

Fine-grained Parallel Programming

The OpenMP Tasking Model

Luis Miguel Pinho

July 2022

Preliminary reading

- This module requires understanding of OpenMP.
- Although not required, preliminary analysis of the following online resources help with understanding the concepts that will be discussed in the classes:
 - OpenMP in Small Bites, https://hpc-wiki.info/hpc/OpenMP_in_Small_Bites

OpenMP Basics

Annotated code

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

int main() {

    #pragma omp parallel
    printf("Hello world\n");

    return 0;
}
```

Annotation in code are comments for non-openmp compilers

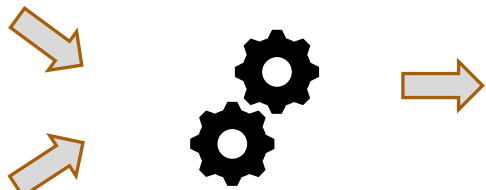
Goal is that the same algorithm can execute sequentially

OpenMP libraries

[illegible]

OpenMP library provides both functions to be used by the programmer, as well as runtime support hidden to programmers

OpenMP aware
compiler, e.g.
gcc -fopenmp



Executable code

```

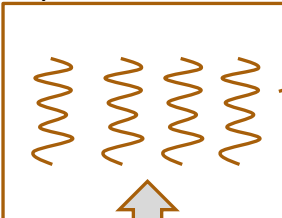
0101010010101001001001001010100100100101
0100010100010001001010110101001001001001
0101001010010010010101010101001001001001
0101101010010010010010010010010101001010
0101010001010010010010010101010101001010
001000100010101000100010101010010101001
01001011010101001010010100100100101001
010010101001001001010010010010101010100
010100100101001010010100101001010100101
00100100101010101010010100100100101001
10010100101010100101001010010101010101
10010100100100100101001001001010101001
0101001010010101000000000000000000000000

```

Program environment

OMP_NUM_THREADS
OMP_DYNAMIC

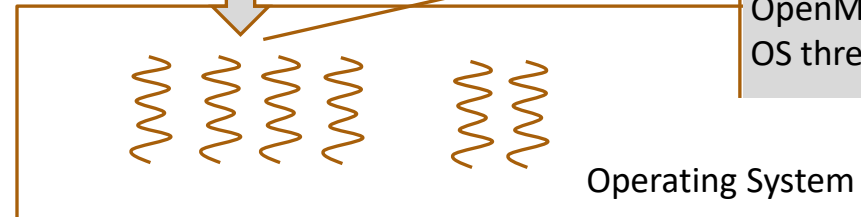
OpenMP runtime



The OpenMP runtime manages the creation and execution of the OpenMP threads

The specification does not specify how the runtime is implemented, only behavior. This includes how OpenMP threads are mapped to OS threads (usually 1-to-1)

Operating System



OpenMP Basics

- First versions of OpenMP had a thread-centric execution model based on the concept of parallel regions and fork-join computation
 - Create a team of threads and partition the work among them
 - This has its roots in the “array-based world of scientific computing”
 - Recursion and dynamic structures were not considered

OpenMP Basics

- It is possible, but ...

```
void traverse_tree(node_t *n)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        process(n);

        #pragma omp section
        if(n->left)
            traverse_tree(n->left);

        #pragma omp section
        if(n->right)
            traverse_tree(n->right);
    }
}
```

traverse_tree(root);

Nested parallel regions

There is a barrier here

OpenMP Tasking

- In version 3 of OpenMP a new model was introduced
 - Follows the model of specifying what to parallelize, not how to parallelize
- The concept of a “task”
 - Block of code that can safely execute in parallel with the surrounding code
 - Programmer annotates these blocks with the task pragma
- Tasks are not executing units
 - They are queued for execution by the threads of the parallel team
 - The moment, and order, tasks are executed depends on the mapping of tasks to threads and scheduling of threads (can be immediate or delayed)
 - Tasks can be forced to complete though synchronization
- OpenMP defines behaviour, not implementation

OpenMP Tasking

```
#pragma omp parallel  
{
```

When a parallel region is created the runtime creates a team of threads

```
    #pragma omp task  
    { code }
```

The programmer specifies that the code can execute in parallel to the rest of the code

```
    #pragma omp task  
    { code }
```

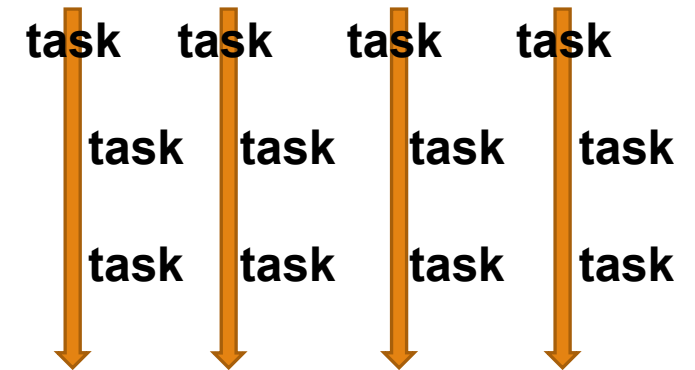
The programmer specifies that the code can execute in parallel to the rest of the code

```
    // code
```

```
}
```

The specification defines that there is an implicit task in the parallel region (in each thread of the team)

How many tasks are in this code?

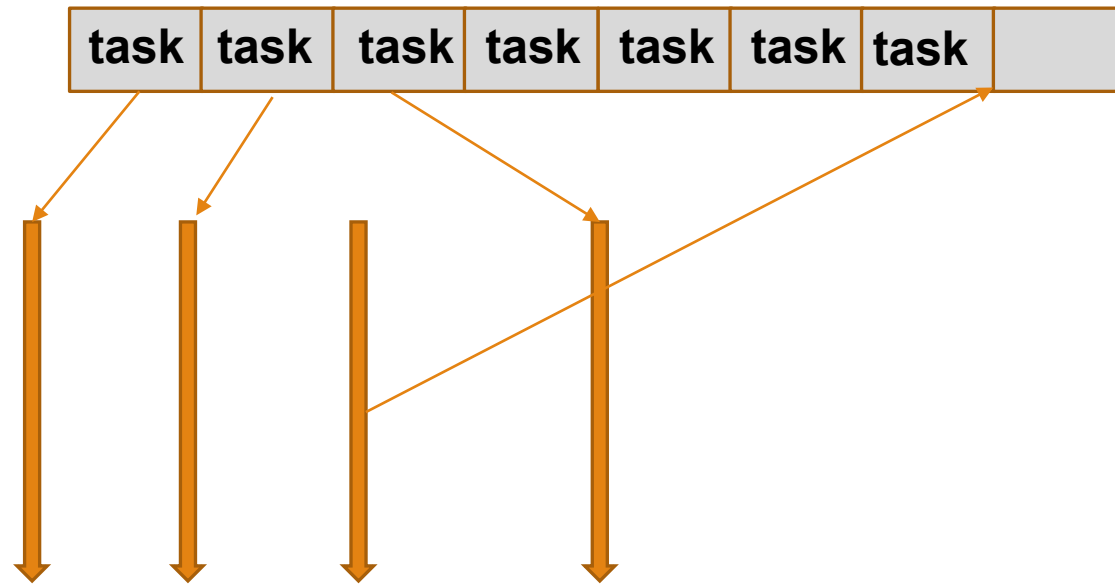


OpenMP Tasking

- Multiple threads are creating tasks, which are themselves later executed by the same threads
 - Very confusing, therefore the pattern is that only one thread starts creating the tasks, using a single directive
 - One thread creates the tasks, which are placed in the thread pool queue, the other threads task work from the queue

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

        // ...
    }
}
```



OpenMP tasking

- Revisiting the tree traversal

```
void traverse_tree_internal(node* n)
{
    #pragma omp task
    process(n);

    #pragma omp task
    if(n->left)
        traverse_tree_internal(n->left);

    #pragma omp task
    if(n->right)
        traverse_tree_internal(n->right);
}

void traverse_tree(node* n)
{
    #pragma omp parallel
    #pragma omp single
    traverse_tree_internal(n);
}
```

OpenMP Tasking

- Tasks are composed of:
 - code to execute
 - data environment
 - internal control variables
- Task construct
 - Task directive plus the structured block
- Task
 - The code and instructions for allocating the data
 - Created when a thread encounters a task construct
- Task region
 - The dynamic sequence of instructions produced by the execution of a task by a thread

OpenMP Tasking

- Task synchronization
 - Tasks are asynchronous, created by a thread, and placed in a queue
 - Depending on the algorithm, the task may take long to execute
 - It is necessary to guarantee that it is completed when its result is needed
- Tasks are guaranteed to be completed
 - At thread barriers (implicit or explicit)

```
#pragma omp parallel
```

```
{
```

```
    #pragma omp task  
    A();
```

```
    #pragma omp barrier
```

```
    #pragma omp single
```

```
{
```

```
        #pragma omp task  
        B();  
    }
```

```
    #pragma omp task  
    C();
```

```
}
```

There are multiple task A(), they are all guaranteed to be completed at the barrier

There is only one B() task, it is guaranteed to be completed at the single implicit barrier

All tasks are guaranteed to be completed at the implicit barrier

OpenMP Tasking

```
int fib ( int n )  
{  
    int x,y;  
    if ( n < 2 ) return n;  
  
    #pragma omp task  
    x = fib(n-1);  
  
    #pragma omp task  
    y = fib(n-2);  
  
    #pragma omp taskwait  
  
    return x+y;  
}
```

RED FLAG

But this code is wrong! Why? We need to revisit data scoping.

The taskwait construct specifies a wait on the completion of child tasks of the current task

The current task is the implicit one executing the fib function. The child tasks are both tasks created in this scope.

Both tasks are guaranteed to be completed in the taskwait

OpenMP Tasking

- The data-sharing attribute rules specifies
 - For constructs **other than task generating constructs**, if no default clause is present, these variables reference the variables with the same names that exist in the enclosing context.
 - The issue with tasks is that they may be executed by a thread which is different than the thread that created that task, and even much later after the variables are out of scope, so accessing the enclosing context is dangerous
 - Therefore, in a task generating construct, if no default clause is present, a variable for which the data-sharing attribute is not determined by the rules above is **firstprivate**.
 - A good rule for safe programming is always declare a default(none) in the clause. This forces to think in each variable being used.

OpenMP Tasking

```
int fib ( int n )  
{  
    int x,y;  
    if ( n < 2 ) return n;
```

n,x,y are firstprivate within the child tasks

- it is ok for n, since it is not changed
- But the changes in x and y are needed at the end

```
    #pragma omp task shared(x)  
    x = fib(n-1);
```

Solved by explicitly forcing x and y to be shared




```
    #pragma omp task shared(y)  
    y = fib(n-2);
```




```
    #pragma omp taskwait
```

```
    return x+y;  
}
```

OpenMP Tasking

- Need to be careful
 - If a variable has already a shared attribute, it inherits the attribute inside the task

```
int x;  
#pragma omp parallel  
#pragma omp single  
{  
    x = 2;  x is shared  
  
    #pragma omp task  x is shared  
    x = 3;  
  
    printf("%d\n", x);  Outputs 3  
  
}
```

```
#pragma omp parallel  
#pragma omp single  
{  
    int x = 2;  x is private  
  
    #pragma omp task  x is firstprivate  
    x = 3;  
  
    printf("%d\n", x);  Outputs 2  
  
}
```

Hands on

- Implement a matrix multiplication program using tasks instead of parallel loops

Taskgroup

- It is possible to specify a region for all created tasks (including descendent tasks) which allows:
 - Synchronizing the completion of all tasks in the region

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp taskgroup
    {
        #pragma omp task
        {
            printf("Task 1\n");

            #pragma omp task
            {
                sleep(1);
                printf("Task 2\n");
            }
        }
    }
    #pragma omp task
    printf("Task 3\n");
}
```

Taskgroup region is created

Difference to taskwait. Task 2 is not a child task of the implicit task in the single where taskwait is. Therefore, Task 3 can execute before Task 2 completes.

All child tasks (1) and descendants (2) complete here

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task
    {
        printf("Task 1\n");

        #pragma omp task
        {
            sleep(1);
            printf("Task 2\n");
        }
    }
    #pragma omp taskwait

    #pragma omp task
    printf("Task 3\n");
}
```

Taskgroup

- It is possible to specify a region for all created tasks (including descendent tasks) which allows:
 - Reducing values among tasks

```
int res = 0;  
node_t* node = NULL;  
...
```

```
#pragma omp parallel  
#pragma omp single  
{
```

```
    #pragma omp taskgroup task_reduction(+: res)
```

Register that there is a reduction in the taskgroup

```
    {
```

This task is participating in the reduction

```
        while (node) {
```

```
            #pragma omp task in_reduction(+: res) firstprivate(node)
```

```
            {
```

```
                res += node->value;
```

Node is shared and will be changed in the next instruction, so needs to make firstprivate

```
            }
```

```
            node = node->next;
```

```
        }
```

```
    }
```

Res is shared, but it is participating in the reduction so the compiler and runtime handle the creation of private (per thread) copy

```
}
```

Value is reduced here

```
printf("%d\n", res);
```

Taskloops

- The taskloop construct specifies that the iterations of one or more associated loops will be executed in parallel using explicit tasks
- Similar to parallel for, but with tasks
 - The taskloop is not a worksharing construct, as the work is not divided by the threads in the team
 - Instead, tasks are generated, to execute iterations of the loop, and dynamically executed by the threads
 - Allows the work to be done concurrently with other tasks

```
void one_task(void);  
void another_task(void);  
void loop_body(int i, int j);
```

```
void parallel_work(void) {  
    int i, j;  
    #pragma omp taskgroup  
    {  
        #pragma omp task  
        one_task(); // can execute concurrently  
  
        #pragma omp taskloop private(j)  
        for (i = 0; i < 10000; i++) { // can execute concurrently  
            for (j = 0; j < i; j++) {  
                loop_body(i, j);  
            }  
        }  
  
        #pragma omp task  
        another_task(); // can execute concurrently?  
    }  
}
```

Taskloops

The tasks in the loop execute concurrently with one_task()

The loop is parallelized with tasks, each task taking a certain amount of iterations. Similar to parallel loops, the for loop needs to be in a canonical form. Also, the programmer cannot rely on any specific order of execution of iterations

The taskloop defines a taskgroup, all tasks in the loop belong to the same group. This means that there is an implicit barrier at the end of the loop (only to the tasks in the taskgroup).

Therefore, another_task() executes concurrently with one_task(), but not with the tasks in the loop (these have to be already completed).

```
void one_task(void);  
void another_task(void);  
void loop_body(int i, int j);
```

```
void parallel_work(void) {  
    int i, j;  
    #pragma omp taskgroup  
    {  
        #pragma omp task  
        one_task(); // can execute concurrently
```

An explicit taskgroup can be used to guarantee that all tasks in the function complete before the function returns

Not actually needed, task generating constructs, variables are firstprivate by default

```
        #pragma omp taskloop private(j)  
        for (i = 0; i < 10000; i++) { // can execute concurrently  
            for (j = 0; j < i; j++) {  
                loop_body(i, j);  
            }  
        }  
    }
```

```
    #pragma omp task  
    another_task(); // can execute concurrently?  
}
```

Taskloops

```
void one_task(void);  
void another_task(void);  
void loop_body(int i, int j);
```

```
void parallel_work(void) {  
    int i, j;  
    #pragma omp taskgroup  
    {  
        #pragma omp task  
        one_task(); // can execute concurrently
```

The nogroup clause in the taskloop defines that a taskgroup is not created.

```
        #pragma omp taskloop private(j) nogroup  
        for (i = 0; i < 10000; i++) { // can execute concurrently  
            for (j = 0; j < i; j++) {  
                loop_body(i, j);  
            }  
        }  
    }
```

Therefore, another_task() now executes also concurrently with all tasks in the loop

```
        #pragma omp task  
        another_task(); // can execute concurrently  
    }  
}
```

Taskloops

Grainsize clause allows to define the number of loop iterations assigned to each task (it will be between 500 and 1000).

Num_tasks clause defines the number of tasks that will be generated (but it will not exceed the number of iterations).

If these are not specified, the number of tasks, and the mapping of iterations is implementation defined.

You can also collapse several loops, similar to parallel for.

```
void one_task(void);  
void another_task(void);  
void loop_body(int i, int j);
```

```
void parallel_work(void) {  
    int i, j;  
    #pragma omp taskgroup  
    {  
        #pragma omp task  
        one_task();
```

```
        #pragma omp taskloop private(j) grainsize(500) num_tasks(100)  
        for (i = 0; i < 10000; i++) {  
            for (j = 0; j < i; j++) {  
                loop_body(i, j);  
            }  
        }  
    }
```

```
        #pragma omp task  
        another_task();  
    }  
}
```

Reductions again

- Reductions have been extended to allow mixing reductions across constructs

```
int sum = 0;
#pragma omp parallel num_threads(2) reduction(task, +=:sum)
{
    #pragma omp single
    #pragma omp taskgroup
    {
        #pragma omp task in_reduction(+: sum)
        sum++;
        #pragma omp task in_reduction(+: sum)
        sum++;
    }
    #pragma omp taskgroup
    {
        #pragma omp task in_reduction(+: sum)
        sum++;
        #pragma omp task in_reduction(+: sum)
        sum++;
    }
}
#pragma omp taskloop in_reduction(+:sum) num_tasks(10)
for(int i = 0; i < 1000; i++)
    sum++;
printf("%d\n", sum);
```

A task modifier in the reduction allows to reduce across task in the region

Hands on

What is the value of sum printed at the end?

Hands on

- Implement the matrix multiplication program using taskloop

Data Dependencies

- A new data synchronization mechanism was introduced in OpenMP 4.0
 - It is possible to specify dependencies between tasks, related to the input/output variables
 - Dependencies are only possible between tasks that have the same parent task (siblings)
 - A task is only ready for execution when all its input dependencies are solved (dependency tasks completed)
 - Dependency can be:
 - in – the task will depend on the completion of all previously generated tasks that specify one of the variables in its out (or inout) clause
 - out and inout are the same – the task will depend on the completion of all previously generated tasks that specify one of the variables in its in, out (or inout) clause
 - The rules of dependency are strict (and strange), but these are to make execution deterministic
 - Dependencies can also be in array slices
 - The dependency exists if slices overlap

Data Dependencies

```
x= 0;  
#pragma omp parallel  
#pragma omp single nowait  
{
```

```
    #pragma omp task depend (out:x) //task1  
    x=1;
```

Task 1 modifies x

```
    #pragma omp task depend(in:x) depend(out:y) // task2  
    y=x+1;
```

Task 2 reads x, and modifies y

```
    #pragma omp task depend (inout:x) //task3  
    x++;
```

Task 3 reads and modifies x

```
    #pragma omp task depend (in:x,y) //task4  
    printf("task4 (x+y): %d\n" , x+y);
```

Task 4 reads x and y

```
}
```

What is the value printed in task 4?

Note that there are dependencies on x and y, so these are shared (contrarily to default)

Data Dependencies

```
x= 0;
#pragma omp parallel
#pragma omp single nowait
{
    #pragma omp task depend (out:x) //task1
    x=1;

    #pragma omp task depend(in:x) depend(out:y) // task2
    y=x+1;

    #pragma omp task depend (inout:x) //task3
    x++;

    #pragma omp task depend (in:x,y) //task4
    printf("task4 (x+y): %d\n" , x+y);
}
```

The *inout* defines that task 3 waits for the completion of task 1 (outputs x), but also of task 2 (which only reads x)!

Why OpenMP forces a dependency between task 2 and task 3, if task 3 does not change task 2?

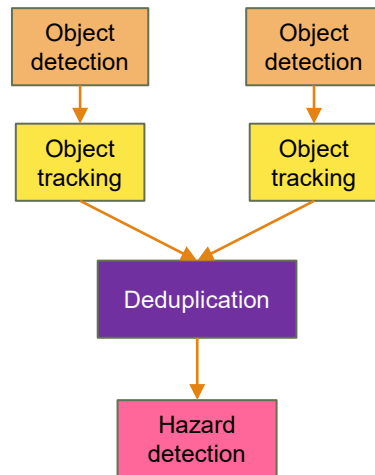
Reason is to guarantee determinism, of execution. If *inout* did not depend on a previous *in*, then result could be:

- Task 3 could execute after task 2, final result would be 4
- Task 3 could execute before task 2, final result would be 5

Note that this example serializes execution, it is not a good example to use data dependencies

Data Dependencies

- Data dependencies are appropriate when we have complex data dependencies
 - Dataflow computation models



```
x= 0;
#pragma omp parallel
#pragma omp single nowait
{
    #pragma omp task depend(out: imageA)
    object_detect(imageA);

    #pragma omp task depend(out:imageB)
    object_detect(imageB);

    #pragma omp task depend(inout: imageA)
    object_track(imageA);

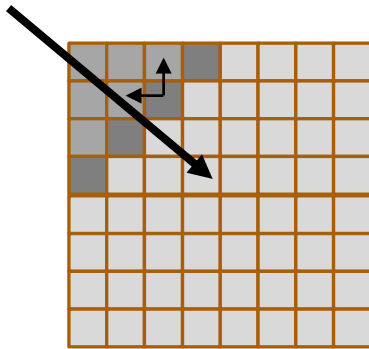
    #pragma omp task depend(inout:imageB)
    object_track(imageB);

    #pragma omp task depend(in:imageA,imageB) depend(out:objects)
    deduplication(imageA, imageB, objects);

    #pragma omp task depend(in:objects)
    hazard_detection(objects);
}
```

Data Dependencies

- Data dependencies are appropriate when we have complex data dependencies
 - Dynamic programming (e.g. wavefront computation)



```
x= 0;
#pragma omp parallel
#pragma omp single
{
    for(i = 1; i < size; i++)
        for(j = 1; j < size; i++)
            #pragma omp task depend(in: A[i][j-1], A[i-1,j]) depend(out: A[i][j])
            process_block(i,j);
}
```

Hands on

- Implement the Fibonacci example without using the taskwait
 - Which one is faster?
 - Can you draw the task dependency graph, assuming 4 levels of recursive tasks?

Hands on

- Parallelize the following code, with tasks and dependencies, dividing the matrix in blocks of BS size
 - Inside each block the execution is sequential
 - Assume N is multiple of BS

```
for(i = 1; i < N - 1; i++)  
    for(j = 1; j < N - 1; j++)  
        M[i][j] = (M[i][j-1] + M[i-1][j] + M[i][j+1] + M[i+1][j])/4.0;
```

- Can you parallelize, without dependencies?
- Can you draw the task dependency graph, assuming N = 12 and BS = 3

Scheduling of tasks

- The OpenMP specification defines two types of tasks, **tied** and **untied**
 - In the default behaviour, tasks are **tied** to the thread that first executes them (may not be the creator of the task)
 - A task always executes in the same thread
 - Tasks run to completion (execute with non-preemptable semantics) in the thread, except they can be suspended at task scheduling points:
 - Task creation, task finish, taskwait, barrier, taskyield directive
 - In tied tasks, if the task is suspended, the thread can only switch to a direct descendant of all tasks tied to the thread
 - A new tied task can only be scheduled in a thread if it is a descendant of all the other tasks suspended in that thread

Scheduling of tasks

- Tasks created with the **untied** clause are never tied
 - Resume at task scheduling points possibly by a different thread
 - More flexibility and freedom to the implementation, e.g., load balancing
- The decision to make the tied model the default was related to the underlying, and legacy, thread model of OpenMP
 - Allowing tasks to migrate across threads, introduces several problems
 - Thread id-based code would break
 - Mutual exclusion would deadlock, as locks are owned by threads
 - Private data to a thread need to migrate which is not easy nor efficient to implement
 - Therefore, most of the OpenMP implementations do not use this model

Scheduling of tasks

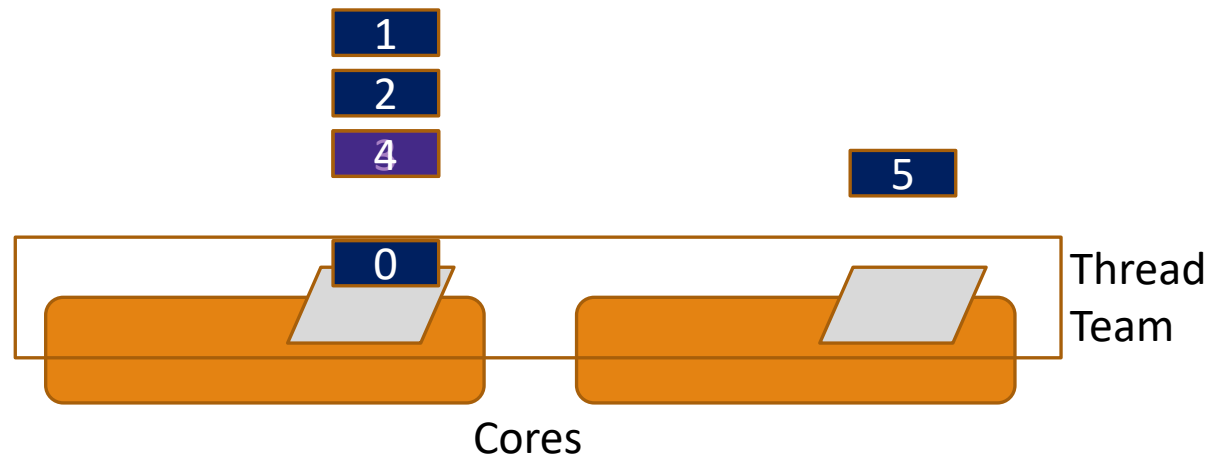
- Although the OpenMP specification leaves open the implementation of how tasks are mapped to the threads, typical implementations use breadth-first (BFS) and work-first (WFS) schedulers
 - BFS creates all children tasks before executing them
 - WFS executes new tasks immediately after they are created
- WFS theoretically is better as it improves cache locality
 - But it completely serialises execution with the tied model of OpenMP (!!)
- Some implementations use a single queue for all tasks (in the same thread team), but better and most recent (e.g. intel or llvm) use work-stealing approaches
 - In this approach each thread has its own queue, tasks created are inserted in the queue of the thread that creates it
 - Threads first try to execute tasks from its queue (LIFO, to increase cache locality)
 - If a thread's queue is empty, a thread will remove (steal) from another's thread queue
 - But FIFO, so in a different end of the queue from the queue owner, this reduces contention, and reduces probability of impact on cache (older "cold" tasks are being taken)

Scheduling of tasks

■ Fairness vs. Performance vs. Memory Awareness

- It is difficult to get fair task scheduling and high performance at the same time.
 - For instance, to be fair, work items should be consumed in the order in which they arrived (FIFO). But for performance (cache), LIFO is better
 - Work-stealing achieves a compromise
 - Memory hierarchy is one of the most important issues for performance tuning—exploiting memory locality, especially for higher numbers of cores, is indispensable.
 - In non-uniform memory access (NUMA) architectures, accessing memory locations in "distant" nodes is expensive and can hurt performance significantly.
 - Therefore, understanding the underlying memory hierarchy is important for scheduling purposes.
 - For example, a way of keeping memory accesses more localized in a multiprocessor environment would be to maintain threads working on a particular task in the same processor.
- To reduce cache impact, recent OpenMP specification allows to define task affinities, not only thread affinities
 - To improve cache locality among tasks

Scheduling of tasks



Going further

■ Cutoff mechanism

- It is possible dynamically during execution to cutoff parallelism testing for conditions, with the if clause
- Allows for debugging (test if parallelism enabled)
- But mainly to dynamically decide to parallelize or go sequential, avoiding creation of “small” tasks

```
int fib ( int n )  
{  
    int x,y;  
    if ( n < 2 ) return n;  
    #pragma omp task if(n>10)  
    x = fib(n-1);  
    #pragma omp task if(n>10)  
    y = fib(n-2);  
    #pragma omp taskwait  
  
    return x+y;  
}
```

If if clause evaluates to false, the new task is executed immediately (task dependences are respected), the current task will resume once the new task is completed.

Going further

- The cancel construct tries to cancel the parallel execution
 - Needs to state which construct is being cancelled (referring always to the innermost one)
 - Cancellation is not immediate
 - The task that made the cancel continues and completes
 - Tasks that are executing either complete or execute until they reach the cancelling point (the cancel directive or a barrier)
 - Tasks that have not yet started may not start (or may start if did not detect cancel in time)

```
int search(int x)
{
    int flag = 0;
    #pragma omp parallel
    #pragma omp for
    for(int i = 0; i < 100000; i++)
    {
        if(array[i] == x) flag = 1;
        #pragma omp cancel for if (flag == 1)
    }
    return flag;
}
```

Going further

- What is the correct parallel chunk size?
 - Granularity is a qualitative measure of the ratio of computation to communication
 - Computation stages are typically separated from periods of communication by synchronization events
 - Fine-grain Parallelism (small chunks)
 - Low computation to communication ratio
 - Small amounts of computational work between communication stages
 - Less opportunity for performance enhancement
 - High communication overhead
 - Coarse-grain Parallelism (large chunks)
 - High computation to communication ratio
 - Large amounts of computational work between communication events
 - More opportunity for performance increase
 - Harder to load balance efficiently
 - More difficult to scale
 - If too coarse, where is the parallelism?
- No silver bullet, many times need to profile

Going further

- Declare reduction can be used to specify user-defined reductions for user data types

```
struct point {  
    int x;  
    int y;  
};
```

```
void minproc ( struct point *out, struct point *in ){  
    if ( in->x < out->x ) out->x = in->x;  
    if ( in->y < out->y ) out->y = in->y;  
}
```

```
void maxproc ( struct point *out, struct point *in ){  
    if ( in->x > out->x ) out->x = in->x;  
    if ( in->y > out->y ) out->y = in->y;  
}
```

```
#pragma omp declare reduction(min : struct point : minproc(&omp_out, &omp_in))  
    initializer( omp_priv = { INT_MAX, INT_MAX } )
```

```
#pragma omp declare reduction(max : struct point : maxproc(&omp_out, &omp_in))  
    initializer( omp_priv = { 0, 0 } )
```

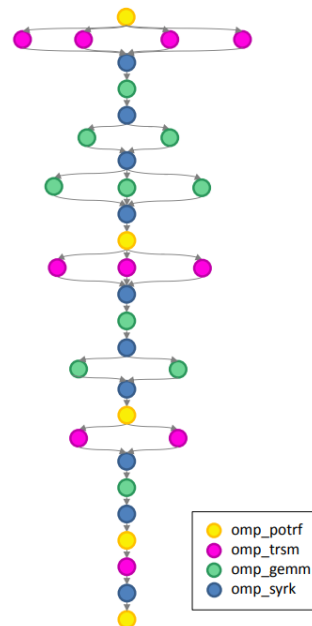
```
void find_enclosing_rectangle ( int n, struct point points[] )  
{  
    struct point minp = { INT_MAX, INT_MAX }, maxp = {0,0};  
    int i;
```

```
#pragma omp parallel for reduction(min:minp)  
reduction(max:maxp)  
    for ( i = 0; i < n; i++ ) {  
        minproc(&minp, &points[i]);  
        maxproc(&maxp, &points[i]);  
    }  
    printf("min = (%d, %d)\n", minp.x, minp.y);  
    printf("max = (%d, %d)\n", maxp.x, maxp.y);  
}
```

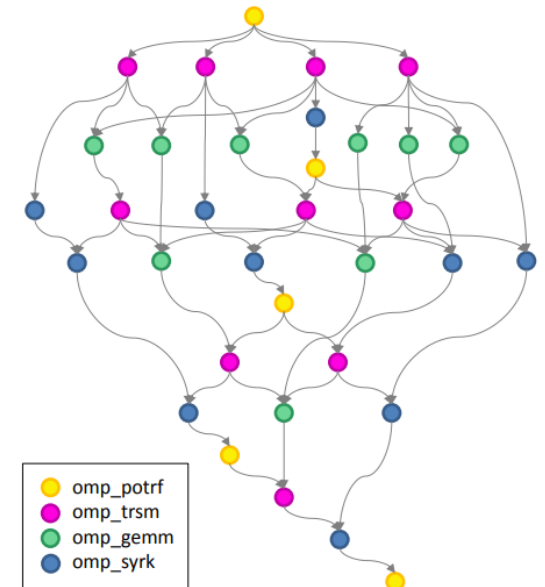

Going further

- Task dependencies enable the implementation of an asynchronous programming model
 - Similar to futures/promises, async/await in other programming paradigms
 - However, it imposes an underlying sequential ordering, not implementing these other paradigms
 - This helps building a static task dependency graph, which can be used with static analysis tools to check data dependency errors

```
void cholesky.c.task.taskwait(struct parallel_data *args) {
    double (*M)[NB][BS+BS] = (*args).M;
    if (omp_get_thread_num() == 0) {
        for (int k = 0; k < NB; k++) {
            omp_potrf((*)M)[k][k], BS, BS);
            for (int i = k + 1; i < NB; i++) {
                struct omp_trsm_data targs;
                targs.M = M; targs.k = k; targs.i = i;
                GOMP_task((void *)(&M)[k][i], BS, BS),
                    &targs, (void *)(&M)[k][i], BS, BS), 0, 24, 8, 1,
                    GOMP_TASK_UNTIED, 0, 0);
            }
            GOMP_taskwait();
        }
        for (int i = k + 1; i < NB; i++) {
            for (int j = k + 1; j < i; j++) {
                struct omp_gemm_data targs;
                targs.M = M; targs.k = k; targs.i = i; targs.j = j;
                GOMP_task((void *)(&M)[k][i], BS, BS),
                    &targs, (void *)(&M)[k][i], BS, BS), 0, 32, 8, 1,
                    GOMP_TASK_UNTIED, 0, 0);
            }
            GOMP_taskwait();
            omp_syrk((*)M)[k][i], (*M)[i][i], BS, BS);
        }
    }
}
```



```
void cholesky.c.task.dependencies(struct parallel_data *args) {
    double (*M)[NB][BS+BS] = (*args).M;
    if (omp_get_thread_num() == 0) {
        for (int k = 0; k < NB; k++) {
            struct omp_potrf_data targs;
            targs.M = M; targs.k = k;
            void *deps[3L] = {[0] = (void *)IU, [1] = (void *)IU,
                             [2] = (*M)[k][k]};
            GOMP_task((void *)(&M)[k][k], BS, BS),
                &targs, (void *)(&M)[k][k], BS, BS), 0, 32, 8, 1,
                UNTIED_DEPEND, deps, 0);
            for (int i = k + 1; i < NB; i++) {
                struct omp_trsm_data targs;
                targs.M = M; targs.k = k; targs.i = i;
                void *deps[4L] = {[0] = (void *)IU, [1] = (void *)IU,
                                 [2] = (*M)[k][k], [3] = (*M)[k][i]};
                GOMP_task((void *)(&M)[k][i], BS, BS),
                    &targs, (void *)(&M)[k][i], BS, BS), 0, 32, 8, 1,
                    UNTIED_DEPEND, deps, 0);
            }
            for (int i = k + 1; i < NB; i++) {
                for (int j = k + 1; j < i; j++) {
                    struct omp_gemm_data targs;
                    targs.M = M; targs.k = k; targs.i = i; targs.j = j;
                    void *deps[5L] = {[0] = (void *)IU, [1] = (void *)IU,
                                     [2] = (*M)[k][k], [3] = (*M)[k][i],
                                     [4] = (*M)[i][i]};
                    GOMP_task((void *)(&M)[k][i], BS, BS),
                        &targs, (void *)(&M)[k][i], BS, BS), 0, 40, 8, 1,
                        UNTIED_DEPEND, deps, 0);
                }
                struct omp_syrk_data targs;
                targs.M = M; targs.k = k; targs.i = i;
                void *deps[4L] = {[0] = (void *)IU, [1] = (void *)IU,
                                 [2] = (*M)[k][k], [3] = (*M)[i][i]};
                GOMP_task((void *)(&M)[i][i], BS, BS),
                    &targs, (void *)(&M)[i][i], BS, BS), 0, 32, 8, 1,
                    UNTIED_DEPEND, deps, 0);
            }
        }
    }
}
```



Advanced reading

- OpenMP Application Programming Interface, Version 5.2 November 2021,
<https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf>
- OpenMP API 5.2 Examples,
<https://www.openmp.org/wp-content/uploads/openmp-examples-5-2.pdf>
- Introduction to OpenMP - Tim Mattson (Intel), series of lectures,
<https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG>
- Klemm, Michael and Cownie, Jim. High Performance Parallel Runtimes: Design and Implementation, Berlin, Boston: De Gruyter Oldenbourg, 2021.
<https://doi.org/10.1515/9783110632729>

Credits

- Version 1.0, Luis Miguel Pinho, with inputs from:
 - L.M. Pinho, Parallel Programming Techniques and Patterns, Scheduling, Computação Avançada, 2011
 - A “Hands-on” Introduction to OpenMP, Tim Mattson, https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf
 - OpenMP Tasking, Christian Terboven, Michael Klemm, <https://www.openmp.org/wp-content/uploads/sc15-openmp-CT-MK-tasking.pdf>