

Timing Analysis of Parallel Real-Time Systems

Luis Miguel Pinho & Tiago Carvalho

CERCIRAS Training School 2023: Advanced Topics in Resource-Aware Computing
Riga, Latvia, September 2023

Overview of OpenMP Tasking Model - Basics

OpenMP Basics

- OpenMP (www.openmp.org) is a specification from a consortium (OpenMP Architecture Review Board)
 - Membership includes the main companies and academic organizations in the high-performance computing domain
- The specification provides
 - an annotation-based (compiler directives) approach for specifying parallelism
 - the required libraries, and
 - runtime environment for the execution of parallel computation.
- The focus is portability across shared memory systems, programmed in C/C++ or Fortran

OpenMP Basics

- It is one of the most common parallel programming models in the high-performance computing domain, being increasingly used in embedded computing systems
 - We will use the OpenMP Tasking Model, which provides a fine-grained model of parallelism allowing to specify graphs of computation
 - The programmer defines implicit and explicit tasks and data dependencies between them
 - Tasks are executed by a team of threads at runtime, which are managed by the underlying runtime, hiding complexity from the programmer
- OpenMP is a large and complex specification
 - We will be addressing only the basics to allow for this course

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

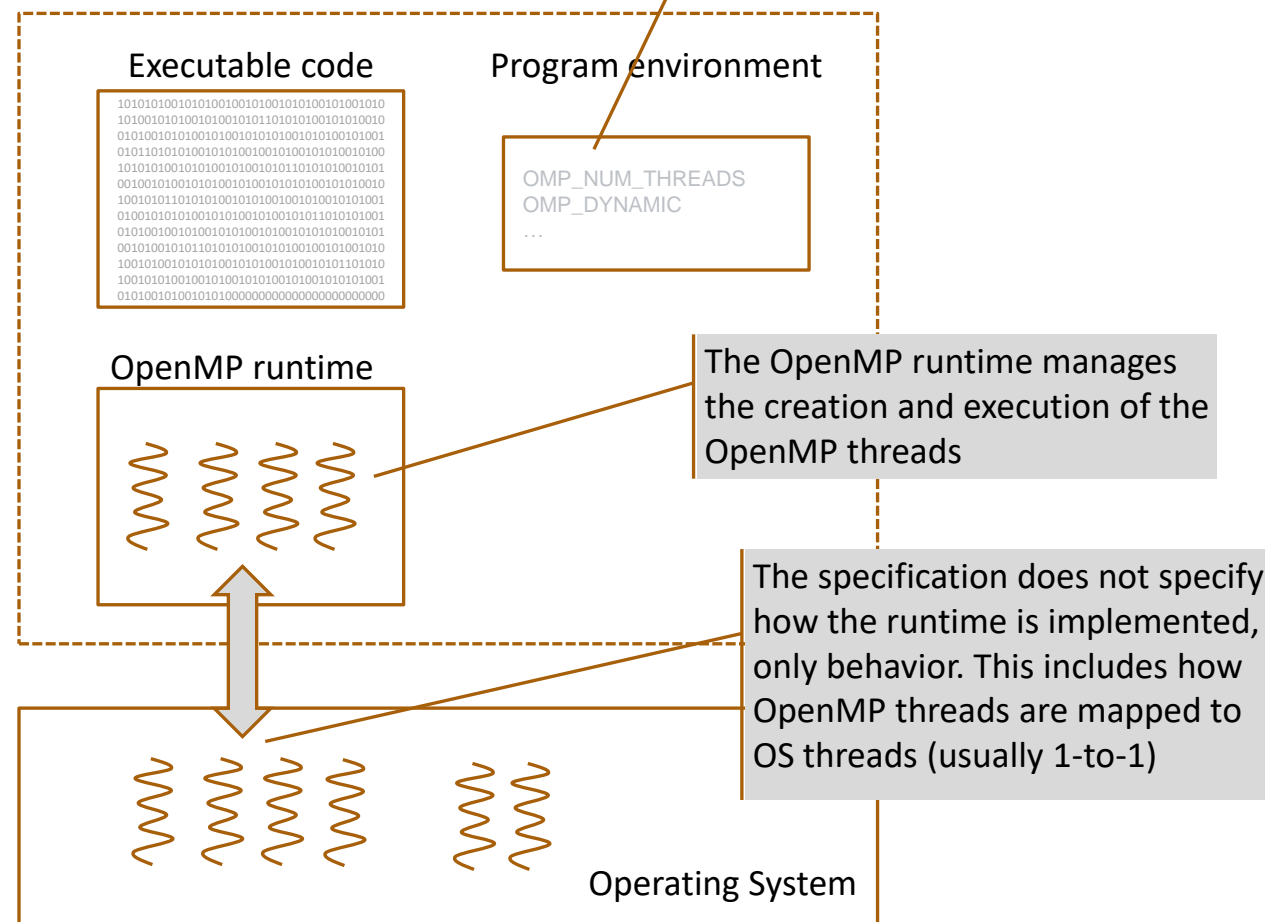
Goal is that the same algorithm can execute sequentially

OpenMP libraries

[illegible]

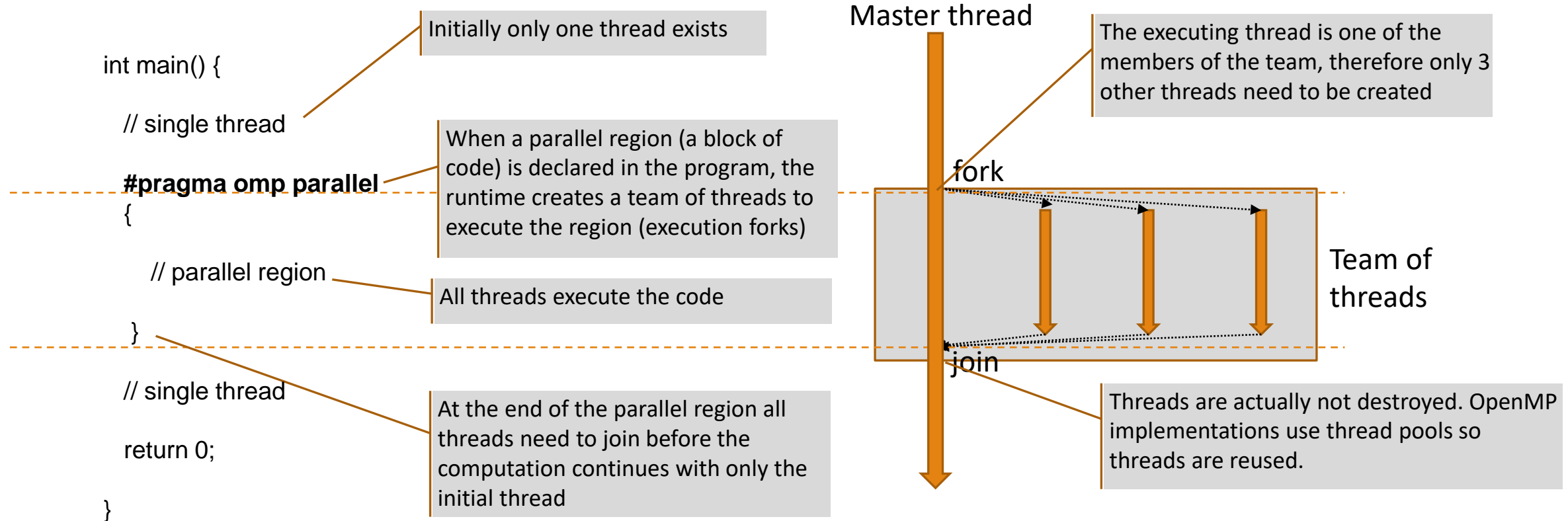
OpenMP library provides both functions to be used by the programmer, as well as runtime support hidden to programmers

OpenMP aware
compiler, e.g.
gcc -fopenmp



OpenMP Basics

- Basic OpenMP is based on the concept of parallel regions and fork-join computation



OpenMP Basics

#include <omp.h> required due to
omp_get_thread_num

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

```
int main() {
```

```
    #pragma omp parallel
    {
```

```
        int myid = omp_get_thread_num();
        printf("Hello world! I am thread %d\n", myid);
```

```
    }
```

```
    printf("Hello world! I am the main thread\n");
```

```
    return 0;
```

```
}
```

Team of 4 threads (default is
usually the number of logical
cores in the machine)

Order of execution inside the
parallel region is arbitrary

"Hello world! I am the main
thread" always last – it is
after the parallel region

Each thread is assigned a unique
id, between 0 (always the master)
and number of threads minus 1

```
user@ubuntu:~/omp$ ./example
Hello world! I am thread 3
Hello world! I am thread 0
Hello world! I am thread 1
Hello world! I am thread 2
Hello world! I am the main thread
user@ubuntu:~/omp$ ./example
Hello world! I am thread 0
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 1
Hello world! I am the main thread
user@ubuntu:~/omp$ ./example
Hello world! I am thread 1
Hello world! I am thread 0
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am the main thread
```

OpenMP Basics

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

int main() {

    #pragma omp parallel
    {
        int myid = omp_get_thread_num();
        if(myid == 0)
            printf("Hello world! I am the main thread (thread 0) \n");
        else
            printf("Hello world! I am thread %d\n", myid);
    }

    return 0;
}
```

Order of execution inside the parallel region is arbitrary

“Hello world! I am the main thread” is now printed as one of the team

```
user@ubuntu:~/omp$ ./example
Hello world! I am the main thread (thread 0)
Hello world! I am thread 3
Hello world! I am thread 1
Hello world! I am thread 2
user@ubuntu:~/omp$ ./ example
Hello world! I am the main thread (thread 0)
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am thread 1
user@ubuntu:~/omp$ ./ example
Hello world! I am the main thread (thread 0)
Hello world! I am thread 2
Hello world! I am thread 3
Hello world! I am thread 1
user@ubuntu:~/omp$ ./a example
Hello world! I am thread 3
Hello world! I am thread 2
Hello world! I am the main thread (thread 0)
Hello world! I am thread 1
```


OpenMP Basics

```
#include <stdio.h>
#ifdef _OPENMP
    #include <omp.h>
#endif

int main() {

    #pragma omp parallel
    {
        #ifdef _OPENMP
            int myid = omp_get_thread_num();
            int tcount = omp_get_num_threads ( );
        #else
            int myid = 0;
            int tcount = 1;
        #endif
        if(myid == 0)
            printf("Hello world! I am the main thread (thread %d), out of %d threads \n", myid, tcount);
        else
            printf("Hello world! I am thread %d out of %d threads\n", myid, tcount);
    }

    return 0;
}
```

There are limits to compiler ignoring parallelism, if using openmp library calls

```
user@ubuntu:~/omp$ gcc -Wall -fopenmp example.c - example
user@ubuntu:~/omp$ ./example
Hello world! I am thread 2 out of 4 threads
Hello world! I am thread 3 out of 4 threads
Hello world! I am the main thread (thread 0), out of 4 threads
Hello world! I am thread 1 out of 4 threads
user@ubuntu:~/omp$ gcc -Wall example.c - example
example.c: In function 'main':
example.c:8: warning: ignoring #pragma omp parallel [-Wunknown-pragmas]
   8 |     #pragma omp parallel
     |
user@ubuntu:~/omp$ ./example
Hello world! I am the main thread (thread 0), out of 1 threads
```

OpenMP Basics

- OpenMP provides several different alternatives to divide the work in the parallel region among the threads
 - Example of explicit handling of threads

```
sum = 0;
#pragma omp parallel
{
    int my_index = omp_get_thread_num();
    int n_threads = omp_get_num_threads ( );
    int inf = my_index*SIZE/n_threads;
    int sup = (my_index == n_threads - 1)? SIZE : (my_index+1)*SIZE/n_threads;
```

Using thread_id and number of threads to control ranges

```
    long long local_sum = 0;
    for(int i = inf; i < sup; i++)
        local_sum +=vector[i];
```

Each thread working in a separate region of the array and summing values in a thread local variable

```
    #pragma omp critical
    sum += local_sum;
}
```

At the end of calculation each thread updates the shared sum variable, in mutual exclusion.

OpenMP Basics

- OpenMP provides several different alternatives to divide the work in the parallel region among the threads
 - Example of implicit handling with parallel for and reductions

```
sum = 0;  
#pragma omp parallel  
#pragma omp for reduction(+:sum)  
for(i = 0; i < SIZE; i++)  
    sum += a[i];
```

```
sum = 0;  
#pragma omp parallel for reduction(+:sum)  
for(i = 0; i < SIZE; i++)  
    sum += a[i];
```

OpenMP Basics

- OpenMP defines default storage attributes for data
 - Variables are either Shared (all threads see the same variable) or Private (each thread has its own)
 - Using C, variables are SHARED if
 - They have a wider scope than the parallel region (they were declared before)
 - They are static
 - They are dynamically allocated
 - Using C, variables are PRIVATE if
 - Declared inside the parallel region
 - Declared in a function called from a parallel region (these are in stack, therefore they need to be private to the thread)
- The default storage attributes can be changed by specific clauses
 - The change applies to the specific variables listed and only in the scope of the construct that has the clause
 - Or the default behavior can be changed with the default clause

OpenMP Basics

- shared (only available in the parallel construct)
 - The variables in the list are shared between the threads
 - The variable needs to exist when the clause is reached, therefore it is shared by default
 - The clause is useful when the default is changed
- private
 - Each thread has a local private copy of the variable
 - The variable is uninitialized, and it is not copied back to the original
- firstprivate
 - Each thread has a local private copy of the variable, which is initialized with the value of the shared variable when the clause was reached
- lastprivate
 - Each thread has a local private copy of the variable, the value of the last sequentially-equivalent value is propagated back to the original

OpenMP Basics

```
void wrong() {  
    int sum = 0;  
  
    #pragma omp parallel for private(sum)  
    for (int j = 0; j < 1000; ++j)  
        sum += j;  
  
    printf("%d\n", sum);  
}
```

Each thread has its own sum variable, which is not initialized

Outside of the parallel region, the original sum variable is still 0

OpenMP Basics

```
void wrong() {  
    int sum = 0;  
  
    #pragma omp parallel for firstprivate(sum)  
    for (int j = 0; j < 1000; ++j)  
        sum += j;  
  
    printf("%d\n", sum);  
}
```

Each thread has its own sum variable, which is correctly initialized at 0

Outside of the parallel region, the original sum variable is still 0

OpenMP Basics

```
void wrong() {  
    int sum = 0;  
  
    #pragma omp parallel for firstprivate(sum) lastprivate(sum)  
    for (int j = 0; j < 1000; ++j)  
        sum += j;  
  
    printf("%d\n", sum);  
}
```

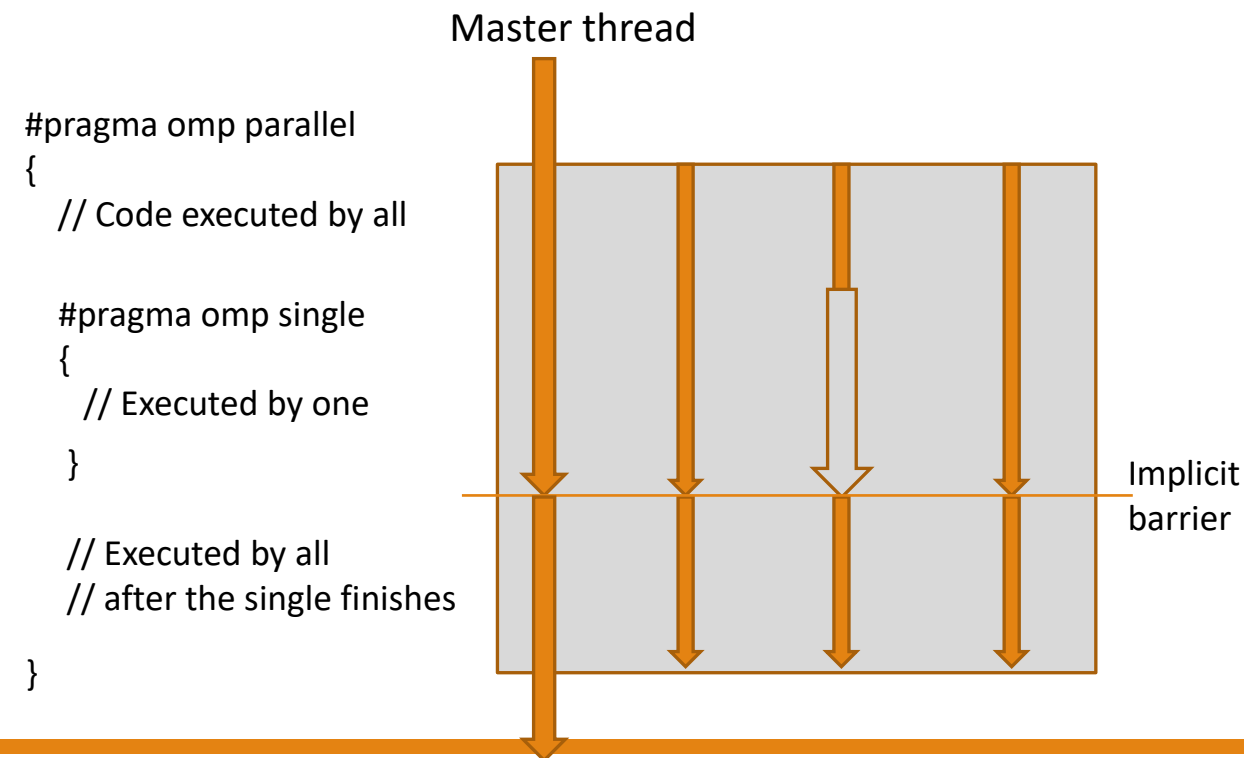
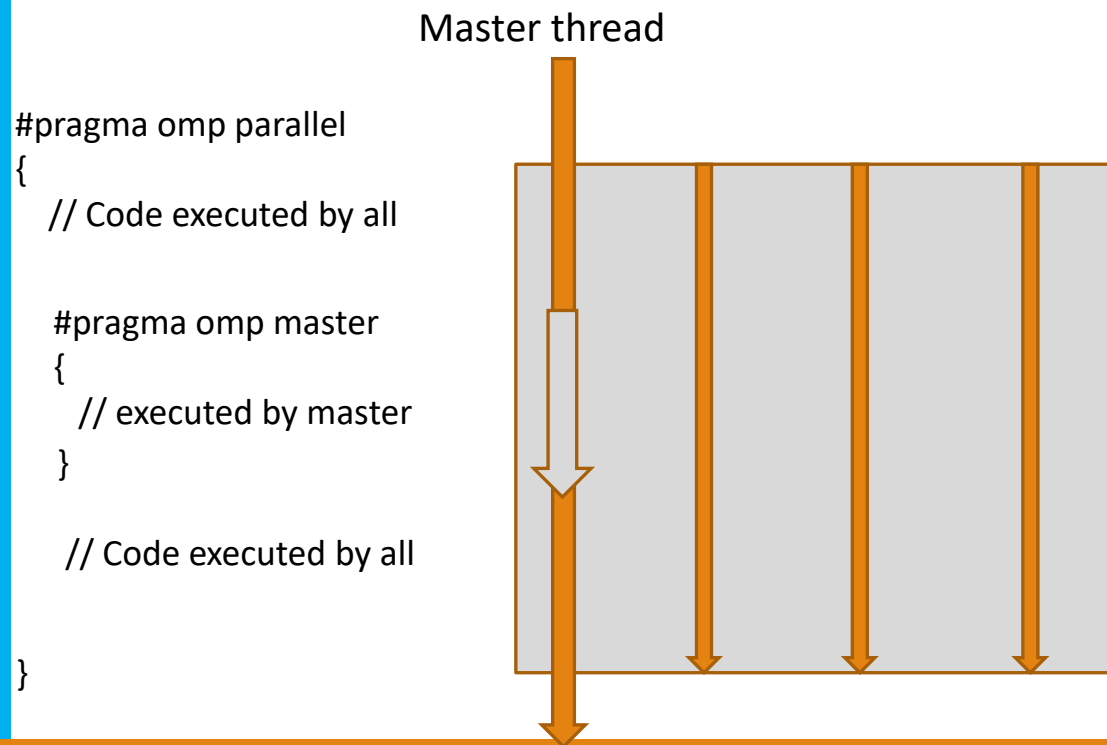
Each thread has its own sum variable, which is correctly initialized at 0

sum is defined as the value at the “last sequential” iteration (i.e., for j=999)

Still not quite correct

OpenMP Basics

- In the examples we have used explicitly the id of the master/primary thread with an if construct
 - But we can use directives to identify
 - Master/masked: only the master/primary thread will execute the block
 - Single: only one thread will execute the block, and all threads must wait for the block to be terminated (there is an implicit barrier)



Overview of OpenMP Tasking Model - Tasks and Dependencies

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
- 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 }
```

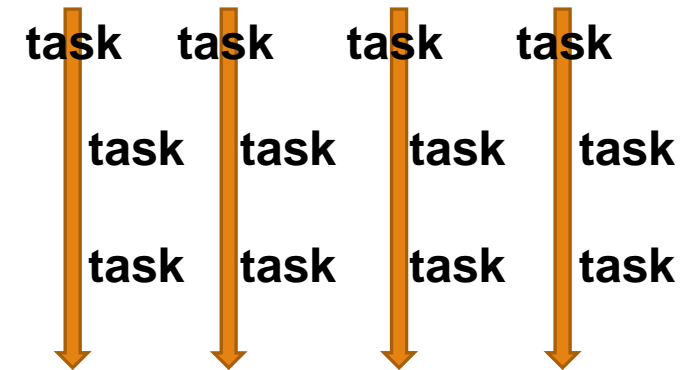
The programmer specifies that the code can execute in parallel to the rest of the code

```
    // 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?

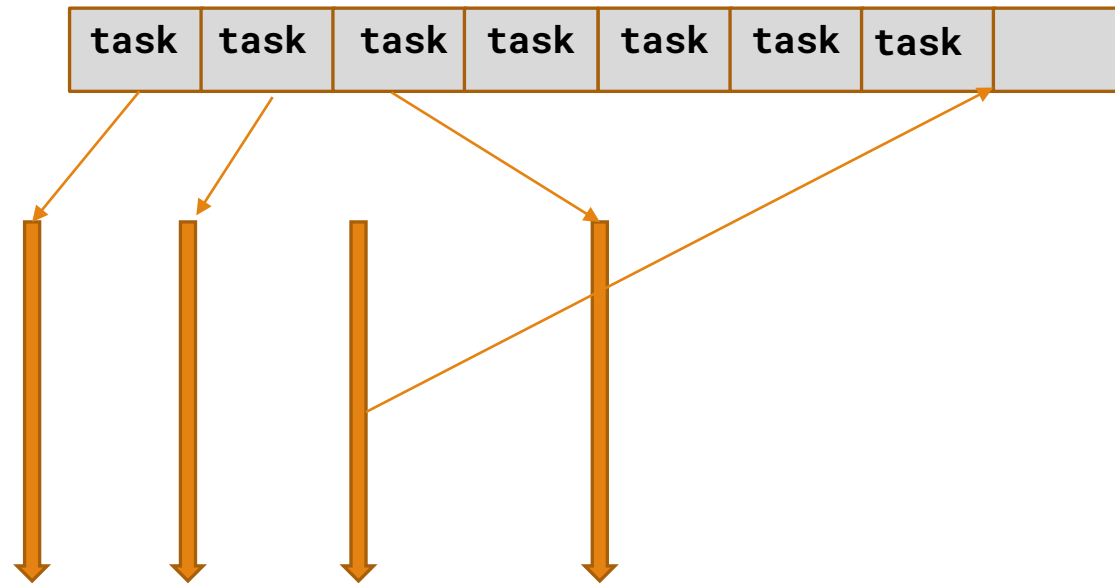


OpenMP Tasking

- 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 }

        // other tasks created
    }
}
```



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;
```

```
}
```

```
int main(){  
    #pragma omp parallel  
    #pragma omp single  
    int res = fib (...);
```

```
// ...
```

RED FLAG

But this code is wrong! Why? We need to address 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

Necessary that this return is guaranteed to already have the values of x and y

OpenMP Tasking changes Data Scoping

- 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;

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

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

    #pragma omp taskwait

    return x+y;
}
```

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

Solved by explicitly forcing x and y to be shared




Note that sequential fibnoacci is much faster in the general case, and requiring less memory.




This solution repeats the calculation, and saves all state in the stack

```
int main(){
    #pragma omp parallel
    #pragma omp single
    int res = fib (...);
    // ...
}
```

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;  
  
#pragma omp taskwait  
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;  
  
#pragma omp taskwait  
printf("%d\n", x);  Outputs 2  
}
```

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

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

```
    Int x, y;
```

```
    #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);
```

```
}
```

Task 1 modifies x

Task 2 reads x, and modifies y

Task 3 reads and modifies x

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?

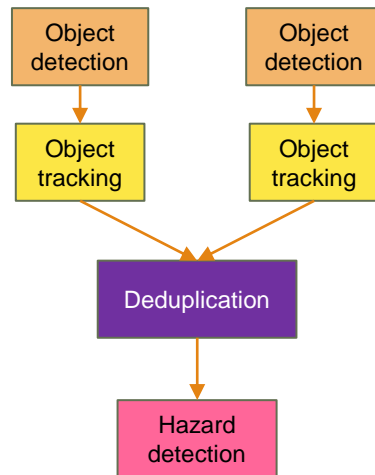
Reason is to guarantee determinism, of execution. If *inout* did not depend on a previous *in*, then result could be:

- 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);

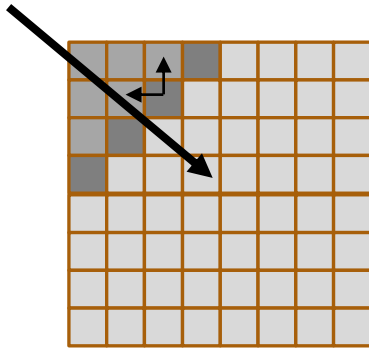
    #pragma omp task depend(inout:imageB)
    object_track(imageB);

    #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)



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

Example

- Parallelize the following code, with tasks and dependencies, dividing the matrix of $N \times N$ in blocks of $BS \times BS$ 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;
```

- Draw a graph with the data dependencies, assuming $BS = 4$

Example

```
void process_block(int i, int j)
{
    int i_inf = (i == 0)?1:i; // Do not process line 0
    int j_inf = (j == 0)?1:j; // Do not process column 0
    int i_sup = (i + BS == N)?(i + BS - 1):(i + BS); // Do not process last line
    int j_sup = (j + BS == N)?(j + BS - 1):(j + BS); // Do not process last column

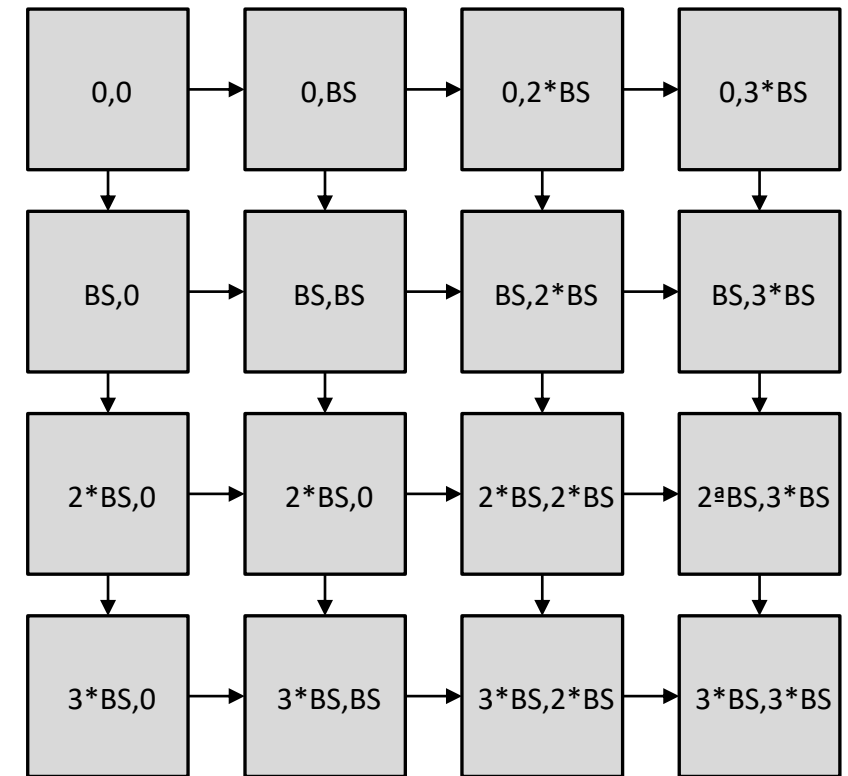
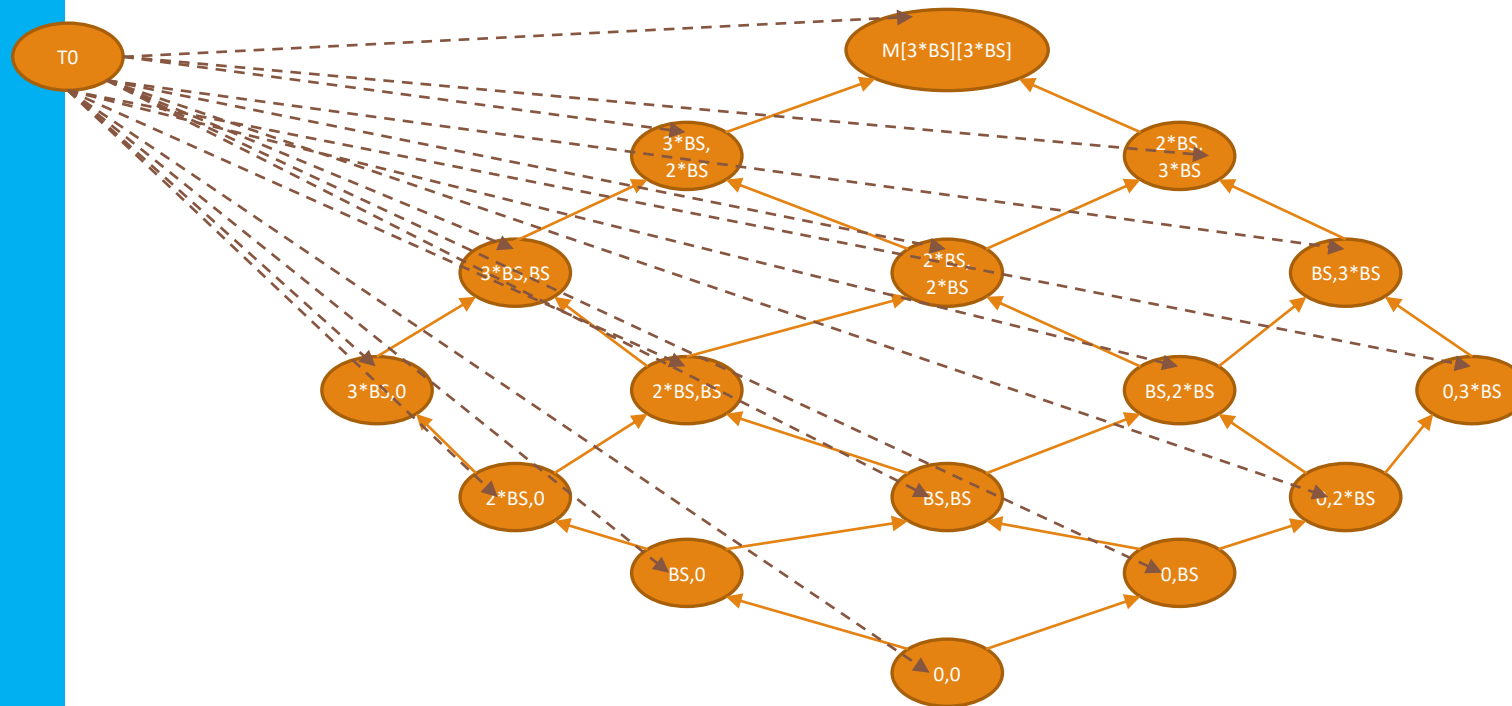
    for(int ii = i_inf ; ii < i_sup ; ii++)
        for(int jj = j_inf; jj < j_sup; jj++)
            M[ii][jj] = (M[ii][jj-1] + M[ii-1][jj] + M[ii][jj+1] + M[ii+1][jj])/4;
}

// ...
for(int i = 0; i < N ; i+=BS)
    for(int j = 0; j < N ; j+=BS)
        #pragma omp task depend(in: M[i-BS][j], M[i][j-BS]) depend(out: M[i][j])
        {
            process_block(i,j);
        }
```

Example

■ Task dependency graph

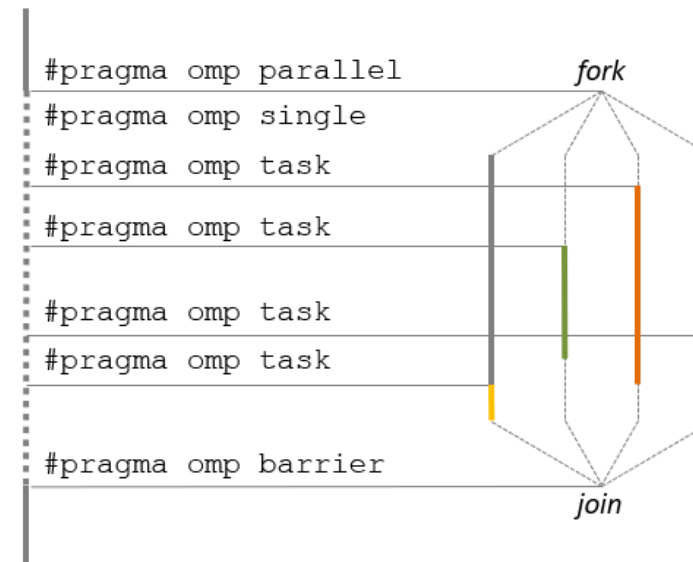
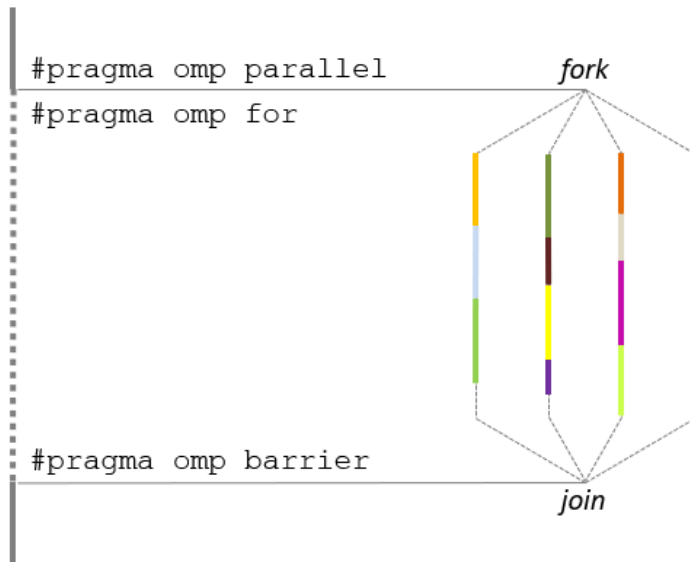
- T0 is the “main” task, all tasks are dependent of T0 (control flow dependency)
- $M[3*BS][3*BS]$ is the last task to execute



Overview of OpenMP Tasking Model - Support to Real-Time

Support to Real-Time Parallel Models

- OpenMP is able to support the Fork/Join model
 - Both structured and unstructured



- OpenMP is also able to support the DAG model
 - Using tasks and data dependencies

Support to Real-Time Parallel Models

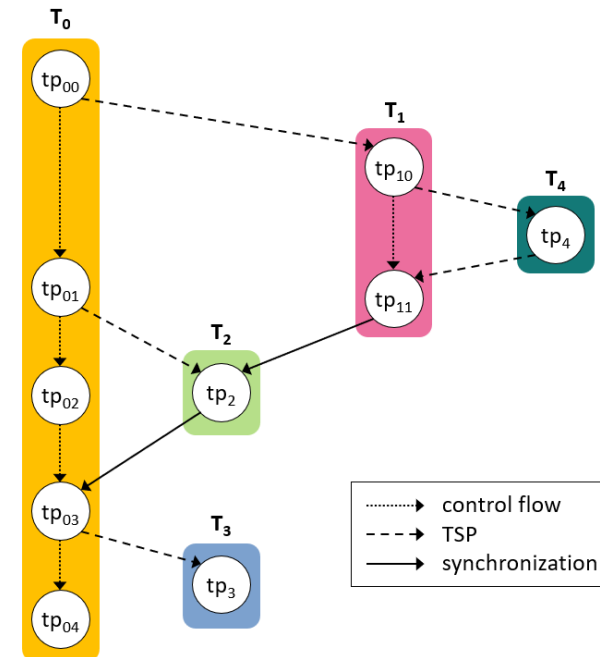
- The OpenMP specification defines two types of tasks, **tied** and **untied**
 - In the default behaviour, tasks are **tied** to the thread that first executes them (may not be the creator of the task), executes always in that thread
 - Tasks created with the **untied** clause are never tied and can migrate between threads
- 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
 - TSP divide the task into different “task parts”
- 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
- Untied tasks have no restrictions
 - Resume at task scheduling points possibly by a different thread
 - More flexibility and freedom to the implementation, e.g., load balancing

Support to Real-Time Parallel Models

```

#pragma omp parallel {
#pragma omp single
{
... // tp00
#pragma omp task depend(out: x)
{
... // tp10
#pragma omp task if(false)
{ ... } // tp4
... // tp11
}
... // tp01
#pragma omp task depend(in: x)
{ ... } // tp2
... // tp02
#pragma omp taskwait
... // tp03
#pragma omp task
{ ... } // tp3
... // tp04
}
}
    
```

// T0
// T1
// T4
// T2
// T3

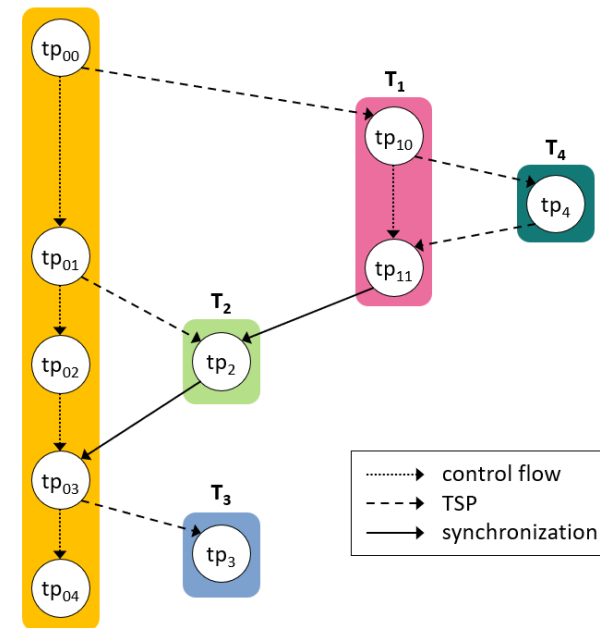


Support to Real-Time Parallel Models

```
#pragma omp parallel {  
  #pragma omp single  
  {  
    ... // tp00  
    #pragma omp task depend(out: x) // T1  
    {  
      ... // tp10  
      #pragma omp task if(false) // T4  
      { ... } // tp4  
      ... // tp11  
    }  
    ... // tp01  
    #pragma omp task depend(in: x) // T2  
    { ... } // tp2  
    ... // tp02  
    #pragma omp taskwait  
    ... // tp03  
    #pragma omp task // T3  
    { ... } // tp3  
    ... // tp04  
  }  
}
```

Task scheduling point: e.g., T1 is created by T0, tp10 can only be executed after tp00

Task scheduling constraints: T4 is immediately executed, tp11 needs to wait for tp4



Synchronization: barrier forces tp03 to wait

Data dependencies: tp2 waits for tp11

Control flow dependency: e.g., tp01 can only be executed after tp00

Support to Real-Time Parallel Models

- The DAG can be created just with data dependencies

```
void cholesky(float *M, int ntiles) {
    for (k = 0; k < ntiles; k++) {
        potrf (M[k][k]);
        for (i = k + 1; i < ntiles; i++)
            trsm (M[k][k], M[k][i]);
        for (i = k + 1; i < ntiles; i++) {
            for (j = k + 1; j < i; j++)
                gemm (M[k][i], M[k][j], M[j][i]);
            syrk (M[k][i], M[i][i]);
        }
    }
}
```


Support to Real-Time Parallel Models

- The DAG can be created just with data dependencies

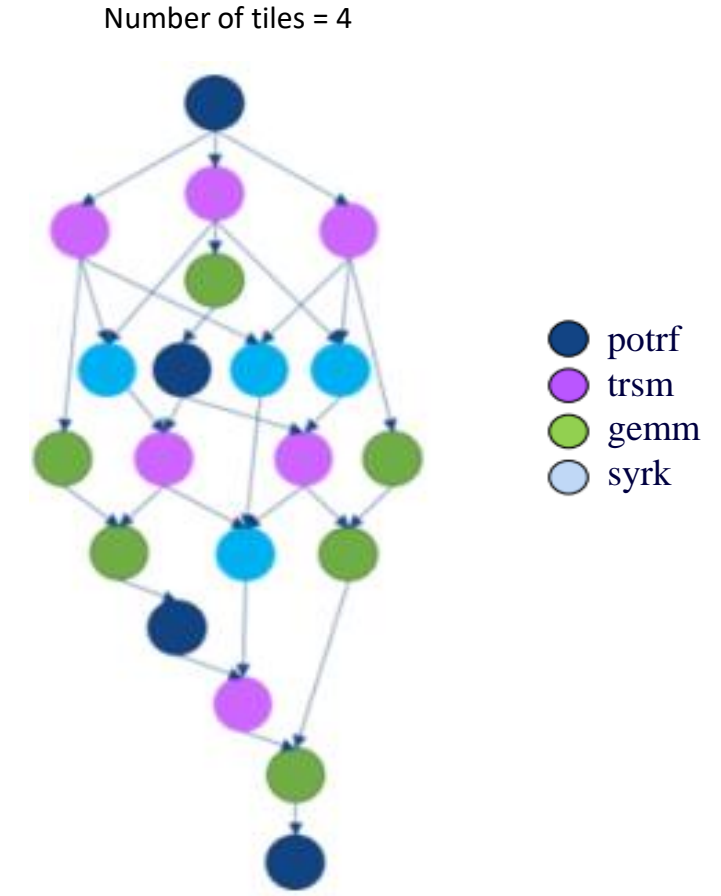
```
void cholesky(float *M, int ntiles) {  
    for (k = 0; k < ntiles; k++) {  
  
        potrf (M[k][k]);  
        for (i = k + 1; i < ntiles; i++)  
  
            trsm (M[k][k], M[k][i]);  
        for (i = k + 1; i < ntiles; i++) {  
            for (j = k + 1; j < i; j++)  
  
                gemm (M[k][i], M[k][j], M[j][i]);  
  
            syrks (M[k][i], M[i][i]);  
        }  
    }  
}
```

Support to Real-Time Parallel Models

- The DAG can be created just with data dependencies

```
void cholesky(float *M, int ntiles) {  
    for (k = 0; k < ntiles; k++) {  
        #pragma omp task depend(out:M[k][k])  
        potrf (M[k][k]);  
        for (i = k + 1; i < ntiles; i++)  
            #pragma omp task depend(in:M[k][k]) depend(out:M[k][i])  
            trsm (M[k][k], M[k][i]);  
        for (i = k + 1; i < ntiles; i++) {  
            for (j = k + 1; j < i; j++)  
                #pragma omp task depend(in:M[k][i],M[k][j])  
                depend(out:M[j][i])  
                gemm (M[k][i], M[k][j], M[j][i]);  
            #pragma omp task depend(in:M[k][i]) depend(out:M[i][i])  
            syrk (M[k][i], M[i][i]);  
        }  
    }  
}
```

To support offline real-time analysis, it is necessary that the full DAG is known offline (including for loop iterations and if conditions). If this is not possible to be done offline, then assumptions on pessimistic behaviour need to be considered, leading to pessimistic schedulability analysis.



Support to Real-Time Parallel Models

- Although enabling the use of real-time parallel models, OpenMP does not support real-time scheduling properties
 - Such as deadlines, time bases or worst-case execution time
 - And the notion of task priorities is very relaxed (it is a “hint” to the scheduler)
- OpenMP also does not specify how the team threads are scheduled, being this left to the implementation
 - This makes it impossible to provide any real-time scheduling control at the programming level

Support to Real-Time Parallel Models

- A proposal was made to define an OpenMP profile for critical real-time systems
 - Tightening of the semantics of the priority clause of tasks, which would provide for priority driven task scheduling algorithms
 - Specification of a deadline property for tasks, allowing for dynamic scheduling algorithms
 - Specification of event, to enable the recurrent release of a task either by a time base, or by a sporadic event

```
#pragma omp task priority(p1)
{
    RT_task();
}

#pragma omp task deadline(d1) event(e1)
{
    RT_task();
}
```

- This proposal was not considered, being the decision to leave the real-time behaviour as an implementation approach
 - Work is continuing in this approach

Communication and Synchronization

- OpenMP provides for small set of mechanisms to support data sharing and synchronization
 - The depend clause
 - The critical construct and locks
- However, there is no support for any kind of priority inheritance or ceilings
 - Priorities are not a first-class entity in OpenMP
- An implementation may decide to build OpenMP threads on top of pthreads and critical/locks using mutexes with priority inheritance support
 - But without any mechanism at the programming level to control this behaviour

Control of Mapping and Scheduling

- The OpenMP tasking model provides a two-level mapping and scheduling approach
 - Tasks specified within the program are mapped to the team of the parallel region
 - The threads of the team are then executed by the operating system
- 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), and recently work-stealing (e.g. LLVM)
 - The use of these dynamic mapping approaches increase the pessimism of the real-time analysis (needs to consider all potential corner cases)
 - It is feasible for untied tasks, since execution is work conserving (no threads remain idle when there is a work to be performed)
 - For tied tasks, analysis is too pessimistic to be useful
- Static mapping of tasks to the threads makes the system more predictable
 - Allows for a more precise (less pessimistic) analysis

Control of Mapping and Scheduling

- OpenMP makes teams from two parallel regions independent, and oblivious of the other teams
 - Although many real-time works consider the schedulability analysis of a single DAG, it is possible that an application is required to implement parallelism in more than one of its concurrent activities
- A proposal to address this was provided together with the proposal to support time properties

```
#pragma omp parallel
#pragma omp single
{
    #pragma omp task priority (p1) // T1 : OpenMP-DAG1
    { RT_task_1() }
    #pragma omp task priority (p2) // T2 : OpenMP-DAG2
    { RT_task_2() }
    // ...
    #pragma omp task priority (pn) // Tn : OpenMP-DAGn
    { RT_task_n() }
}
```

Only one parallel region for all DAGs (not yet accepted)

- However, this approach precludes the logical and data separation of activities
 - No logical code separation
 - Same stack and data access

Control of Mapping and Scheduling

- Alternative would be to create pthreads of different priorities with internal parallelism

```
void *T1(void *arg) {  
    // T1 : OpenMP-DAG1  
    #pragma omp parallel {  
        #pragma omp single {  
            RT_Task_1();  
        }  
    }  
    return NULL;  
}
```

```
void *T2(void *arg) {  
    // T2 : OpenMP-DAG2  
    #pragma omp parallel {  
        #pragma omp single {  
            RT_Task_2();  
        }  
    }  
    return NULL;  
}
```

```
// ...
```

```
void main(){  
    //...  
  
    pthread_attr_init(&custom_attr_fifo1);  
    pthread_attr_setschedpolicy(&custom_attr_fifo1, SCHED_FIFO);  
    fifo_param1.sched_priority = p1;  
    pthread_attr_setschedparam(&custom_attr_fifo1, &fifo_param1);  
    pthread_create(&thread_id1, &custom_attr_fifo1, T1, NULL);  
  
    pthread_attr_init(&custom_attr_fifo2);  
    pthread_attr_setschedpolicy(&custom_attr_fifo2, SCHED_FIFO);  
    fifo_param1.sched_priority = p2;  
    pthread_attr_setschedparam(&custom_attr_fifo2, &fifo_param2);  
    pthread_create(&thread_id1, &custom_attr_fifo2, T2, NULL);  
    // ...
```

There is no mechanism to specify that the priority of the thread is propagated to the thread team.

But an implementation of the runtime can do it (e.g. UpScale, AMPERE).

Heterogenous Platforms

- Most of the work done so far targets multicore homogenous platforms
- OpenMP supports acceleration with the target directive
 - This is compatible with the tasking model
 - Therefore, the application of the proposed model to accelerator devices is mostly straightforward
- It is nevertheless still necessary to address
 - How to control the scheduling of the data transfers and concurrent execution
 - The interplay of the accelerator execution with CPU
 - Particularly considering that multiple DAGs can be executing at different priority levels

Summary

OpenMP for Real-Time Systems

- Current research to support real-time parallel systems comes from the intersection of activities of groups from different communities: high-performance computing, real-time and critical systems
 - OpenMP comes from the high-performance community, based on a complete and complex parallel programming model
 - Ada is from the critical and real-time systems communities, providing extensive and complex support for programming real-time systems.
- The use of the OpenMP tasking model in real-time systems is still the subject of research, particularly on
 - Approach for specifying real-time properties
 - Integration of multiple parallel real-time activities in the same application
 - Control the mapping of tasks to threads for more predictable (and less pessimistic) analysis
- This work was started in the scope of the P-SOCRATES European project (which led to the development of the UpScale SDK) and continued in the scope of the AMPERE European project

