Timing Characterization of OpenMP4 Tasking Model

Alessandra Melani

OpenMP and Embedded Systems

- Convergence of **High Performance Computing** (HPC) and **Embedded Computing** (EC)
  - High-end EC systems are increasingly requiring HPC-like performance in real time
- **Parallel programming models** for massive parallelism exploitation in a predictable way
  - Reduce the complexity of parallel programming
  - Abstraction level
- OpenMP is **widely used** in the HPC domain and is increasingly adopted in the EC domain as well

OpenMP and Embedded Systems

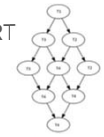
- How about **timing predictability**?
  - The recent specification v4.0 offers a sophisticated **tasking** execution model, which shares certain similarities with traditional real-time task graphs
  - But is completely agnostic to timing requirements
- Key questions
  - Can OpenMP tasks be used to describe a real-time application?
  - How to enable classical timing analysis and real-time scheduling within the OpenMP tasking model?
  - How to use standard real-time scheduling techniques without violating the semantics of the OpenMP execution model?

OpenMP and Embedded Systems

- OpenMP4 tasking model allows expressing fine-grained and irregular parallelism
  - **Task** (independent parallel unit of work)
  - **Data dependencies**

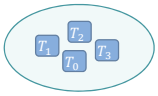
```
#pragma omp task depend(in: a)
                 depend(out: b)
{
  int c = 10;
  b = a + c;
}
```

- OpenMP4 tasking model resembles the way RT application are modelled
  - Sporadic Directed Acyclic Graph (DAG) model

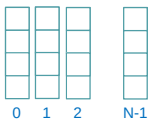


OpenMP4 Tasking Model

```
#pragma omp parallel num_threads(N)
#pragma omp task // T0
{
  p00
  #pragma omp task depend(out:x) // T1
    p10
  p01
  #pragma omp task depend(in:x) // T2
    p20
  p02
  #pragma omp task // T3
    p30
  p03
}
```



Task pool



Team of threads

OpenMP4 Tasking Model

```
#pragma omp parallel num_threads(N)
#pragma omp task // T0
{
  p00
  #pragma omp task depend(out:x) // T1
    p10
  p01
  #pragma omp task depend(in:x) // T2
    p20
  p02
  #pragma omp task // T3
    p30
  p03
}
```

Task Scheduling Points (TSPs):

- Points in the program where the task can be suspended and the hosting thread can be rescheduled to a different task
- They occur upon **task creation** and **completion**, and at task **synchronization points** such as `taskwait` and `barrier` directives

OpenMP4 Tasking Model

```
#pragma omp parallel num_threads(N)
#pragma omp task // T0
{
  p00
  #pragma omp task depend(out:x) // T1
    p10
  p01
  #pragma omp task depend(in:x) // T2
    p20
  p02
  #pragma omp task // T3
    p30
  p03
}
```

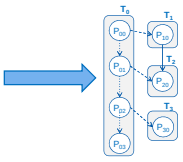
Task Scheduling Points (TSPs) divide tasks into parts executed uninterruptedly from start to end

Task part
- Uninterruptedly executed unit of code

OpenMP4 vs. DAG-based model

```
#pragma omp parallel num_threads(N)
#pragma omp task // T0
{
  p00
  #pragma omp task depend(out:x) // T1
    p10
  p01
  #pragma omp task depend(in:x) // T2
    p20
  p02
  #pragma omp task // T3
    p30
  p03
}
```

From an OpenMP program, an **OpenMP-DAG** can be derived



OpenMP4	DAG-based
Task parts	Nodes
Dependencies and TSPs	Edges
OpenMP programs	Tasks

OpenMP-DAG derivation

- Task parts correspond to nodes in the DAG, upon which WCET estimation is derived
- Edges are then incorporated in the DAG
  - `depend` clauses force tasks to be synchronized
  - Task creation also imposes a dependency relation
  - The same holds for `taskwait` and other synchronization directives

This OpenMP-DAG contains all information to derive a real-time schedule that complies with the semantics of the OpenMP tasking execution model

OpenMP-DAG derivation

- Task parts correspond to nodes in the DAG, upon which WCET estimation is derived
- Edges are then incorporated in the DAG
  - `depend` clauses force tasks to be synchronized
  - Task creation also imposes a dependency relation
  - The same holds for `taskwait` and other synchronization directives

DAG scheduling techniques can be applied to OpenMP-DAGs to provide **timing guarantees**

However, some features in the OpenMP tasking model complicate the analysis, due to **backward compatibility…**

OpenMP backward compatibility

Up to OpenMP 2.5: **Thread** based model

```
#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_id()==1)
    work1(i);
  else
    work2(i);
}
```

Thread 1 will compute `work1()`
Thread 0 will compute `work2()`

Since OpenMP 3.0: **Task** based model

```
#pragma omp parallel num_threads(2)
{
  #pragma omp task
    work3(i);
  #pragma omp task
    work4(i);
}
```

Any thread will compute `work3()` and `work4()`

OpenMP backward compatibility

Up to OpenMP 2.5: **Thread** based model

```
#pragma omp parallel num_threads(2)
{
  if (omp_get_thread_id()==1)
    work1(i);
  else
    work2(i);
}
```

Aware of threads

Since OpenMP 3.0: **Task** based model

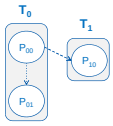
```
#pragma omp parallel num_threads(2)
{
  #pragma omp task
    work3(i);
  #pragma omp task
    work4(i);
}
```

Not aware of threads

Thread/task model compatibility

- Backward compatibility

```
#pragma omp task
{
  if (omp_get_thread_id()==1)
  {
     P00
     #pragma omp task { P10 }
     P01
  }
}
```

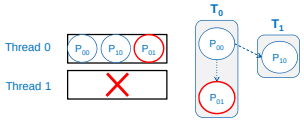


- Make both models compatible
  - **tied** vs. **untied** tasks

Tied vs. untied tasks

- `tied` tasks (default): compatible with both thread- and task-based model

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

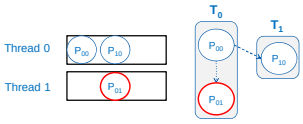


- **Can only be executed by the thread that started it**
- Task scheduling constraint (TSC): A new **tied** task can only be scheduled in a thread if it is a descendant of all the other tasks suspended in that thread

Tied vs. untied tasks

- `untied` tasks: compatible with only task-based model

```
#pragma omp task untied
{
     ...
}
```



- **Can be resumed by any thread after being suspended**
- Not subject to TSC

Why Task Scheduling Constraint?

- It prevents the run-time from **deadlocks**

```
#pragma omp task // Task A
{
     #pragma omp critical
     {
          #pragma omp task // Task C
          {
          }
          #pragma omp taskyield
     }
}

#pragma omp task // Task B
{
     #pragma omp critical
     {
     }
}
```

- Tasks A, B, C are **tied** tasks
- The thread executing Task A is about to enter the critical `taskyield` region and the thread **owns the lock** associated with the critical region
- Because `taskyield` is a task scheduling point, the thread executing Task A may choose to suspend
- Tasks B and C are in the task pool. By TSC, the thread executing Task A cannot execute Task B because **it is not a descendant** of Task A. Only Task C can be scheduled at this point, because it is a descendant of Task A

Why Task Scheduling Constraint?

- It prevents the run-time from **deadlocks**

```
#pragma omp task // Task A
{
     #pragma omp critical
     {
          #pragma omp task // Task C
          {
          }
          #pragma omp taskyield
     }
}

#pragma omp task //
{
     #pragma omp cr
     {
     }
}
```

`taskyield`: The current task can be suspended in favor of a different task
But the lock is not released!

- Tasks A, B, C are **tied** tasks
- The thread executing Task A is about to enter the critical `taskyield` region and the thread **owns the lock** associated with the critical region
- Because `taskyield` is a task scheduling point, the thread executing Task A may choose to suspend
- Tasks B and C are in the task pool. By TSC, the thread executing Task A cannot execute Task B because **it is not a descendant** of Task A. Only Task C can be scheduled at this point, because it is a descendant of Task A

Why Task Scheduling Constraint?

- It prevents the run-time from **deadlocks**

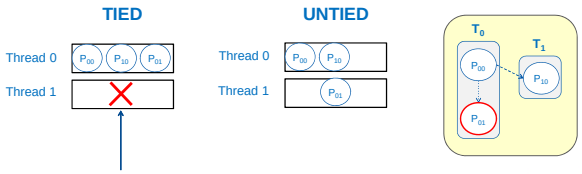
```
#pragma omp task // Task A
{
     #pragma omp critical
     {
          #pragma omp task // Task C
          {
          }
          #pragma omp taskyield
     }
}

#pragma omp task // Task B
{
     #pragma omp critical
     {
     }
}
```

- If Task B were to be scheduled, the thread to which Task A is tied cannot enter the critical region in Task B because the thread already holds the lock. Therefore, a **deadlock occurs**
- The purpose of TSC is to avoid this kind of deadlocks
- Note that a deadlock can also occur if the programmer nests a critical section inside Task C, but that would be a **programming error**

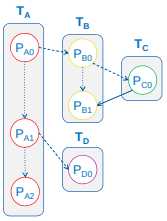
`tied` / `untied` tasks implications

- **Timing analysis**
  - In real-time systems, the use of **work-conserving** schedulers facilitates the timing characterization
    - Work-conserving schedulers never idle threads whenever workload is available
  - `tied` tasks are not compatible, `untied` tasks are



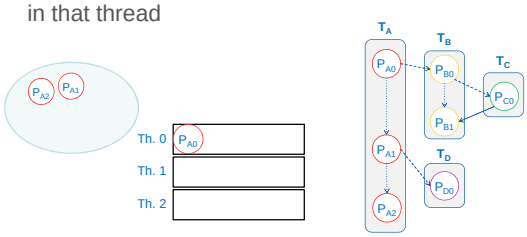
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



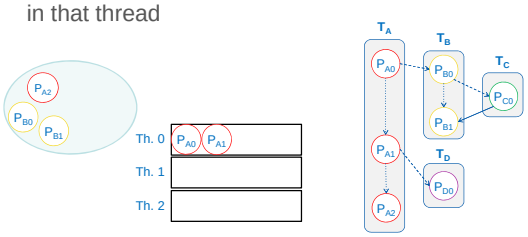
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



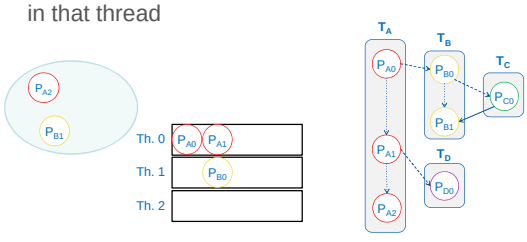
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



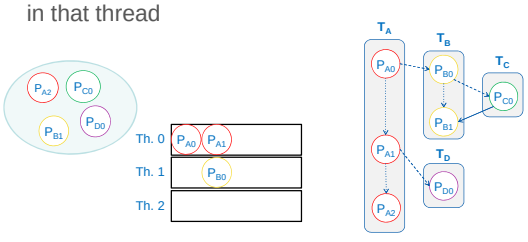
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



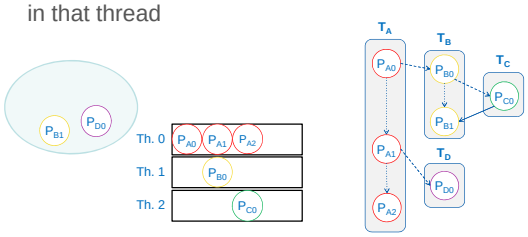
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



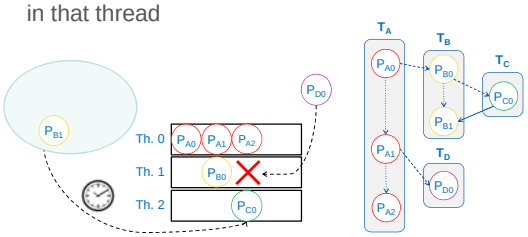
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



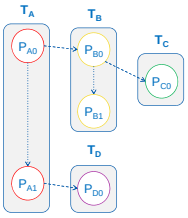
Impact of tied tasks on scheduling

- Reduced number of available threads for a new `tied` task
- 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



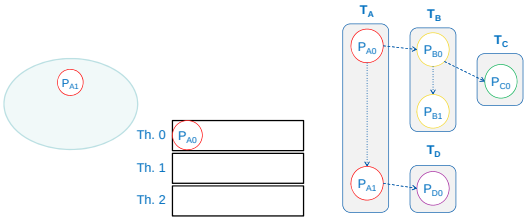
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



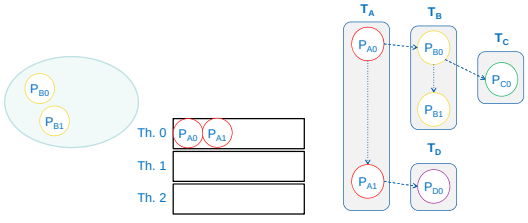
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



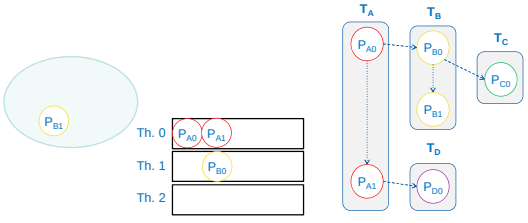
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



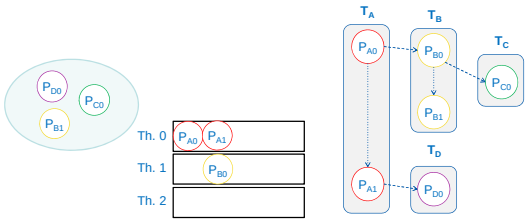
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



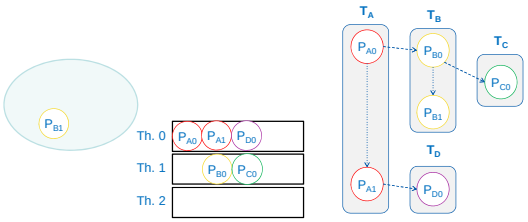
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



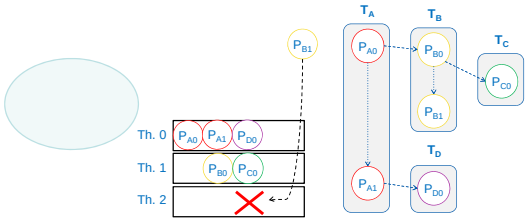
Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



Impact of tied tasks on scheduling

- `tied` tasks cannot resume their execution in a thread different than the one that started it



Impact of `tied` tasks on scheduling

- Impact on **time predictability**
  - Non-work conserving policy
  - Schedulability analysis without introducing unacceptable pessimism is prohibitive, or at least very difficult to achieve
- Impact on **performance**
  - The number of effective threads is reduced at task creation and resumption

Schedulability analysis of `untied` tasks

- **Work-conserving** policy
  - Tasks can be freely migrated across threads
  - Each task part is executed in one of the available threads as soon as all its dependencies have been fulfilled
- OpenMP4 DAG ↔ Real-time DAG task
  - `untied` tasks
  - WCET of each task part (nodes)
  - Relative deadline (D)
  - Period (T)

Schedulability test

Schedulability analysis of `untied` tasks

- Schedulability test
  - G: OpenMP-DAG
  - D: Relative deadline
  - m: number of processors/threads
  - $len(G)$: length of the critical path (longest chain)
  - $vol(G)$: sum of all WCETs of the nodes (volume)

$$R^{ub} = len(G) + \frac{1}{m}(vol(G) - len(G)) \leq D$$

Response time upper-bound

Relative deadline

Schedulability analysis of untied tasks

- Schedulability test
  - G: OpenMP-DAG
  - D: Relative deadline
  - m: number of processors/threads
  - $len(G)$: length of the critical path (longest chain)
  - $vol(G)$: sum of all WCETs of the nodes (volume)

$$R^{ub} = len(G) + \frac{1}{m}(vol(G) - len(G)) \leq D$$

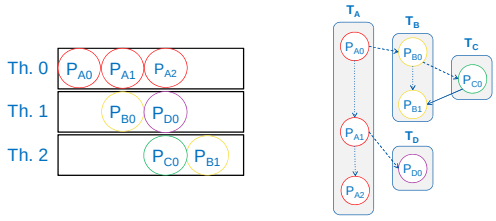
Critical path | Divided among processors | Rest of work

Schedulability analysis of untied tasks

- Schedulability of a DAG ≈ **Makespan minimization** problem
  - Makespan: response time of the collection of jobs
  - The makespan minimization problem is NP-hard
- Graham's **List Scheduling** algorithm
  - Polynomial time complexity
  - **Approximation bound**: $R \leq \left(2 - \frac{1}{m}\right) R^{opt}$
    - The obtained makespan is at most $\left(2 - \frac{1}{m}\right)$ times the optimal one
    - It implements a **work-conserving** scheduling strategy

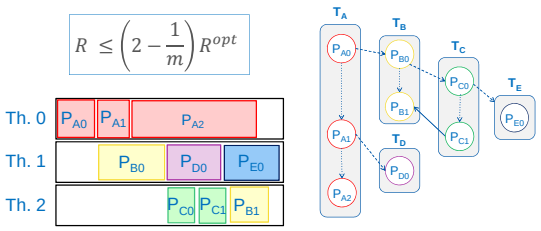
List Scheduling example

- Example



List Scheduling approximation bound

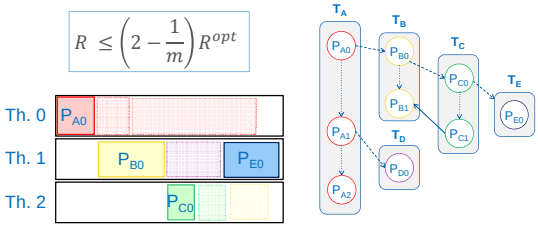
$$R \leq \left(2 - \frac{1}{m}\right) R^{opt}$$



- Proof
  - Construct a **critical chain** of jobs λ^* as follows
    - Take the job v_z that completes last, and let t_z be its starting time
    - Let v_{z-1} be the predecessor of v_z that completes last
    - Go on in this way until a job without predecessors is reached
  - We get a chain of jobs $\lambda^* = (v_1, \ldots, v_z)$

List Scheduling approximation bound

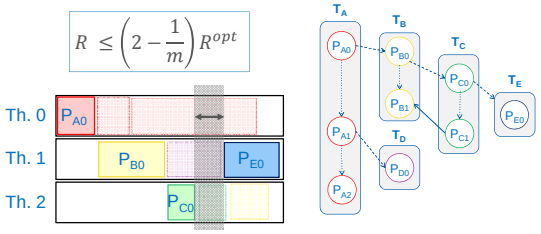
$$R \leq \left(2 - \frac{1}{m}\right) R^{opt}$$



- Proof
  - Construct a **critical chain** of jobs λ^* as follows
    - Take the job v_z that completes last, and let t_z be its starting time
    - Let v_{z-1} be the predecessor of v_z that completes last
    - Go on in this way until a job without predecessors is reached
  - We get a chain of jobs $\lambda^* = (v_1, \ldots, v_z)$

List Scheduling approximation bound

$$R \leq \left(2 - \frac{1}{m}\right) R^{opt}$$



- Proof (continued)
  - Observation: between the completion time $t_i + C_i$ of each job of λ^* and the starting time of the next job, **all threads must be busy** (otherwise job v_{i+1} would have started earlier)
  - Some job belonging to λ^* is executing at every time instant when not all threads are busy

List Scheduling approximation bound

- Proof (continued)
- R equal to the sum of
  - Time instants when some of the threads are **idle**:
    $\leq len(\lambda^*)$
  - Time instants when all the threads are **busy**:
    $\leq \frac{1}{m}(vol(G) - len(\lambda^*))$

  that gives

$$R \leq len(\lambda^*) + \frac{1}{m}(vol(G) - len(\lambda^*))$$

List Scheduling approximation bound

- Proof (continued)
  - Total work executed on m threads

$$R^{opt} \geq \frac{1}{m}\sum_{v_i \in V} C_i = \frac{1}{m} vol(G)$$

  - Longest chain executed sequentially

$$R^{opt} \geq \max_{\lambda \in G} \sum_{v_i \in \lambda} C_i = len(G)$$

List Scheduling approximation bound

- Proof (continued)

$$\begin{aligned} R &\leq len(\lambda^*) + \frac{1}{m}\left(vol(G) - len(\lambda^*)\right) \\ &= len(\lambda^*) + \frac{1}{m} vol(G) - \frac{1}{m} len(\lambda^*) \\ &\leq R^{opt} + R^{opt} - \frac{1}{m} R^{opt} \\ &= \left(1 - \frac{1}{m} + 1\right) R^{opt} = \left(2 - \frac{1}{m}\right) R^{opt} \end{aligned}$$

- Since $len(\lambda^*) \leq len(G)$, we obtain

$$R^{ub} = len(G) + \frac{1}{m}(vol(G) - len(G))$$

Conclusions

The OpenMP4 tasking model resembles the sporadic DAG scheduling model

However, to provide timing guarantees we need to consider the following OpenMP features

- `tied` tasks: do not allow the use of work-conserving schedulers, complicating the schedulability analysis
- `untied` tasks: allow the use of work-conserving schedulers

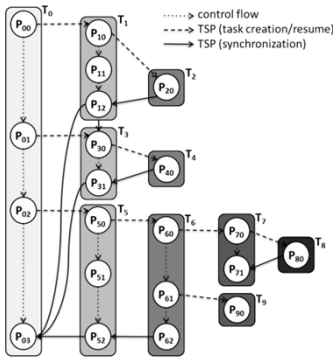
A **schedulability analysis** can be easily derived for the `untied` tasking model, enabling the applicability of OpenMP4 in real-time systems

Example

```
1  #pragma omp parallel num_threads(N) {
2  #pragma omp single { // T0
3    part00
4    #pragma omp task depend(out:x) // T1
5    { part10
6      #pragma omp task  { part20 } // T2
7      part11
8      #pragma omp taskwait
9      part12
10   }
11   part01
12   #pragma omp task depend(in:x) // T3
13   { part30
14     #pragma omp task { part40 } // T4
15     #pragma omp taskwait
16     part31
17   }
18   part02
19   #pragma omp task // T5
20   { part50
21     #pragma omp task { // T6
22       part60
23       #pragma omp task  // T7
24       { part70
25         #pragma omp task { part80 } // T8
26         #pragma omp taskwait
27         part71
28       }
29       part61
30       #pragma omp task { part90 } // T9
31       part62
32     }
33     part51
34     #pragma omp taskwait
35     part52
36   }
37   #pragma omp taskwait
38   part03
39 }}
```

What is the corresponding OpenMP-DAG?

Solution



Thank you!

Alessandra Melani
alessandra.melani@sssup.it