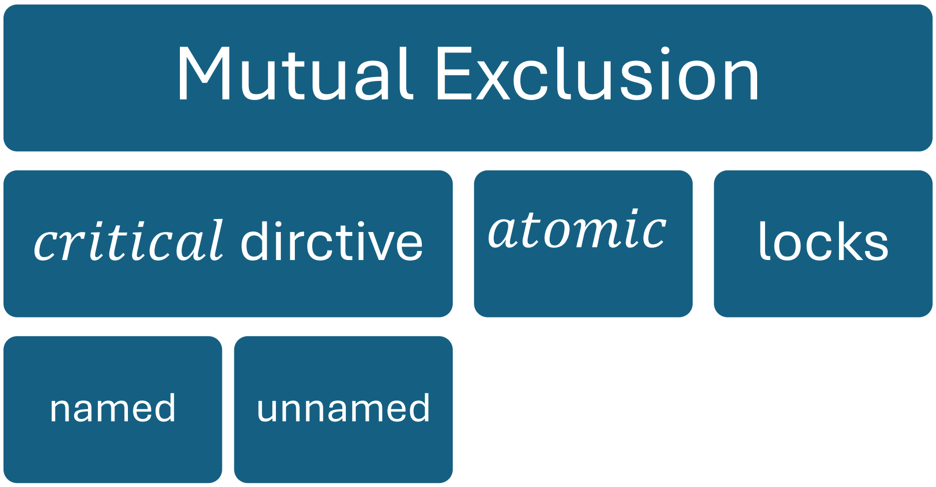
Introduction to Parallel Computing

Shared-memory programming with OpenMP

Table of Contents

Critical directives, *atomic* directives, or locks?

- So far, we introduced **three** basic mutual exclusion mechanisms:



Mechanism	Properties	Syntax	Group regions under one directive?
atomic	Simple and the fastest	#pragma omp atomic x <op>= <expression>; x++; --x;	**Yes**

***critical* (unnamed)**	Easier than locks	#pragma omp critical
		{
		…
		}
			Yes

***critical* (named)**	-	#pragma omp critical(name)
		{
		…
		}
			No

Locks	Better for data structures	omp_set_lock(lock);
		…
		omp_unset_lock(lock);
			No

- Critical regions specified by *atomic* or *critcal* (unnamed) directives **may** treat all their regions as one block.
- For example:

```
#pragma omp parallel
{
    #pragma omp atomic
    var1 = var1 + 1;  // Atomic operation on var1

    #pragma omp atomic
    var2 = var2 + 1;  // Atomic operation on var2
}
```

- **In one implementation**, the runtime might enforce exclusive access, meaning that the operations on $var1$ and $var2$ cannot happen at the same time, even though they operate on different variables.
- **In another implementation**, the runtime might allow these operations to run concurrently if they access different variables, as there's no dependency or conflict between the two operations.

- The same applies if using (**unnamed**) $critical$ directive:

```
#pragma omp parallel
{
    #pragma omp critical
    {
        var1 = var1 + 1;
    }

    #pragma omp critical
    {
        var2 = var2 + 1;
    }
}
```

- **Named** $critical$ directive and **locks** avoid that.

```
#pragma omp parallel
{
    #pragma omp critical(section1)
    {
        var1 = var1 + 1;  // Atomic operation on var1
    }

    #pragma omp critical(section2)
    {
        var2 = var2 + 1;  // Atomic operation on var2
    }
}
```

Some caveats

1. **Don't mix** the different types of mutual exclusion for a **single** critical section.
   - The code below uses two mutual exclusive mechanisms for the variable x.

```
#   pragma omp atomic          #   pragma omp critical
    x += f(y);                     x = g(x);
```

   - It's possible that the two sections get executed **concurrently**, leading to **incorrect** results.
   - You can either: **use** $critical$ directive **for both sections** or **rewrite** $g()$ to have the **form** required by the $atomic$ directive.
2. There is no **guarantee of fairness** in mutual exclusion constructs.
   - For example, the code below may allow one thread to **always** be executing the function g, thus preventing other threads from accessing it.

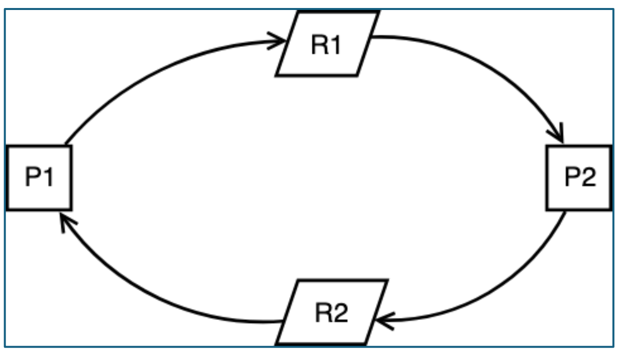
```
    while (1) {
        . . .
#       pragma omp critical
        x = g(my_rank);
        . . .
    }
```

- This won't happen if the *while* loop **terminates**.

3. It can be dangerous to "**nest**" mutual exclusion constructs.

```
#   pragma omp critical(one)
    y = f(x);
    . . .
    double f(double x) {
#       pragma omp critical(two)
        z = g(x);   /* z is global */
        . . .
    }
```

- This will cause **deadlock**: a situation in which a group of threads are waiting for each other thread to finish.



- If a thread is executing the first block, it won't be able to **enter the second block**. At the same time, it **will not leave the first block** until it proceeds to the second block.
- Deadlocks can occur as following:

Time	Thread *u*	Thread *v*
0	Enter crit. sect. `one`	Enter crit. sect. `two`
1	Attempt to enter `two`	Attempt to enter `one`
2	Block	Block

Tasking

- **Tasking** is a functionality that applies to some problems that are **hard to parallelize** using the previous OMP techniques.
  - Programs that include $while$, $do - while$ loops, or unbounded for loop.
  - Programs that include recursive algorithms, such as graph traversal.
- It allows us to specify **independent units of computation** with the $task$ directive:
  - `#pragma omp task`
  - When a thread reaches a block of code with this directive, a new task is generated by the OpenMP run-time that will be scheduled for execution.
- Tasks may not **start immediately**, as other tasks may be already **pending execution**.
- Tasks must be launched from within a $parallel$ region but generally by only **one thread** of the team.

```
# pragma omp parallel
# pragma omp single
  {
      ...
#     pragma omp task
      ...
  }
```

  - The $parallel$ directive creates a **team of threads**.
  - The $single$ directive tells the runtime to **only** launch tasks from a **single thread**.
  - If the $single$ directive is **omitted**, subsequent $task$ instances will be **launched multiple times**, one for each thread in the team.
- To explain this further:

#pragma omp single	#pragma omp task
• Specify that a block of code should be **executed by only one** thread in the team • Used for tasks like **initialization**, **I/O operations**, or **creating tasks** that other threads can execute later • The parallelism comes from what happens ***after*** the $single$ block	• Define **independent units of work** (tasks) that can be executed by any available thread in the team • Each task is a **self-contained block** of code that can **run concurrently** with other tasks or code in the program • When a thread encounters $\#pragma\ omp\ task$, it creates a task and **adds it to a pool**

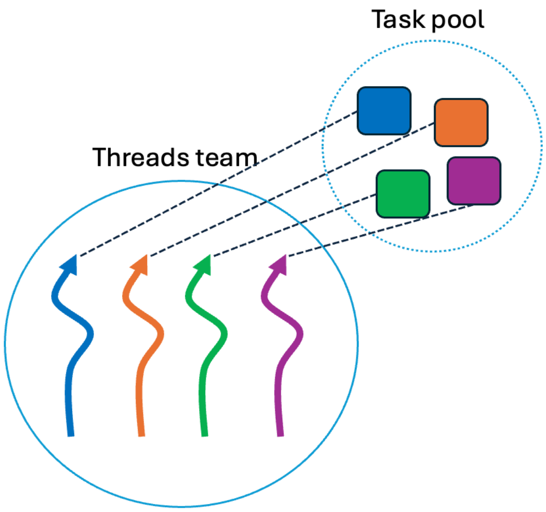
- Try these programs and notice their outputs

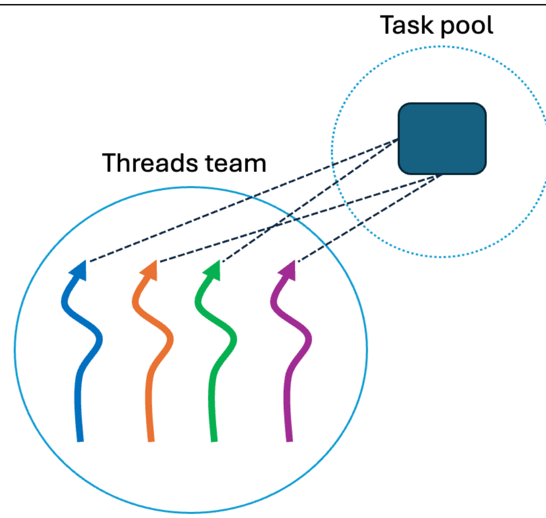
```c
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel num_threads(4)
    {
        for (int i = 0; i < 4; i++) {
#pragma omp task
            printf("I am thread no. %d.\n", omp_get_thread_num());
        }
    }
    return 0;
}
```

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

int main() {
#pragma omp parallel num_threads(4)
#pragma omp single
    {
        for (int i = 0; i < 4; i++) {
#pragma omp task
            printf("I am thread no. %d.\n", omp_get_thread_num());
        }
    }
    return 0;
}
```





- Note that the *task* directive is inside a *parallel* construct. Each thread in the team:
  - encounters the *task* construct,
  - creates the corresponding task and
  - either executes the task immediately or defer its execution to one of the other threads in the team
- When using *single* directive, **only one task is created**, and any thread in the team can execute the task.

- Consider this recursive Fibonacci program

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

long fib(long n) {
    long i = 0; long j = 0;

    if (n <= 1) {
        return n;
    }
    i = fib(n - 1);
    j = fib(n - 2);
    return i + j;
}

int main(int argc, char *argv[]) {
    double start = omp_get_wtime();
    long res = fib(45);
    double end = omp_get_wtime();
    printf("fib(45) = %ld\n", res);
    printf("elapsed time = %lf\n", end - start);
    return 0;
}
```

- We can parallelize the program using *task* directive

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

long fib(long n) {
    long i = 0; long j = 0;

    if (n <= 1) {
        return n;
    }
#pragma omp task shared(i) firstprivate(n) if (n>40)
    i = fib(n - 1);

#pragma omp task shared(j) firstprivate(n) if (n>40)
    j = fib(n - 2);

#pragma omp taskwait
    return i + j;
}

int main(int argc, char *argv[]) {
    long res;
    double start = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp single
        res = fib(45);
    }

    double end = omp_get_wtime();
    printf("fib(45) = %ld\n", res);
    printf("elapsed time = %lf\n", end - start);
    return 0;
}
```

- Or better use the serial function as an auxiliary function.

```
#include <stdio.h>
#include <omp.h>
long serial_fib(long n) {
    long i=0, j=0;

    if (n <= 1) return n;
    i = serial_fib(n-1);
    j = serial_fib(n-2);
    return i+j;
}
long fib(long n) {
    long i=0, j=0;

    if (n <= 20) return serial_fib(n);

#pragma omp task firstprivate(n) shared(i) if(n>40)
    i = fib(n-1);
#pragma omp task firstprivate(n) shared(j) if(n>40)
    j = fib(n-2);
#pragma omp taskwait
    return i+j;
}
int main(int argc, char *argv[]) {
    long res;
    double start = omp_get_wtime();
#pragma omp parallel
    {
#pragma omp single
        res = fib(45);
    }

    double end = omp_get_wtime();
    printf("fib(45) = %ld\n", res);
    printf("elapsed time = %lf\n", end - start);
    return 0;
}
```

- $shared(i), shared(j)$ tells the runtime that i and j are **shared variables among the tasks**. This ensures correct computations.
- $if\ (n > 40)$ this will ensure that task creation is restricted when $n > 40$.
  - Try changing the condition to $n > 20$, $n > 30$ and remove it to see the performance.
  - **Altering the condition or removing it will cause very bad performance.**
- $firstprivate$ specifies that **each thread should have its own instance of a variable,** and that the **variable should be initialized with the value of the variable**, because it exists before the parallel construct.
- The $taswait$ directive forces the subtasks (threads) to complete.

Thread safety

- A block of code is **thread-safe** if it can be simultaneously executed by multiple threads **without causing problems**.
- Non-thread-safe functions in C are very common and **very hard to detect their errors**.
- Example, suppose we write a program that tokenizes a file:
    - E.g., “Hello abc 123” → “Hello”, “abc”, “123”
    - Suppose we have a **file** that include **lines of text**.
    - Assign the lines to threads in a **round robin fashion**: line 0 → thread 0, line 1 → thread 1, and so on.
    - Read the file into **an array of strings**, with **one line per string**.
    - Use the $\boldsymbol{parallel\ for}$ directive with $\boldsymbol{schedule(static, 1)}$ clause to divide the lines among the threads.
    - Use the $\boldsymbol{strtok}$ function in $\boldsymbol{string.h}$ to tokenize a line.

```
char* strtok(
      char*          string        /* in/out */,
      const char*    separators    /* in     */);
```

  - The first argument is the **string to be tokenized**
  - The second argument is the **separator**, which are spaces, tabs, and new lines.
  - When calling $strtok$ for the first time, it will take the string to be tokenized as first argument. But **subsequent calls will take $NULL$** for the first argument to return each token of the input string.
  - The idea is that in the first call, $strtok$ **caches a pointer** to $string$, and for subsequent calls it **returns successive tokens** taken from the **cached copy**.

- Full program

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <omp.h>
const int MAX_LINES = 1000;
const int MAX_LINE = 80;

void Get_text(char *lines[], int *line_count_p) {
    char *line = malloc(MAX_LINE * sizeof(char));
    int i = 0;
    char *fg_rv = fgets(line, MAX_LINE, stdin);

    while (fg_rv != NULL) {
        lines[i++] = line;
        line = malloc(MAX_LINE * sizeof(char));
        fg_rv = fgets(line, MAX_LINE, stdin);
    }
    *line_count_p = i;}

void Tokenize(char *lines[], int line_count, int thread_count) {
    int my_rank, i, j;
    char *my_token;

#  pragma omp parallel num_threads(thread_count) \
    default(none) private(my_rank, i, j, my_token) shared(lines, line_count)
    {
        my_rank = omp_get_thread_num();
#      pragma omp for schedule(static, 1)
        for (i = 0; i < line_count; i++) {
            printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
            j = 0;
            my_token = strtok(lines[i], " \t\n");
            while (my_token != NULL) {
                printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
                my_token = strtok(NULL, " \t\n");
                j++;
            }
            if (lines[i] != NULL)
                printf("Thread %d > After tokenizing, my line = %s\n",
                       my_rank, lines[i]);
        } }}

/*-------------------------------------------------------------------*/
int main(int argc, char *argv[]) {
    int thread_count=4, i;
    char *lines[1000];
    int line_count;

    printf("Enter text\n");
    Get_text(lines, &line_count);
    Tokenize(lines, line_count, thread_count);

    for (i = 0; i < line_count; i++)
        if (lines[i] != NULL) free(lines[i]);

    return 0;
} /* main */
```

- Run:
  `gcc-14 p5_26_Tokenizer.c -fopenmp`
  `./a.out < Tokenizer_in`

Where $Tokenizer\_in$ is a file that contains this text:

```
Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old
```

- Experiment the program with 1, 2, and 4 threads:

With 1 thread, the output is correct

```
Thread 0 > line 0 = Pease porridge hot.
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = hot.
Thread 0 > After tokenizing, my line = Pease
Thread 0 > line 1 = Pease porridge cold.
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = cold.
Thread 0 > After tokenizing, my line = Pease
Thread 0 > line 2 = Pease porridge in the pot
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = in
Thread 0 > token 3 = the
Thread 0 > token 4 = pot
Thread 0 > After tokenizing, my line = Pease
Thread 0 > line 3 = Nine days old
Thread 0 > token 0 = Nine
Thread 0 > token 1 = days
Thread 0 > token 2 = old
Thread 0 > After tokenizing, my line = Nine
```

With 2 threads, the output is correct

```
Thread 0 > line 0 = Pease porridge hot.
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = hot.
Thread 0 > After tokenizing, my line = Pease
Thread 0 > line 2 = Pease porridge in the pot
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = in
Thread 0 > token 3 = the
Thread 0 > token 4 = pot
Thread 0 > After tokenizing, my line = Pease
Thread 1 > line 1 = Pease porridge cold.
Thread 1 > token 0 = Pease
Thread 1 > token 1 = porridge
Thread 1 > token 2 = cold.
Thread 1 > After tokenizing, my line = Pease
Thread 1 > line 3 = Nine days old
Thread 1 > token 0 = Nine
Thread 1 > token 1 = days
Thread 1 > token 2 = old
Thread 1 > After tokenizing, my line = Nine
```

With 4 threads, the output is **incorrect** (*the error may not occur from the first run!)*

```
Thread 1 > line 1 = Pease porridge cold.
Thread 2 > line 2 = Pease porridge in the pot
Thread 0 > line 0 = Pease porridge hot.
Thread 0 > token 0 = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = hot.
Thread 0 > After tokenizing, my line = Pease
Thread 3 > line 3 = Nine days old
Thread 3 > token 0 = Nine
Thread 3 > token 1 = days
Thread 3 > token 2 = old
Thread 3 > After tokenizing, my line = Nine
Thread 1 > token 0 = Pease
Thread 1 > After tokenizing, my line = Pease
Thread 2 > token 0 = Pease
Thread 2 > After tokenizing, my line = Pease
```

- The problem is that $strtok$ caches its input. It declares a variable to have $static$ modifier.
    - Static variables in C **retain their value between function calls**, are initialized only once, and **exist for the duration of the program**.
- Cached strings will be shared among the threads, not private.
- So, a call to $strtok$ can **overwrite** the contents of the previous calls.
- **The $strtok$ function is therefore not thread-safe:** if multiple threads call it simultaneously, the output it produces may not be correct.
- To fix this issue, we must use a **thread-safe function**, $strtok\_r$.

```
char* strtok_r(
      char*          string       /* in/out */,
      const char*    separators   /* in     */,
      char**         saveptr_p    /* in/out */);
```

- $strtok\_r$ is the **reentrant** version of $strtok$.
- **Reentrant** means that the function can be **safely interrupted** and then called again before the interrupted process can finish.
    - This is used as a synonym for **thread-safe**.
- $strtok\_r$ has a third argument, $saveptr\_p$. It allows the function to **preserve the context** of what has and hasn't been tokenized.

- To fix the previous program, replace *strtok* with *strtok_r*.

```
void Tokenize(char *lines[], int line_count, int thread_count) {
    int my_rank, i, j;
    char *my_token, *saveptr;

#   pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token, saveptr) \
      shared(lines, line_count)
    {
        my_rank = omp_get_thread_num();
#       pragma omp for schedule(static, 1)
        for (i = 0; i < line_count; i++) {
            printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
            j = 0;
            my_token = strtok_r(lines[i], " \t\n", &saveptr);
            while (my_token != NULL) {
                printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
                my_token = strtok_r(NULL, " \t\n", &saveptr);
                j++;
            }
            if (lines[i] != NULL)
                printf("Thread %d > After tokenizing, my line = %s\n",
                       my_rank, lines[i]);
        }
    }
}
```

- Run the previous program with 4 threads and check the output

```
Thread 0 > line 0 = Pease porridge hot.
Thread 0 > token 0 = Pease
Thread 3 > line 3 = Nine days old
Thread 3 > token 0 = Nine
Thread 2 > line 2 = Pease porridge in the pot
Thread 2 > token 0 = Pease
Thread 2 > token 1 = porridge
Thread 2 > token 2 = in
Thread 2 > token 3 = the
Thread 2 > token 4 = pot
Thread 1 > line 1 = Pease porridge cold.
Thread 3 > token 1 = days
Thread 3 > token 2 = old
Thread 3 > After tokenizing, my line = Nine
Thread 1 > token 0 = Pease
Thread 1 > token 1 = porridge
Thread 1 > token 2 = cold.
Thread 1 > After tokenizing, my line = Pease
Thread 0 > token 1 = porridge
Thread 0 > token 2 = hot.
Thread 0 > After tokenizing, my line = Pease
Thread 2 > After tokenizing, my line = Pease
```