```

```
int main()
{
    #pragma omp parallel sections
    {
        #pragma omp section
        function_1();

        #pragma omp section
        function_2();
    }

    return 0;
}
```

TIME



JAKASCORNER.COM

OMP_GET_WTIME();

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

```
int main() {
```

```
    double dtime = omp_get_wtime(); //value in seconds
```

```
    //run some code
```

```
    dtime = omp_get_wtime() - dtime;
```

```
}
```

EXTRAS — ON YOUR OWN

Directives

- task

Clauses

- taskwait
- flush

Data scope

- Default, copyin, copyprivate