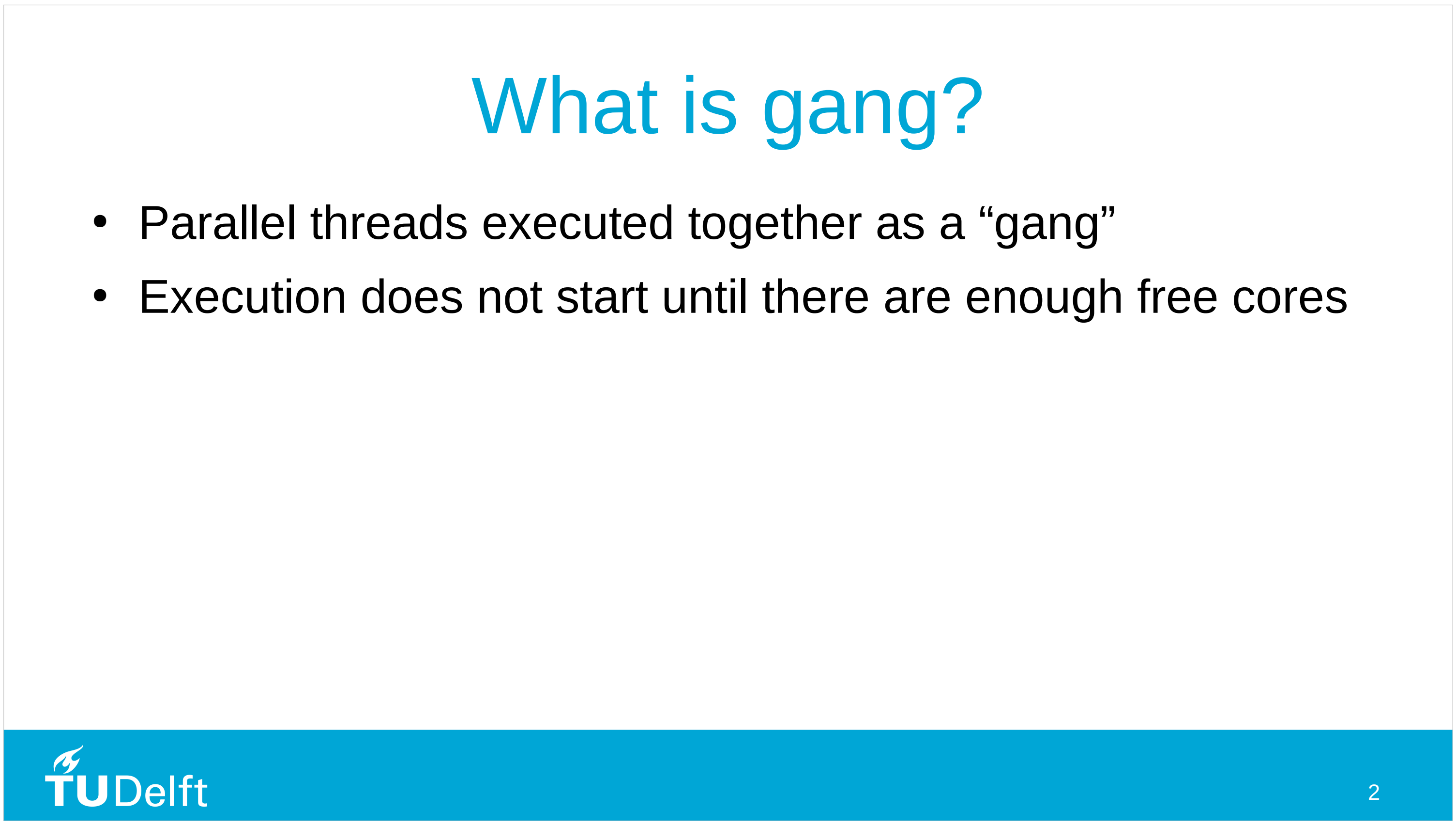
Scheduling and Analys of Limited-Preemptive Modable Gang Tasks

Joan Marcè i Igual   Geoffrey Nelissen   Mitra Nasri   Paris Panagiotou

24th of February, 2020

TUDelft



What is gang?

- Parallel threads executed together as a “gang”
- Execution does not start until there are enough free cores

TUDelft

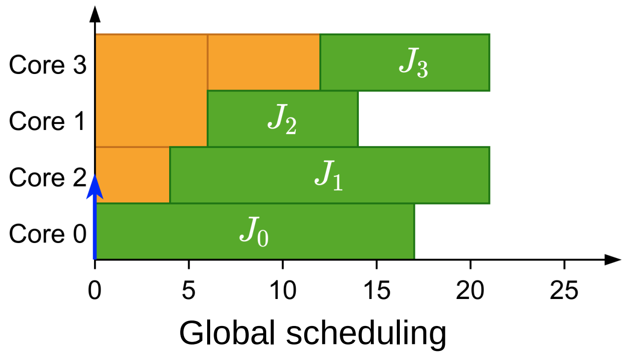
First of all, let’s explain what gang scheduling is.

It’s the execution of multiple parallel threads together as a “gang”. In these threads the execution does not start until there are enough cores to execute them together.
Let’s see an example:<click>

1. We have these jobs assigned to different cores.
2. If we release them as a gang job then we should pack them like a single task. Obtaining the following group. <click>
3. Then,<click> we have obtained the gang task result of merging the jobs

What is gang?

- Parallel threads executed together as a “gang”
- Execution does not start until there are enough free cores



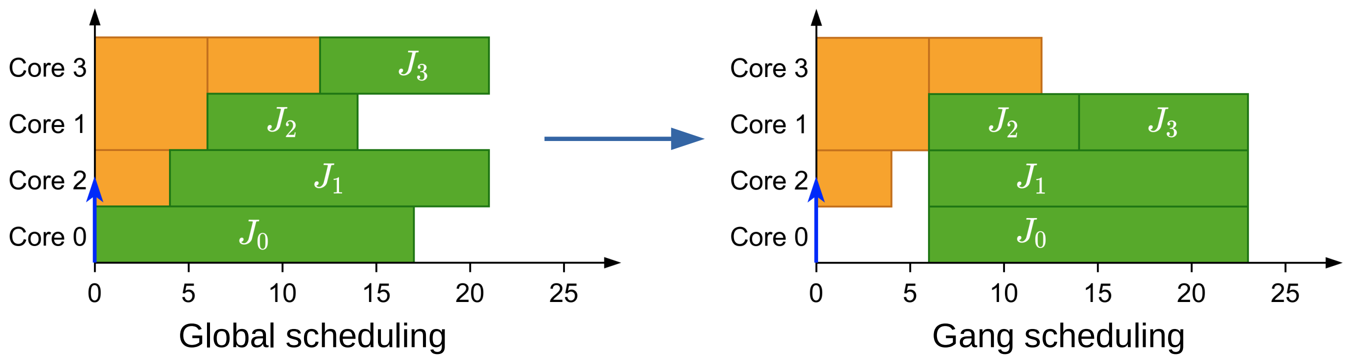
First of all, let’s explain what gang scheduling is.

It’s the execution of multiple parallel threads together as a “gang”. In these threads the execution does not start until there are enough cores to execute them together.
Let’s see an example:<click>

1. We have these jobs assigned to different cores.
2. If we release them as a gang job then we should pack them like a single task. Obtaining the following group. <click>
3. Then,<click> we have obtained the gang task result of merging the jobs

What is gang?

- Parallel threads executed together as a “gang”
- Execution does not start until there are enough free cores



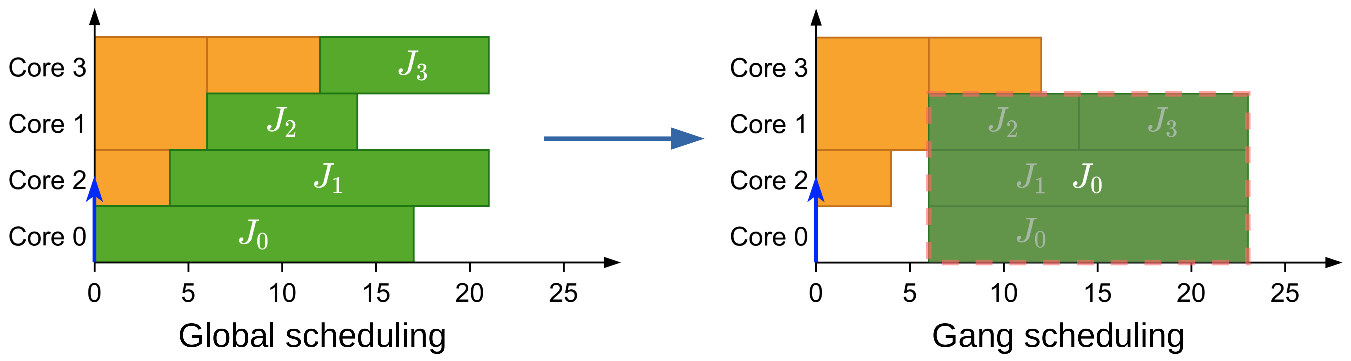
First of all, let’s explain what gang scheduling is.

It’s the execution of multiple parallel threads together as a “gang”. In these threads the execution does not start until there are enough cores to execute them together.
Let’s see an example:<click>

1. We have these jobs assigned to different cores.
2. If we release them as a gang job then we should pack them like a single task. Obtaining the following group. <click>
3. Then,<click> we have obtained the gang task result of merging the jobs

What is gang?

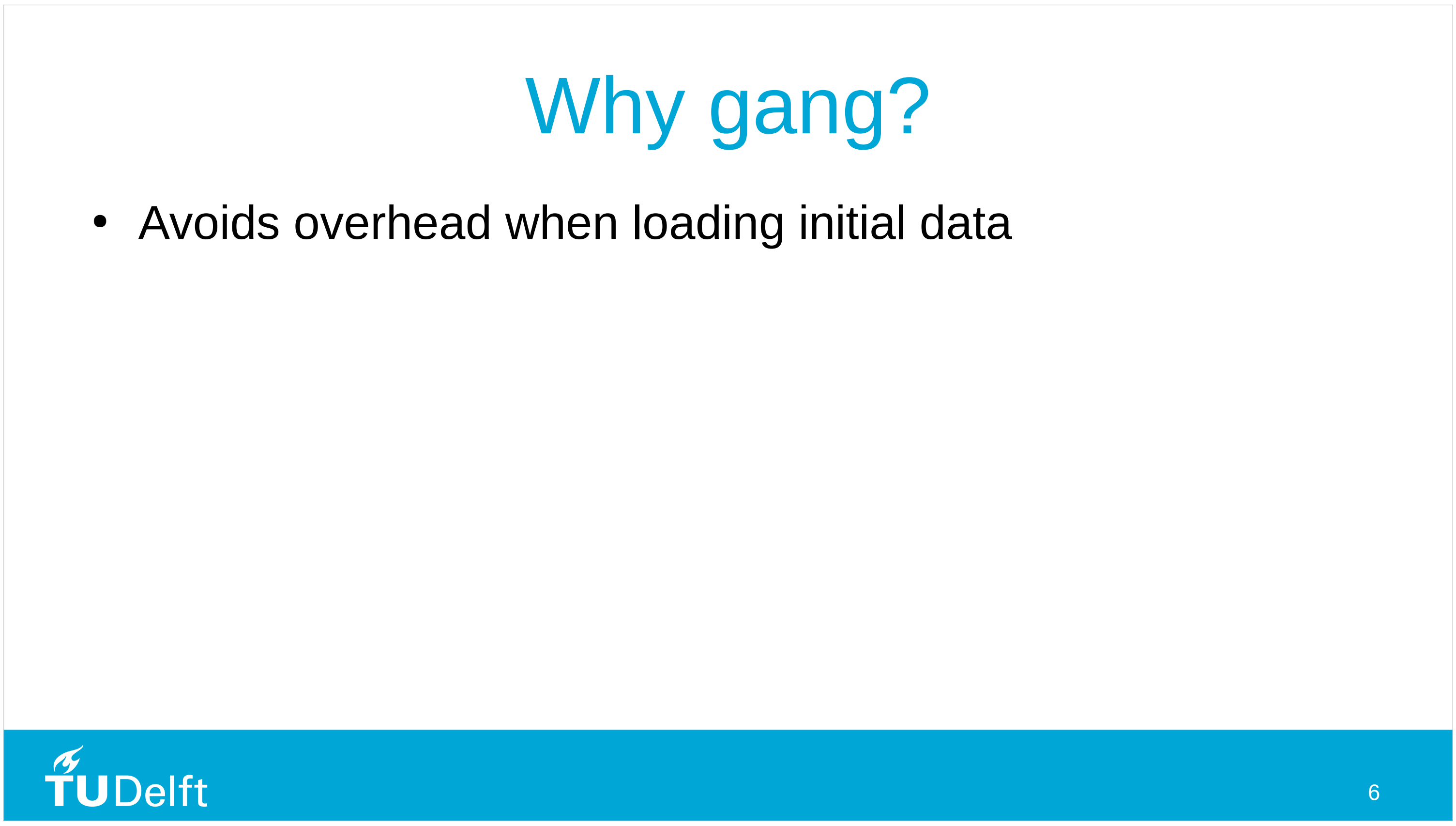
- Parallel threads executed together as a “gang”
- Execution does not start until there are enough free cores



First of all, let’s explain what gang scheduling is.

It’s the execution of multiple parallel threads together as a “gang”. In these threads the execution does not start until there are enough cores to execute them together.
Let’s see an example:<click>

1. We have these jobs assigned to different cores.
2. If we release them as a gang job then we should pack them like a single task. Obtaining the following group. <click>
3. Then,<click> we have obtained the gang task result of merging the jobs



Why gang?

- Avoids overhead when loading initial data

TU Delft

So then, why do we need gang scheduling?

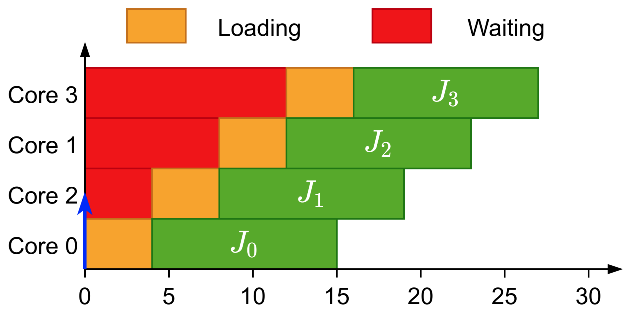
Well, it helps in reducing overhead when loading data initially.

<click>
In this example all these threads have been scheduled at the same time but have to wait until the previous one have finished loading.

On the contrary<click>, if we know that all these threads are starting at the same time we can load the data once for all the threads at the same time which is what we would do with gang scheduling.

Why gang?

- Avoids overhead when loading initial data



Global scheduling

So then, why do we need gang scheduling?

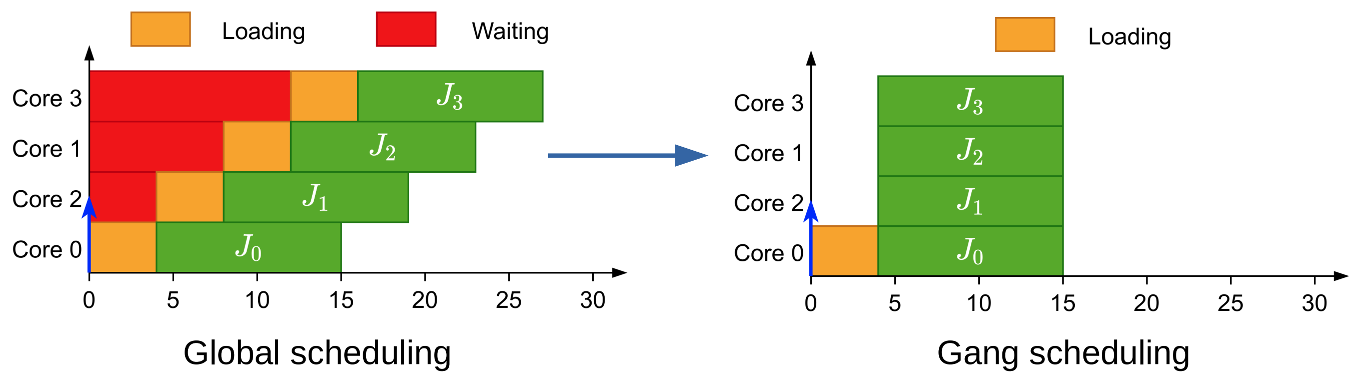
Well, it helps in reducing overhead when loading data initially.

<click>
In this example all these threads have been scheduled at the same time but have to wait until the previous one have finished loading.

On the contrary<click>, if we know that all these threads are starting at the same time we can load the data once for all the threads at the same time which is what we would do with gang scheduling.

Why gang?

- Avoids overhead when loading initial data



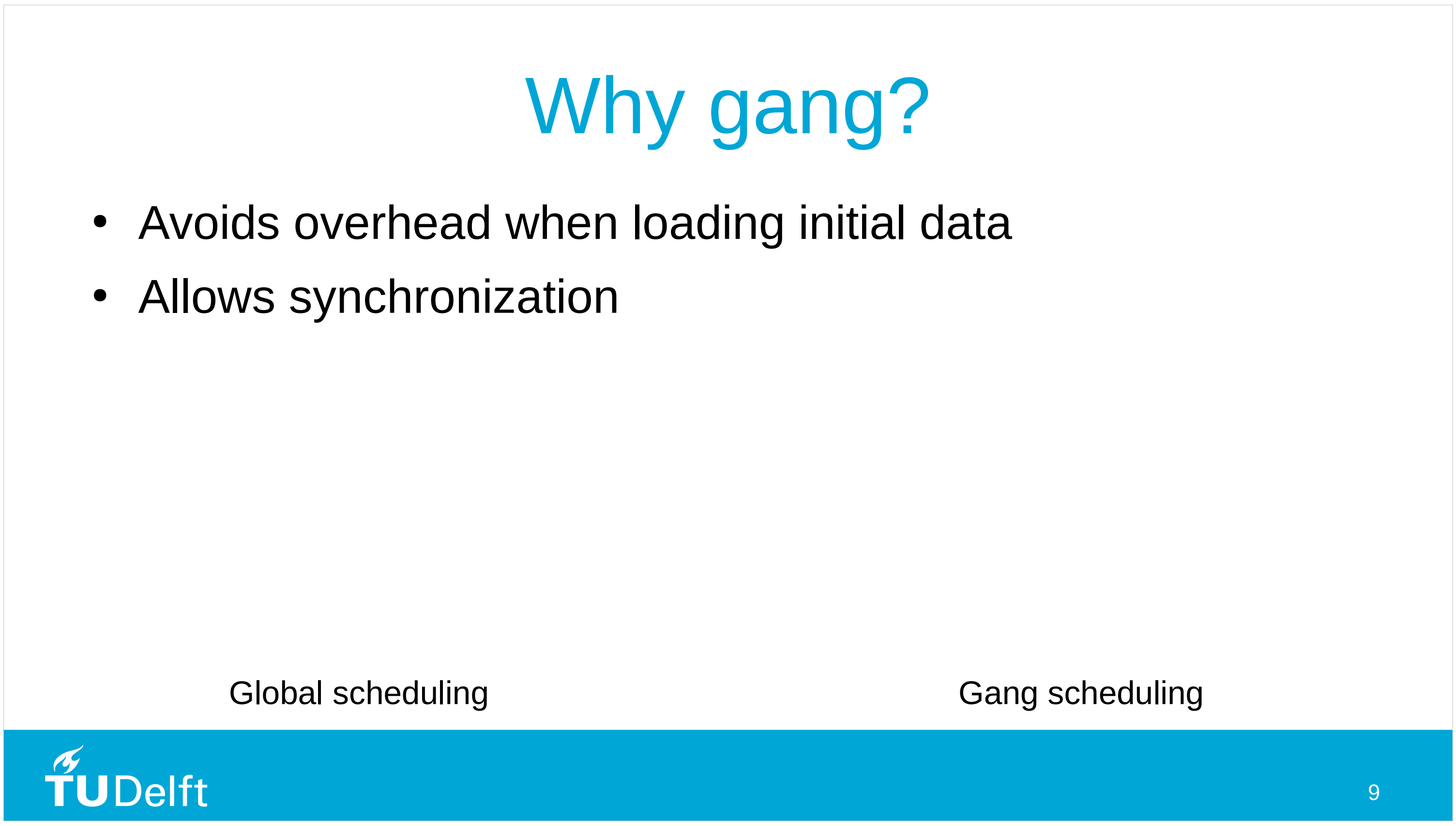
TUDelft

So then, why do we need gang scheduling?

Well, it helps in reducing overhead when loading data initially.

<click>
In this example all these threads have been scheduled at the same time but have to wait until the previous one have finished loading.

On the contrary<click>, if we know that all these threads are starting at the same time we can load the data once for all the threads at the same time which is what we would do with gang scheduling.



Why gang?

- Avoids overhead when loading initial data
- Allows synchronization

Global scheduling

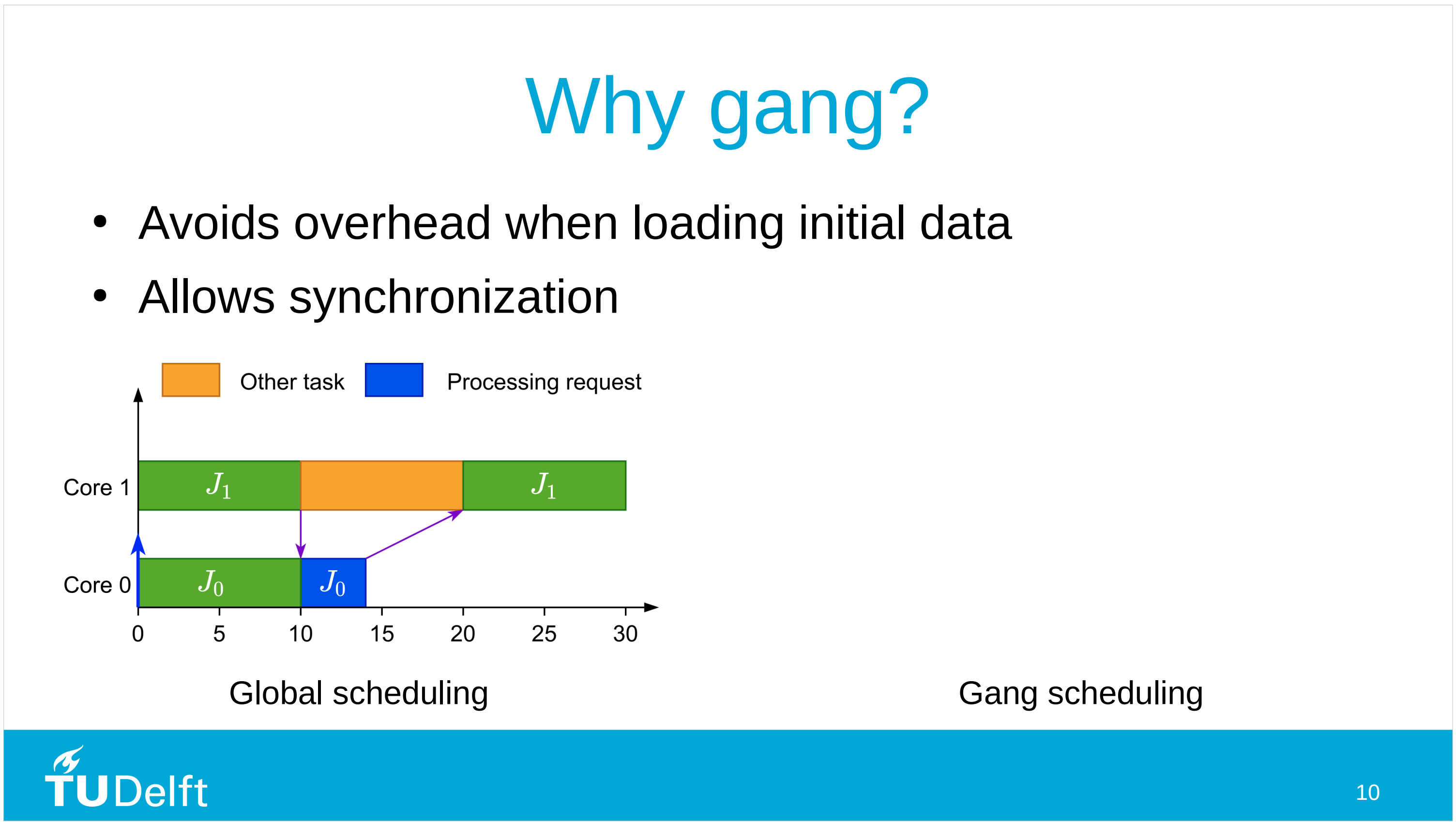
Gang scheduling

TUDelft

Additionally it helps in synchronization situations, like in the following example<click>

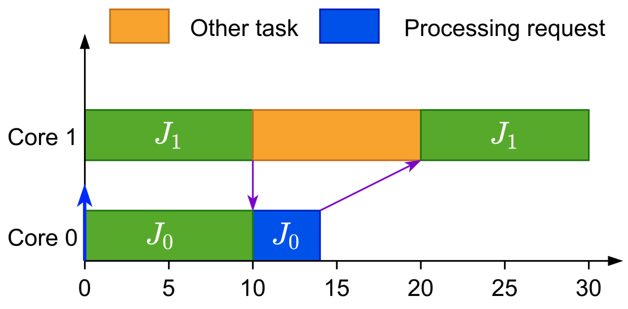
Here there are two jobs and job 1 sends some data to job 0 that is processed and sent back. In this case another task has been scheduled after job 1 sent the data and after the other task finishes then job 1 can continue its execution

On the other hand if we use gang<click> the job would wait for job 0 to finish the processing part and return the results as no other task would be allowed to execute while job 1 is waiting.



Why gang?

- Avoids overhead when loading initial data
- Allows synchronization



Global scheduling

Gang scheduling

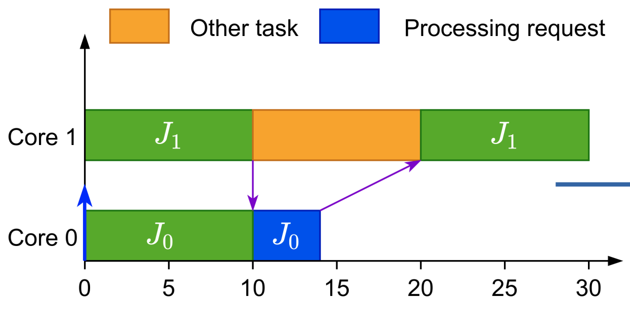
Additionally it helps in synchronization situations, like in the following example<click>

Here there are two jobs and job 1 sends some data to job 0 that is processed and sent back. In this case another task has been scheduled after job 1 sent the data and after the other task finishes then job 1 can continue its execution

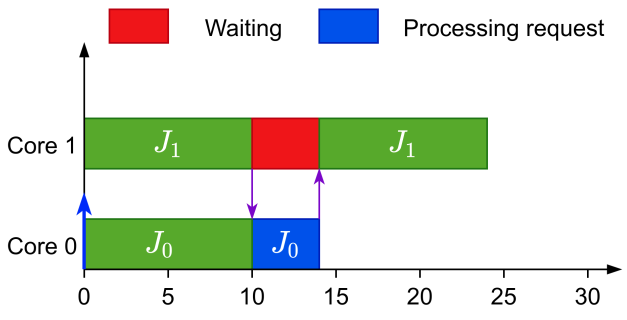
On the other hand if we use gang<click> the job would wait for job 0 to finish the processing part and return the results as no other task would be allowed to execute while job 1 is waiting.

Why gang?

- Avoids overhead when loading initial data
- Allows synchronization



Global scheduling

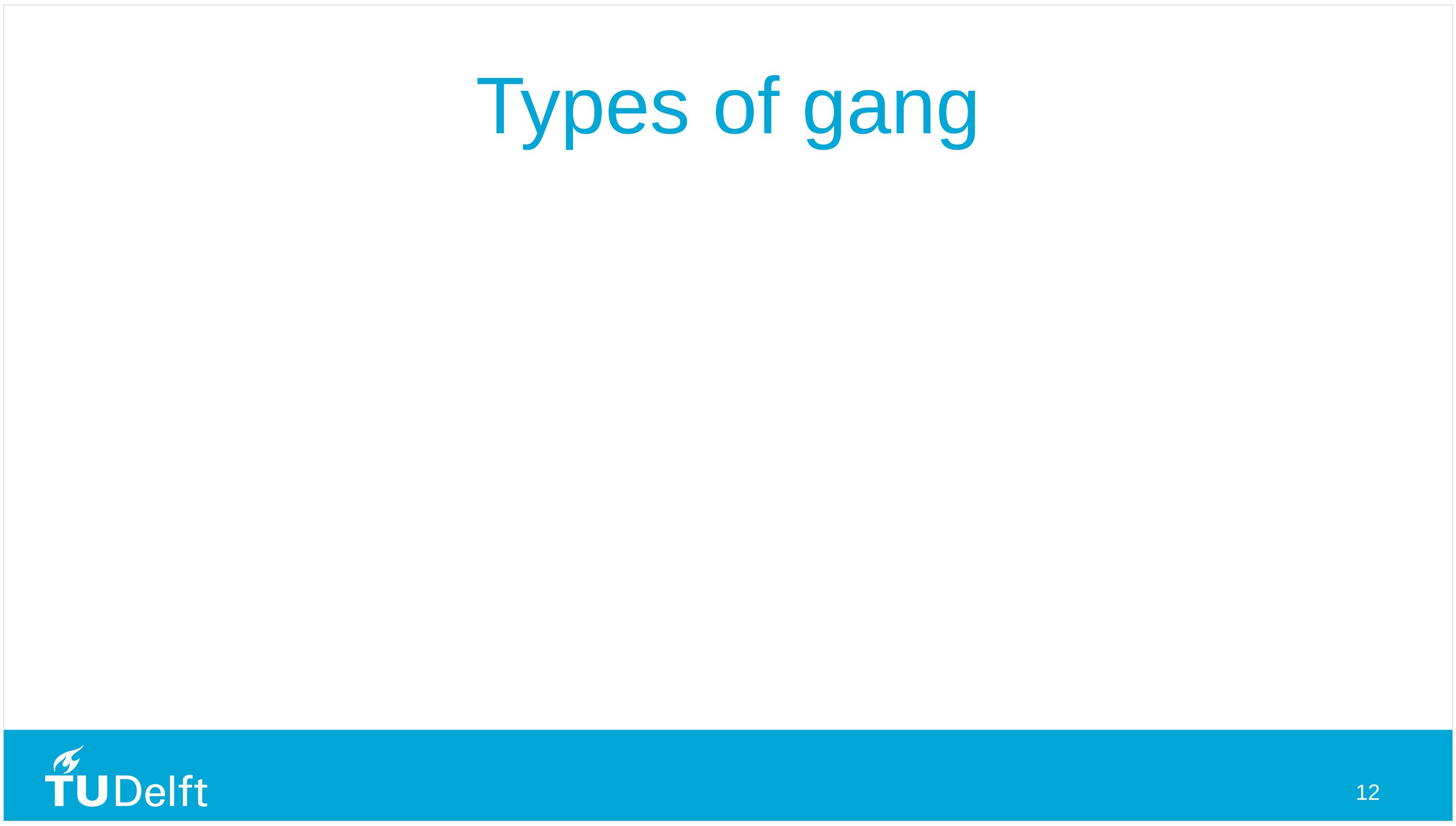


Gang scheduling

Additionally it helps in synchronization situations, like in the following example<click>

Here there are two jobs and job 1 sends some data to job 0 that is processed and sent back. In this case another task has been scheduled after job 1 sent the data and after the other task finishes then job 1 can continue its execution

On the other hand if we use gang<click> the job would wait for job 0 to finish the processing part and return the results as no other task would be allowed to execute while job 1 is waiting.

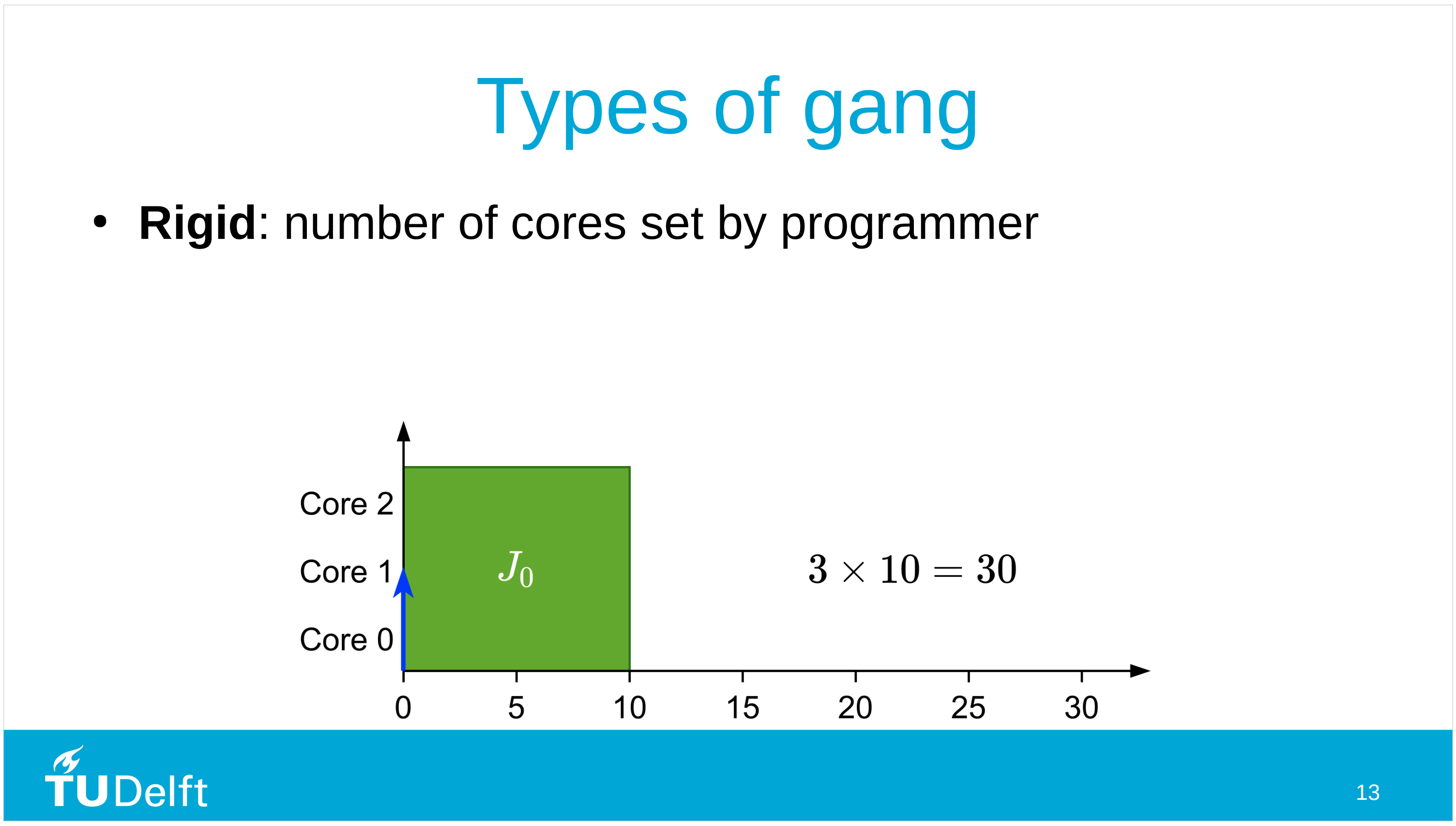


Types of gang

TUDelft

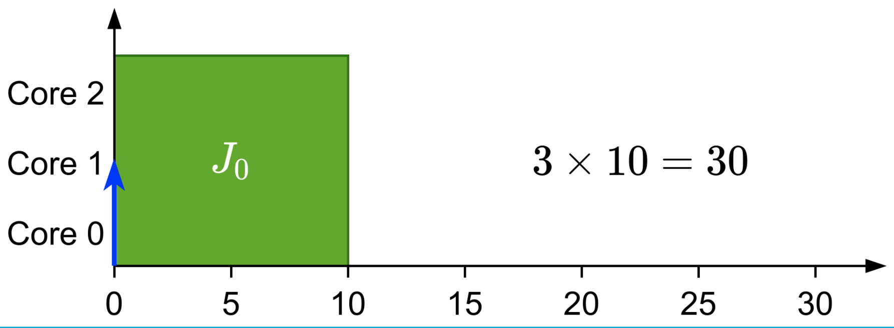
When talking about gang scheduling, we can have three types:

<click>Rigid gang, where the number of cores is set by the programmer and fixed during execution. In this example this job executes with 3 cores and requires 10 time units to compute



Types of gang

- **Rigid**: number of cores set by programmer

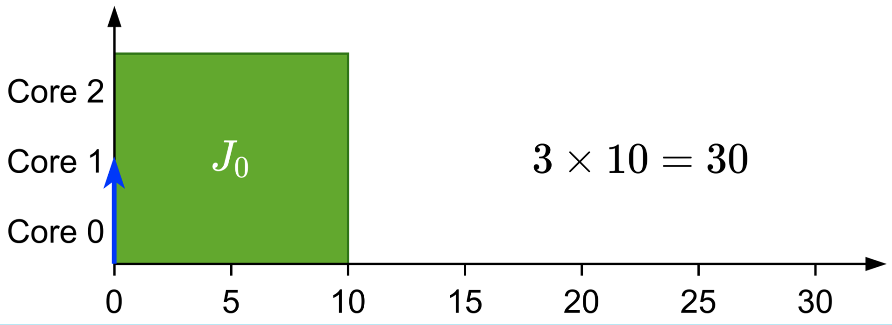


When talking about gang scheduling, we can have three types:

<click>Rigid gang, where the number of cores is set by the programmer and fixed during execution. In this example this job executes with 3 cores and requires 10 time units to compute

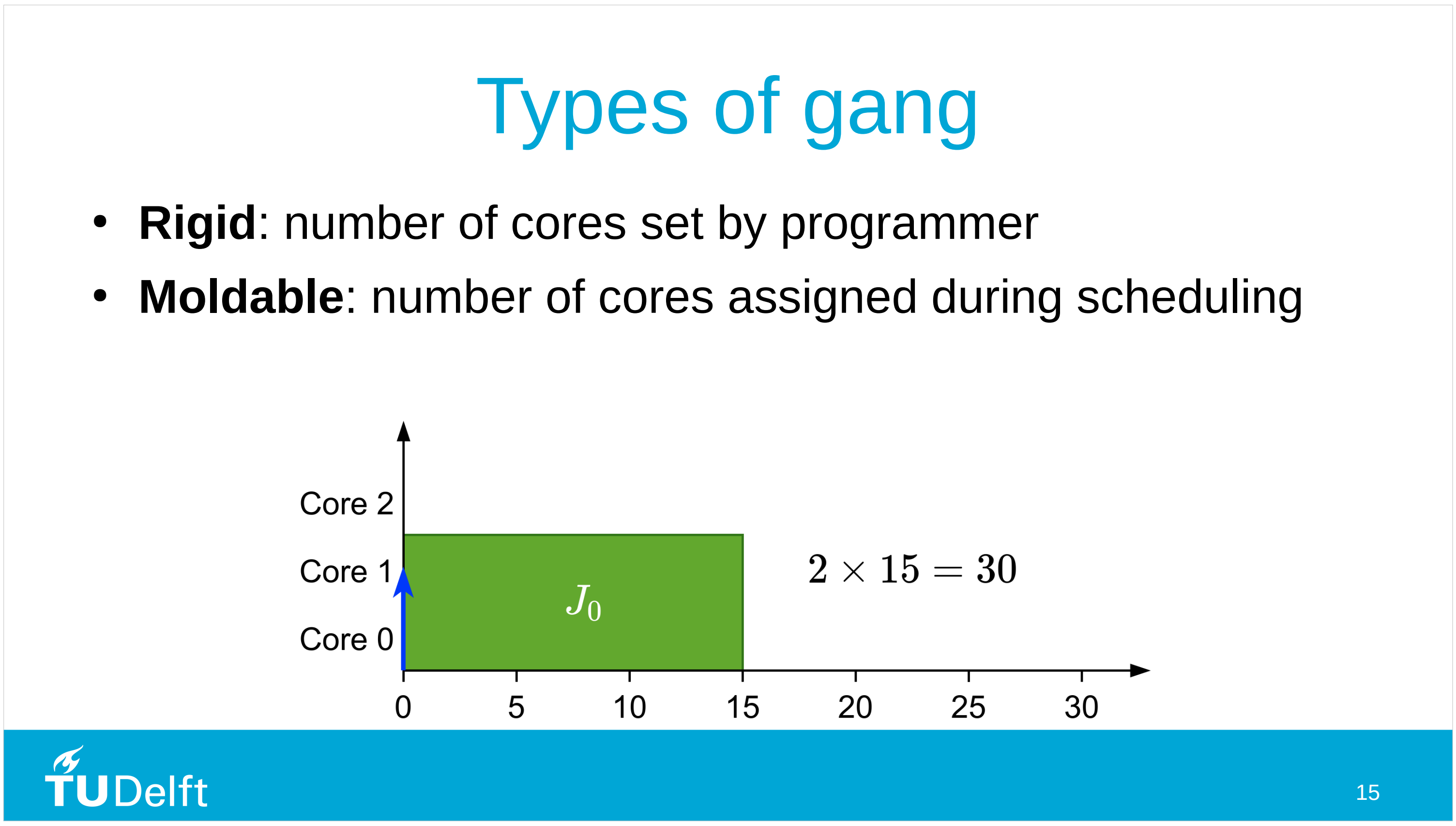
Types of gang

- **Rigid**: number of cores set by programmer
- **Moldable**: number of cores assigned during scheduling



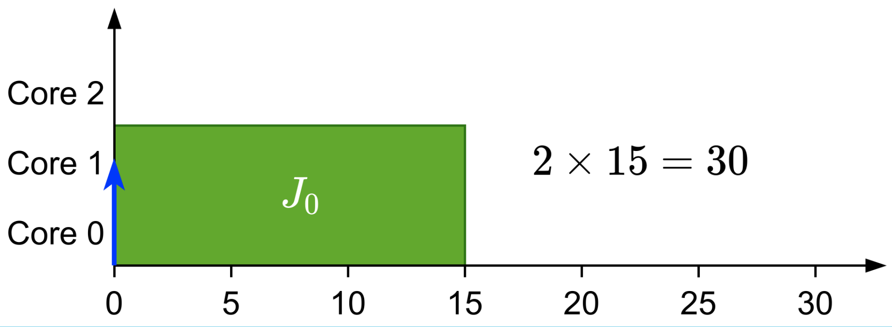
Moldable gang, where the number of cores can be in a range of values and the actual chosen value is decided when scheduling the task

In the example now the job can also execute <click> in two cores and <click> in one core. While also increasing the execution time.



Types of gang

- **Rigid**: number of cores set by programmer
- **Moldable**: number of cores assigned during scheduling

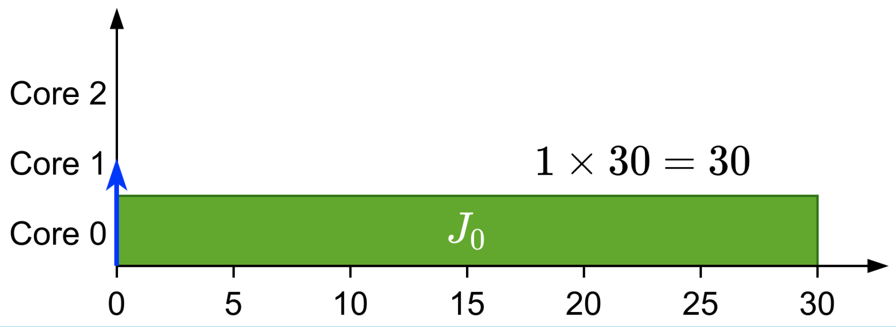


Moldable gang, where the number of cores can be in a range of values and the actual chosen value is decided when scheduling the task

In the example now the job can also execute <click> in two cores and <click> in one core. While also increasing the execution time.

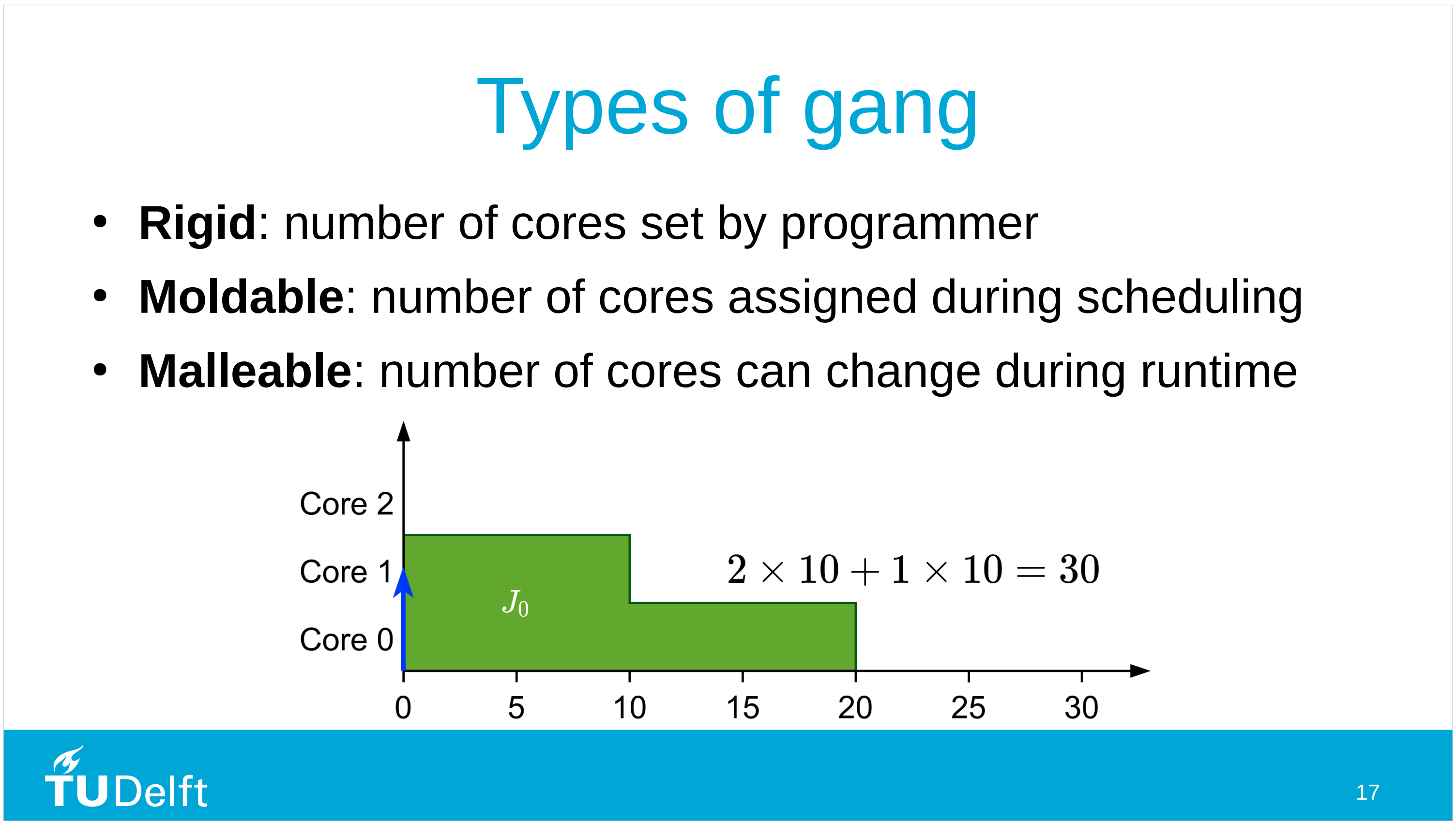
Types of gang

- **Rigid**: number of cores set by programmer
- **Moldable**: number of cores assigned during scheduling



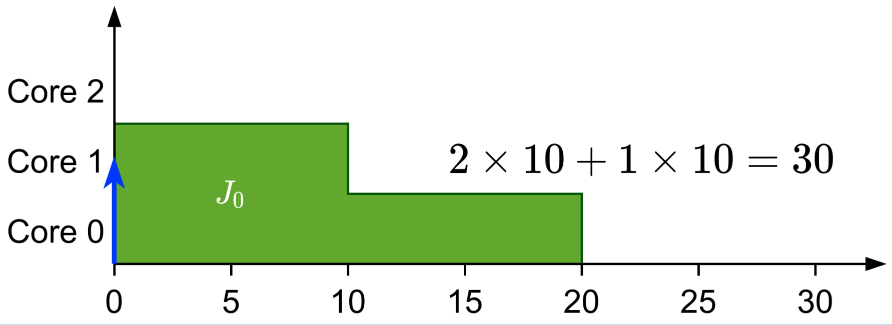
Moldable gang, where the number of cores can be in a range of values and the actual chosen value is decided when scheduling the task

In the example now the job can also execute <click> in two cores and <click> in one core. While also increasing the execution time.



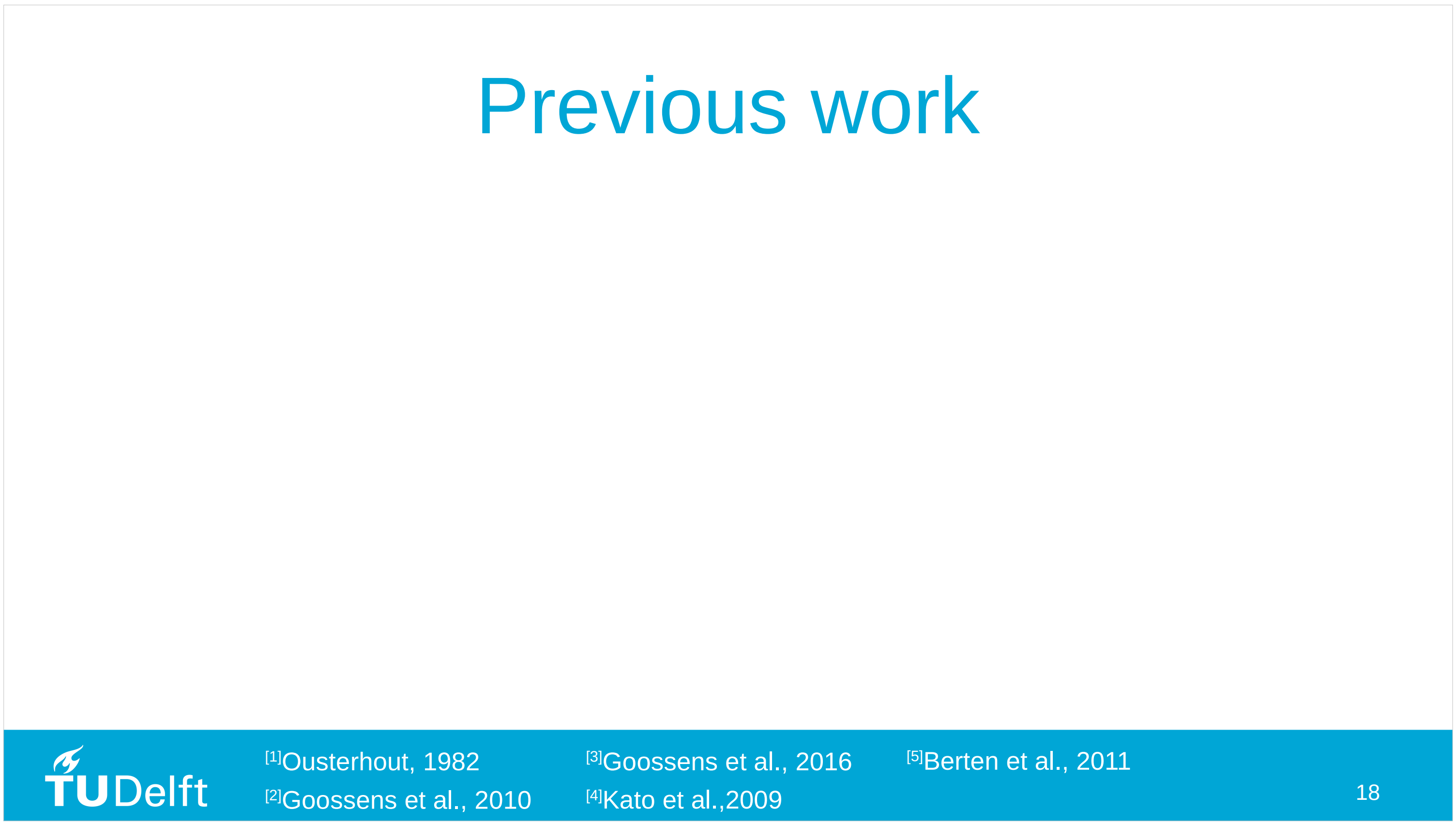
Types of gang

- **Rigid**: number of cores set by programmer
- **Moldable**: number of cores assigned during scheduling
- **Malleable**: number of cores can change during runtime



Finally there is malleable gang, where the number of cores can vary during the execution of the job and in the example the job starts executing with 2 cores and then at time 10 it switches to one core.

This work focuses on moldable gang



Previous work

[1]Ousterhout, 1982
[2]Goossens et al., 2010
[3]Goossens et al., 2016
[4]Kato et al.,2009
[5]Berten et al., 2011

Alright, so what has been done previously in gang scheduling? <click>First, the gang task model was introduced in high-performance computing as a way to reduce synchronization overhead and optimize access to shared data

<click>From a real-time viewpoint the results are rather limited.

- <click>For rigid gang taks it has been proven that the Job-Level Fixed-Priority scheduler is not predictable and an optimal scheduler was proposed, although it requires a high number of preemptions.
- <click>For moldable gang tasks, the global EDF scheduler has been adapted and studied. Additionally a preemptive scheduler was proposed that tries to be smart and choose the appropiate cores in order to meet deadlines.

Previous work

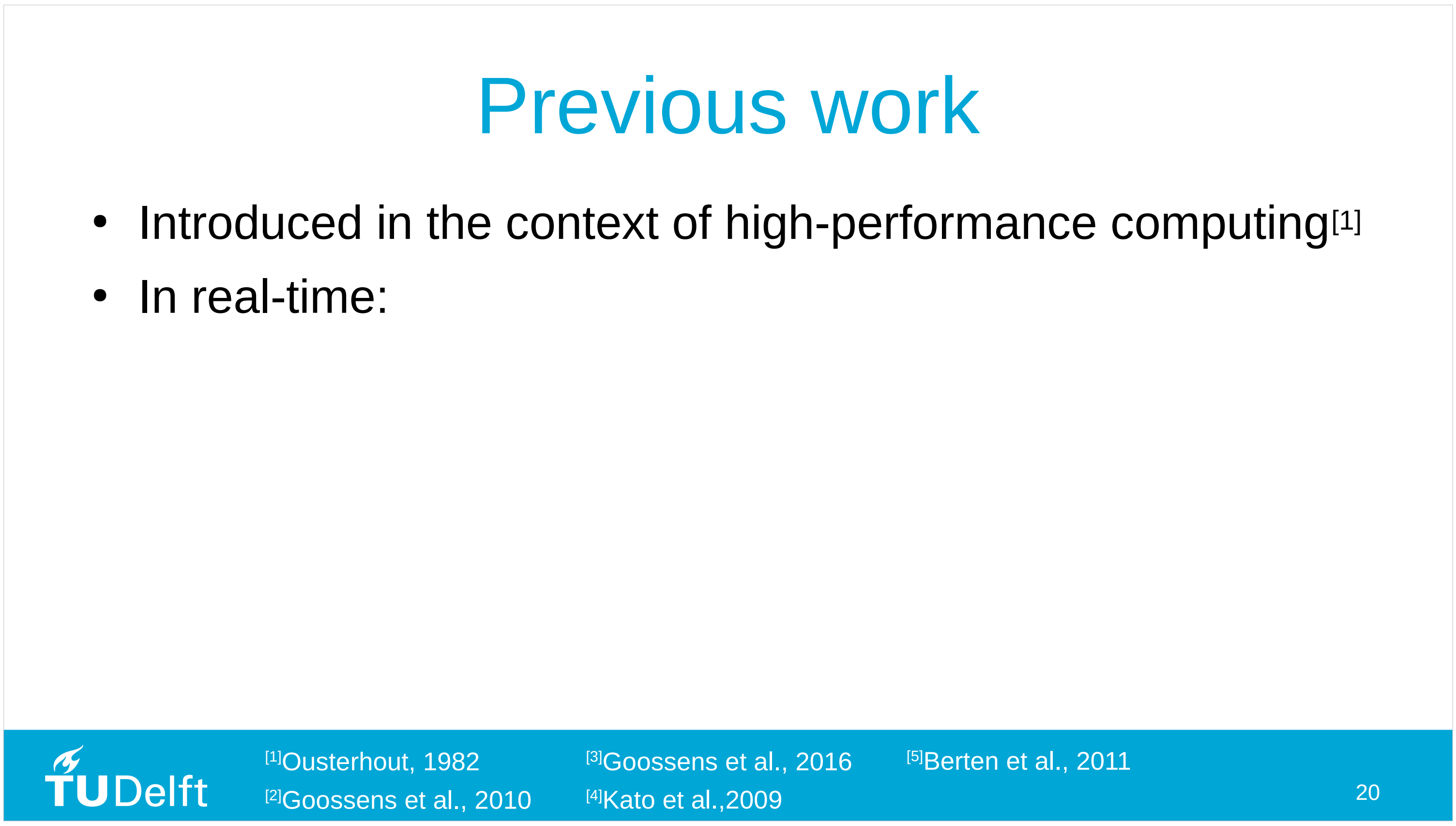
- Introduced in the context of high-performance computing[1]

[1]Ousterhout, 1982
[2]Goossens et al., 2010
[3]Goossens et al., 2016
[4]Kato et al.,2009
[5]Berten et al., 2011

Alright, so what has been done previously in gang scheduling? <click>First, the gang task model was introduced in high-performance computing as a way to reduce synchronization overhead and optimize access to shared data

<click>From a real-time viewpoint the results are rather limited.

- <click>For rigid gang taks it has been proven that the Job-Level Fixed-Priority scheduler is not predictable and an optimal scheduler was proposed, although it requires a high number of preemptions.
- <click>For moldable gang tasks, the global EDF scheduler has been adapted and studied. Additionally a preemptive scheduler was proposed that tries to be smart and choose the appropiate cores in order to meet deadlines.



Previous work

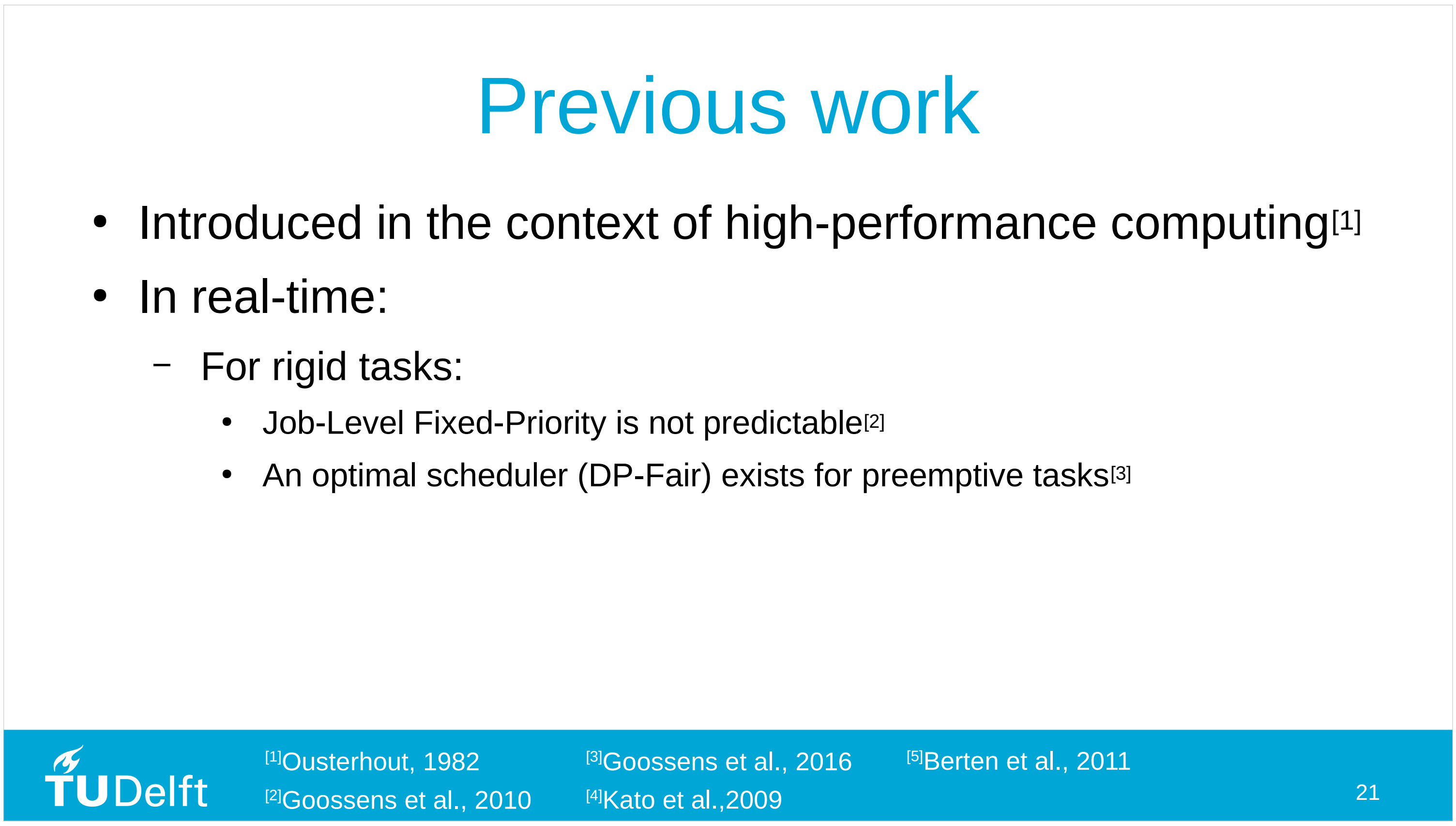
- Introduced in the context of high-performance computing[1]
- In real-time:

[1]Ousterhout, 1982
[2]Goossens et al., 2010
[3]Goossens et al., 2016
[4]Kato et al.,2009
[5]Berten et al., 2011

Alright, so what has been done previously in gang scheduling? <click>First, the gang task model was introduced in high-performance computing as a way to reduce synchronization overhead and optimize access to shared data

<click>From a real-time viewpoint the results are rather limited.

- <click>For rigid gang taks it has been proven that the Job-Level Fixed-Priority scheduler is not predictable and an optimal scheduler was proposed, although it requires a high number of preemptions.
- <click>For moldable gang tasks, the global EDF scheduler has been adapted and studied. Additionally a preemptive scheduler was proposed that tries to be smart and choose the appropiate cores in order to meet deadlines.



Previous work

- Introduced in the context of high-performance computing[1]
- In real-time:
  - For rigid tasks:
    - Job-Level Fixed-Priority is not predictable[2]
    - An optimal scheduler (DP-Fair) exists for preemptive tasks[3]

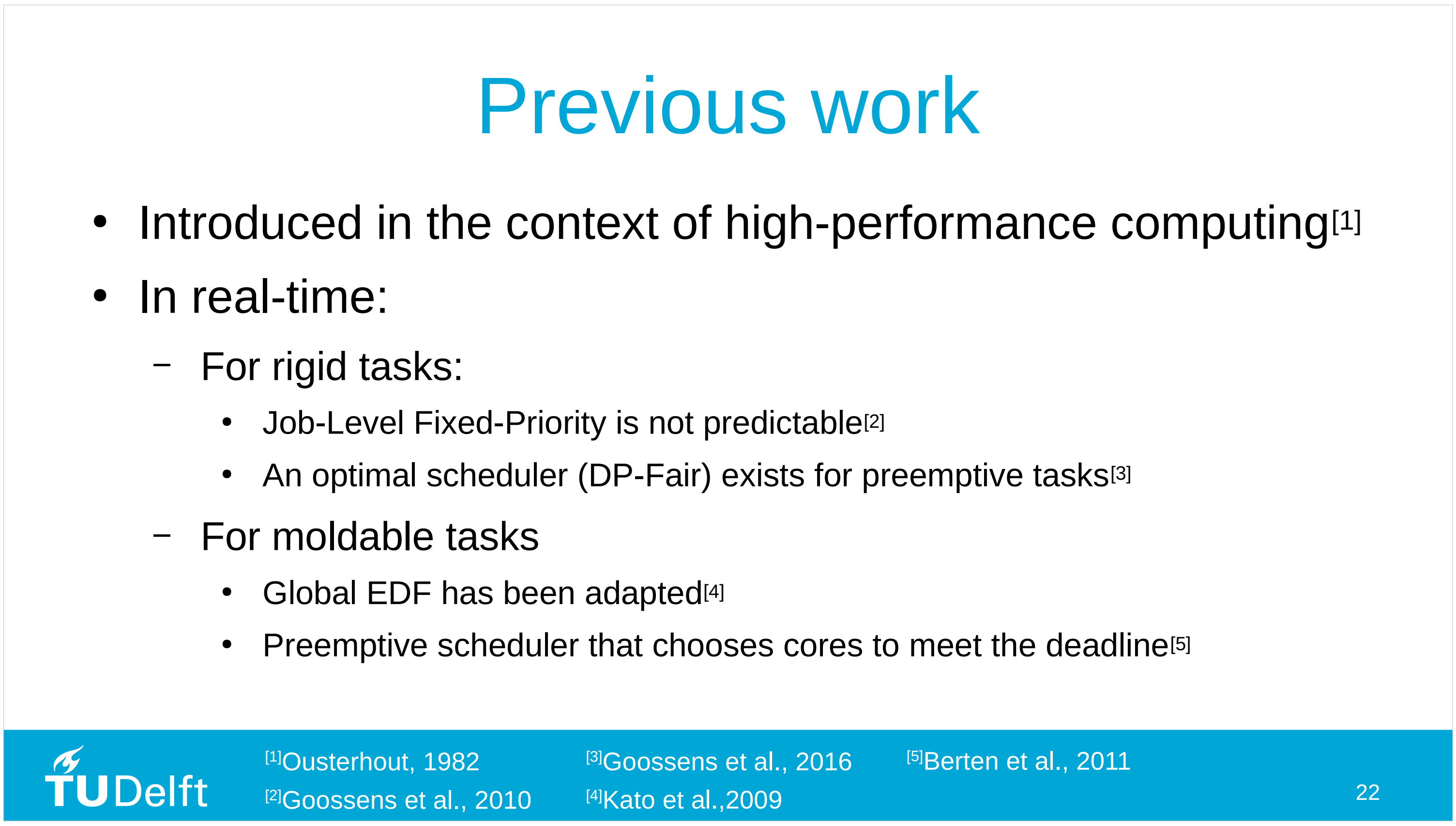
TUDelft

[1]Ousterhout, 1982
[2]Goossens et al., 2010
[3]Goossens et al., 2016
[4]Kato et al.,2009
[5]Berten et al., 2011

Alright, so what has been done previously in gang scheduling? <click>First, the gang task model was introduced in high-performance computing as a way to reduce synchronization overhead and optimize access to shared data

<click>From a real-time viewpoint the results are rather limited.

- <click>For rigid gang taks it has been proven that the Job-Level Fixed-Priority scheduler is not predictable and an optimal scheduler was proposed, although it requires a high number of preemptions.
- <click>For moldable gang tasks, the global EDF scheduler has been adapted and studied. Additionally a preemptive scheduler was proposed that tries to be smart and choose the appropiate cores in order to meet deadlines.



Previous work

- Introduced in the context of high-performance computing[1]
- In real-time:
  - For rigid tasks:
    - Job-Level Fixed-Priority is not predictable[2]
    - An optimal scheduler (DP-Fair) exists for preemptive tasks[3]
  - For moldable tasks
    - Global EDF has been adapted[4]
    - Preemptive scheduler that chooses cores to meet the deadline[5]

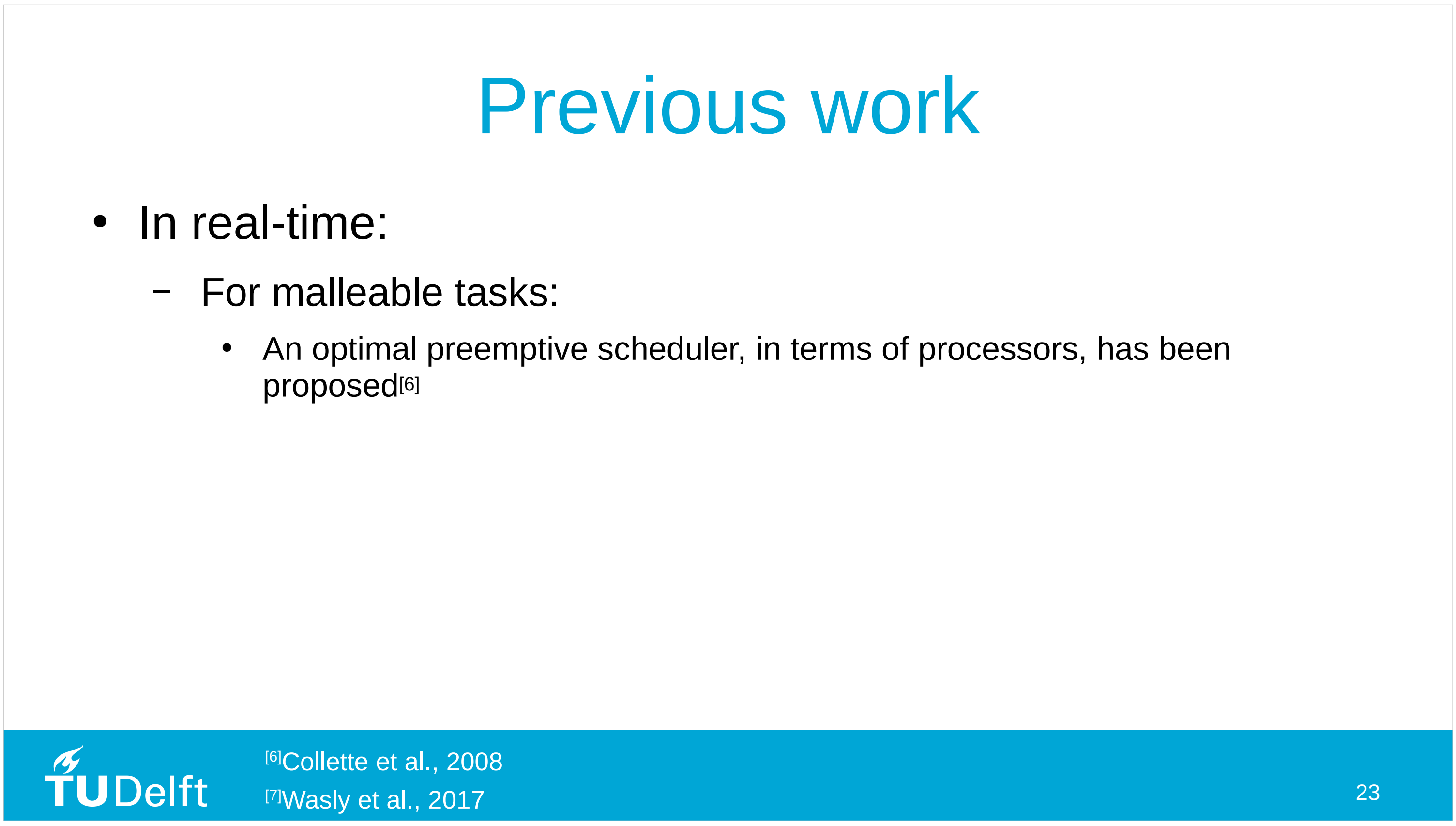
TUDelft

[1]Ousterhout, 1982
[2]Goossens et al., 2010
[3]Goossens et al., 2016
[4]Kato et al.,2009
[5]Berten et al., 2011

Alright, so what has been done previously in gang scheduling? <click>First, the gang task model was introduced in high-performance computing as a way to reduce synchronization overhead and optimize access to shared data

<click>From a real-time viewpoint the results are rather limited.

- <click>For rigid gang taks it has been proven that the Job-Level Fixed-Priority scheduler is not predictable and an optimal scheduler was proposed, although it requires a high number of preemptions.
- <click>For moldable gang tasks, the global EDF scheduler has been adapted and studied. Additionally a preemptive scheduler was proposed that tries to be smart and choose the appropiate cores in order to meet deadlines.



Previous work

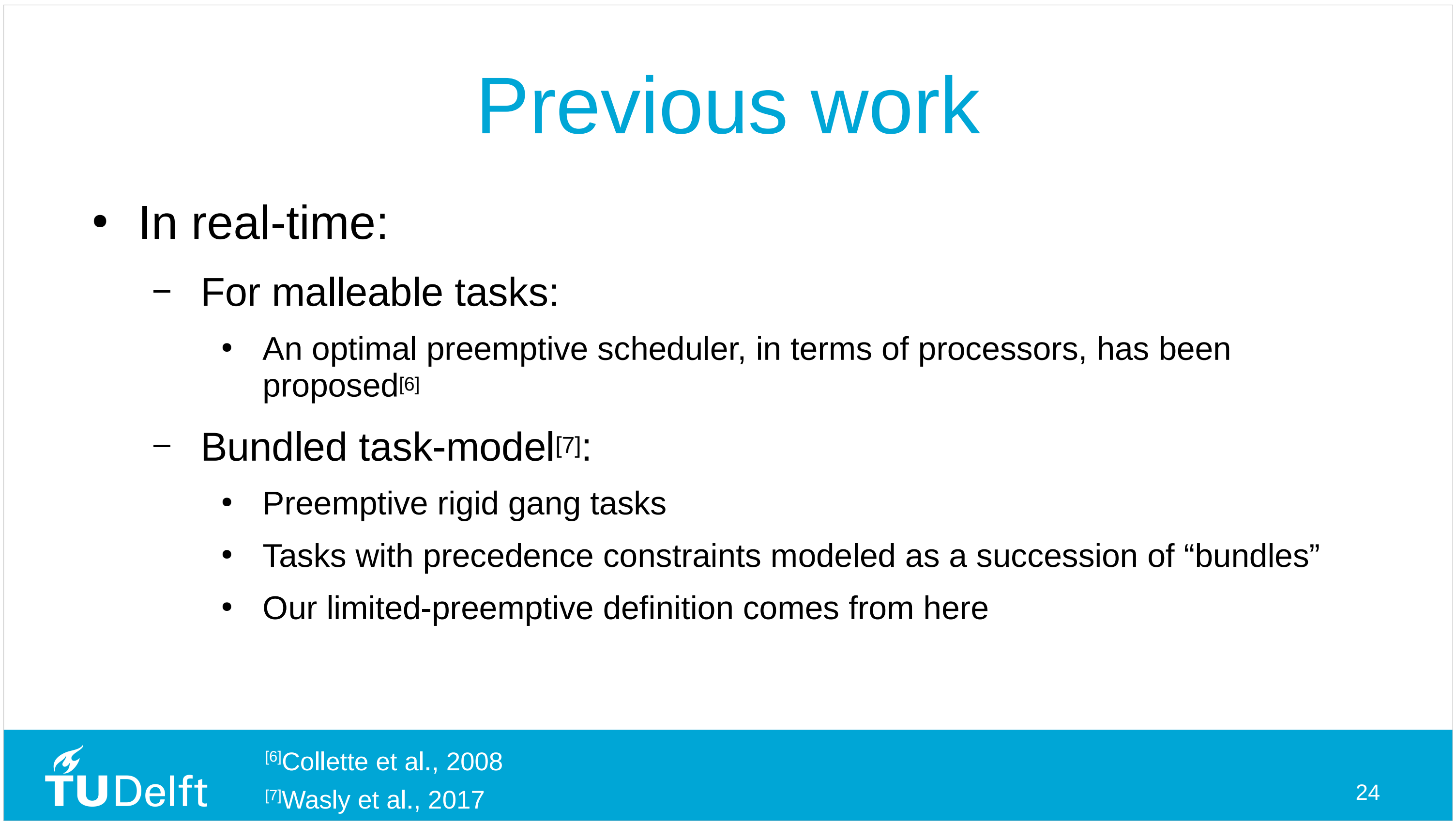
- In real-time:
  - For malleable tasks:
    - An optimal preemptive scheduler, in terms of processors, has been proposed[6]

TUDelft

[6]Collette et al., 2008

[7]Wasly et al., 2017

- For malleable taks an optimal, in terms of processors, preemptive scheduler has been proposed. It was one of the first works of gang scheduling with real-time analysis
- <click>Additionally there is the bundled task model that works with preemptive rigid gang tasks and models the tasks with precedence constraints as a succession of “bundles”. The limited-preemptive definition that we use for our work comes from this definition.



Previous work

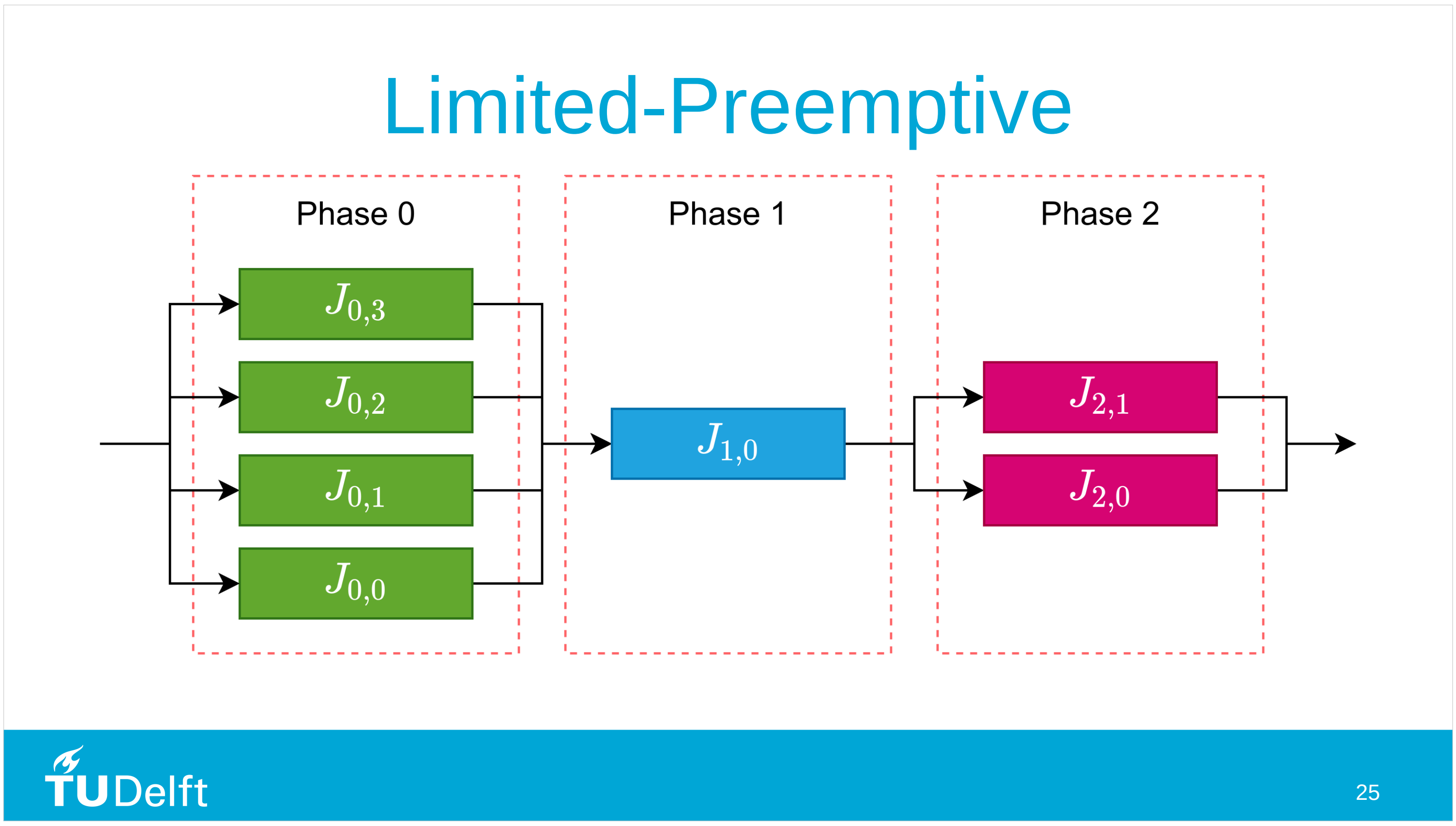
- In real-time:
  - For malleable tasks:
    - An optimal preemptive scheduler, in terms of processors, has been proposed[6]
  - Bundled task-model[7]:
    - Preemptive rigid gang tasks
    - Tasks with precedence constraints modeled as a succession of “bundles”
    - Our limited-preemptive definition comes from here

TU Delft

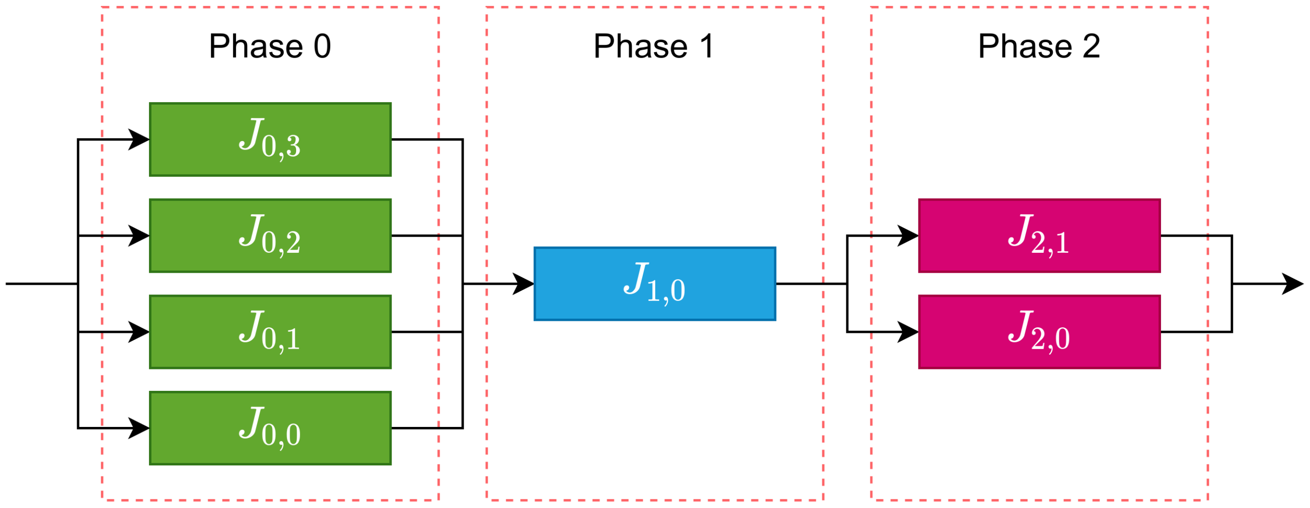
[6]Collette et al., 2008

[7]Wasly et al., 2017

- For malleable taks an optimal, in terms of processors, preemptive scheduler has been proposed. It was one of the first works of gang scheduling with real-time analysis
- <click>Additionally there is the bundled task model that works with preemptive rigid gang tasks and models the tasks with precedence constraints as a succession of “bundles”. The limited-preemptive definition that we use for our work comes from this definition.



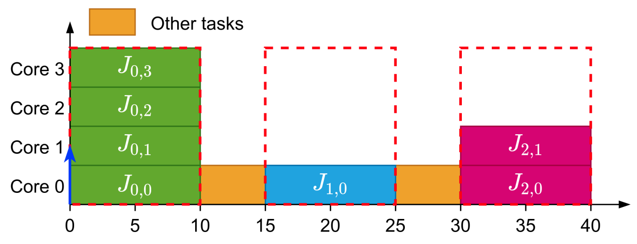
Limited-Preemptive



Let's see an example to understand the limited-preemptive definition. Here we have a job that is divided in three phases with dependencies between them.

Limited-Preemptive

- Rigid gang could ask for cores that does not use



TU Delft

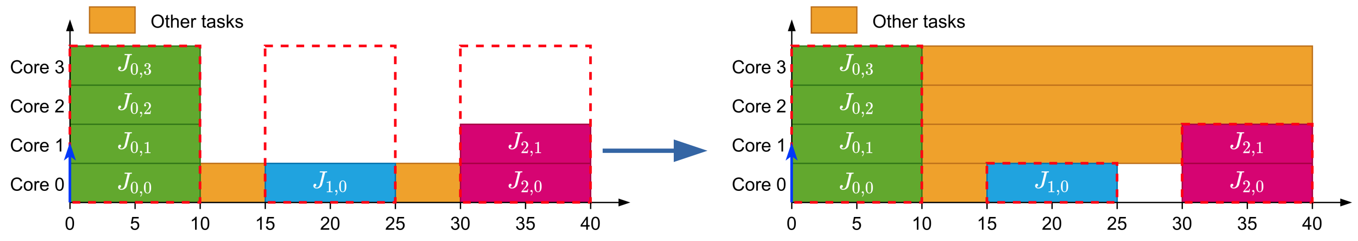
If we try to schedule them with rigid gang we could obtain the following schedule were the gang task is taking cores that does not use.<click>

That's why we have the bundled task model where each bundle only asks for the cores it requires but it allows preemptions to happen inside the bundle

<click>And here is where our limited-preemptive definition comes in where the preemptions are only allowed between bundles and not inside them.

Limited-Preemptive

- Rigid gang could ask for cores that does not use
- Bundled[7] asks only the required cores but preemptions can happen inside a bundle



[7]Wasly et al., 2017

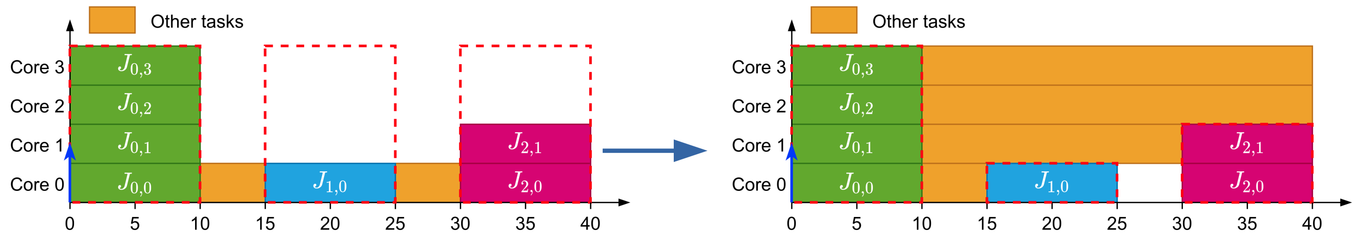
If we try to schedule them with rigid gang we could obtain the following schedule were the gang task is taking cores that does not use.<click>

That’s why we have the bundled task model where each bundle only asks for the cores it requires but it allows preemptions to happen inside the bundle

<click>And here is where our limited-preemptive definition comes in where the preemptions are only allowed between bundles and not inside them.

Limited-Preemptive

- Rigid gang could ask for cores that does not use
- Bundled[7] asks only the required cores but preemptions can happen inside a bundle
- LP only allows preemptions between bundles

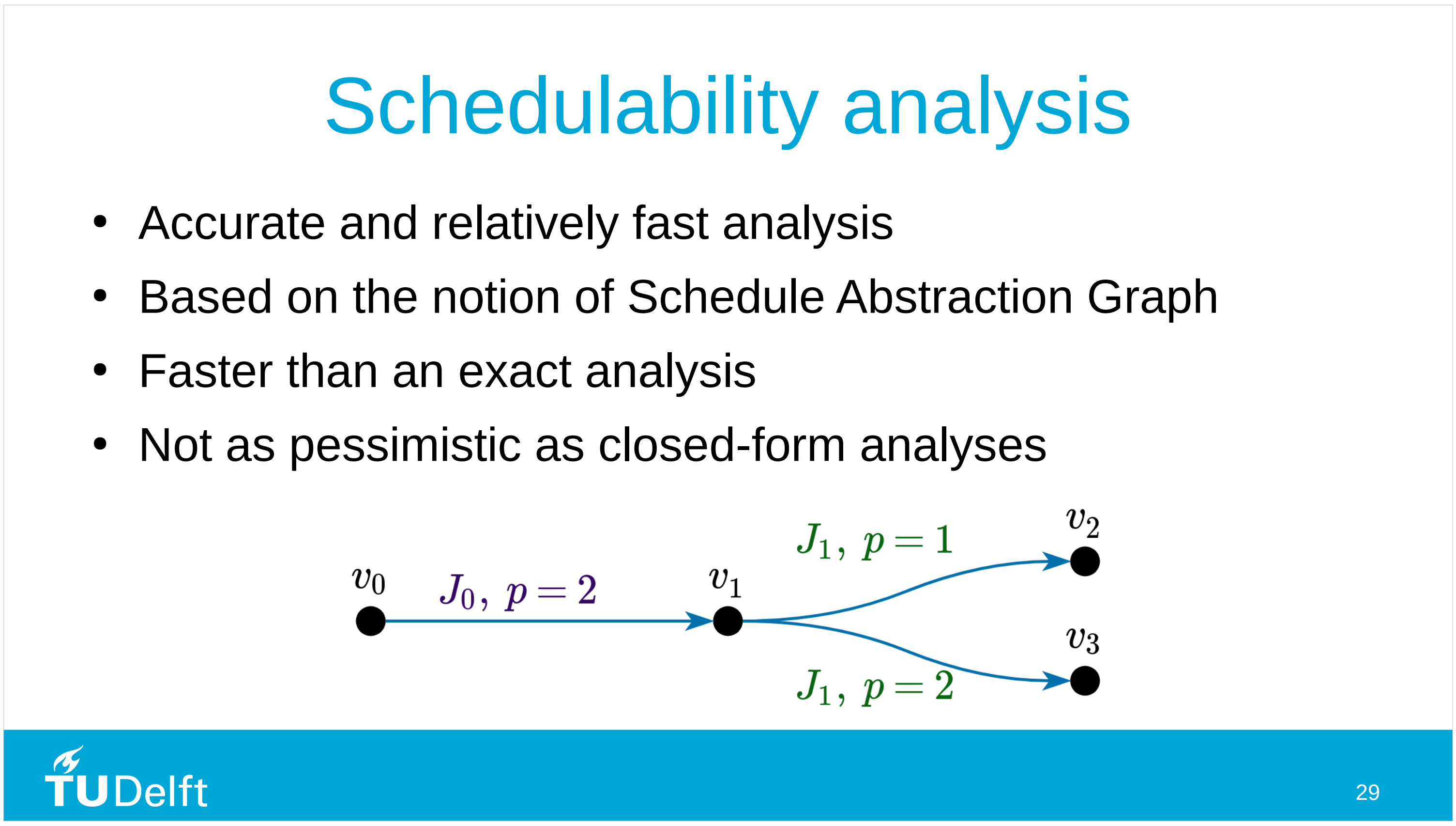


[7]Wasly et al., 2017

If we try to schedule them with rigid gang we could obtain the following schedule were the gang task is taking cores that does not use.<click>

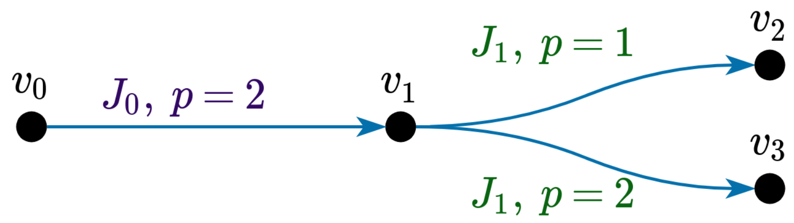
That’s why we have the bundled task model where each bundle only asks for the cores it requires but it allows preemptions to happen inside the bundle

<click>And here is where our limited-preemptive definition comes in where the preemptions are only allowed between bundles and not inside them.



Schedulability analysis

- Accurate and relatively fast analysis
- Based on the notion of Schedule Abstraction Graph
- Faster than an exact analysis
- Not as pessimistic as closed-form analyses

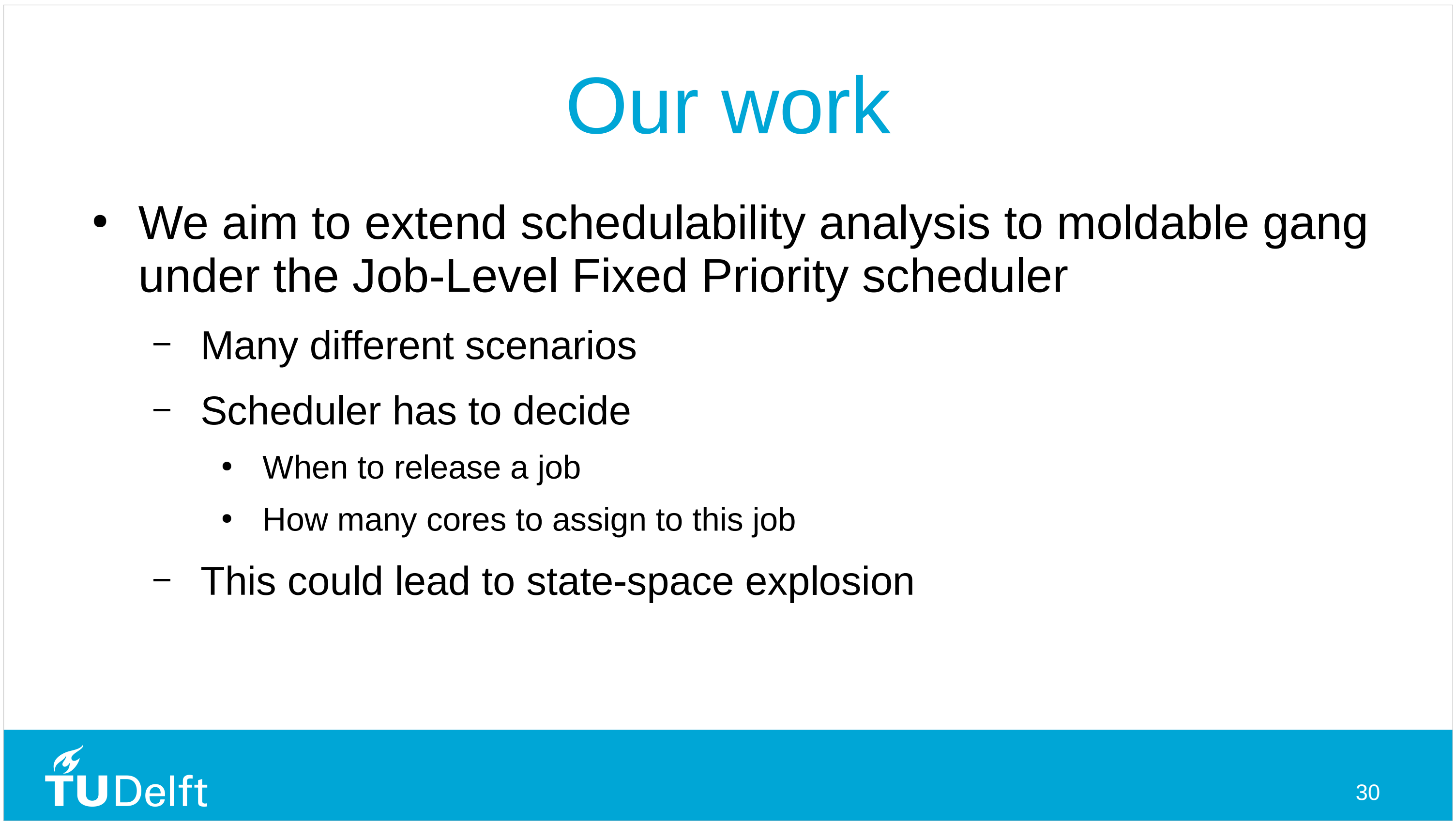


TUDelft

Now, to analyze gang tasks we use schedulability analysis. It is a technique that allows for accurate and relatively fast analysis as it is based o the notion of the Schedule Abstraction Graph.

It is faster than an exact analysis and not as pessimistic as closed-form analyses.

Here we can see the scheduling decisions that a scheduler has taken



Our work

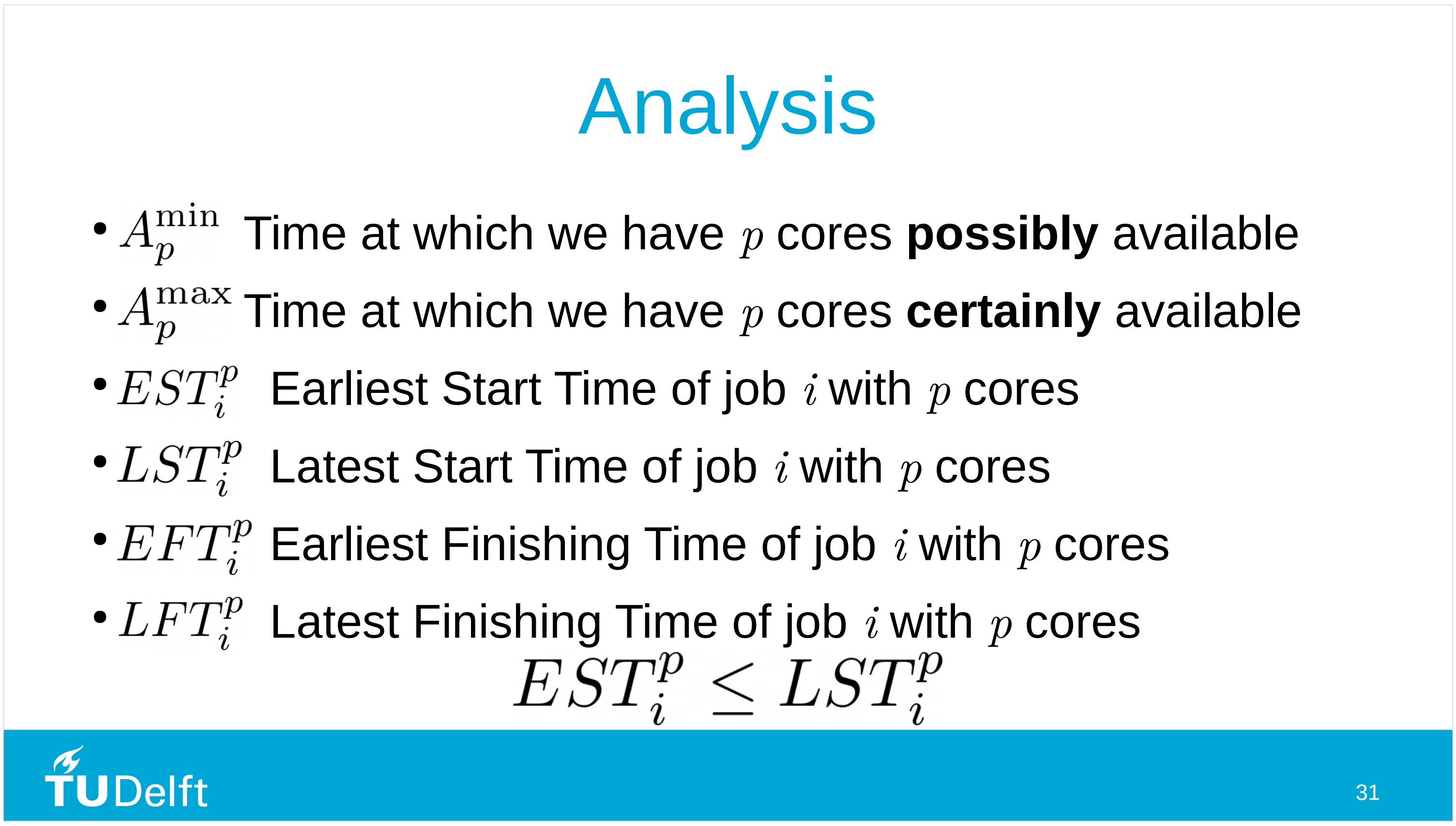
- We aim to extend schedulability analysis to moldable gang under the Job-Level Fixed Priority scheduler
  - Many different scenarios
  - Scheduler has to decide
    - When to release a job
    - How many cores to assign to this job
  - This could lead to state-space explosion

TUDelft

In our work we aim to extend the schedulability anylsis to incorporate the limited-preemptive moldable gang tasks.

However it has some difficulties as there are many different scenarios. In this case the scheduler has to decide when to release a job and how many cores it asssigns to the job.

This could lead to state-space explosion



Analysis

- $A_p^{\min}$ Time at which we have p cores **possibly** available
- $A_p^{\max}$ Time at which we have p cores **certainly** available
- EST_i^p Earliest Start Time of job i with p cores
- LST_i^p Latest Start Time of job i with p cores
- EFT_i^p Earliest Finishing Time of job i with p cores
- LFT_i^p Latest Finishing Time of job i with p cores

$$EST_i^p \leq LST_i^p$$

TUDelft

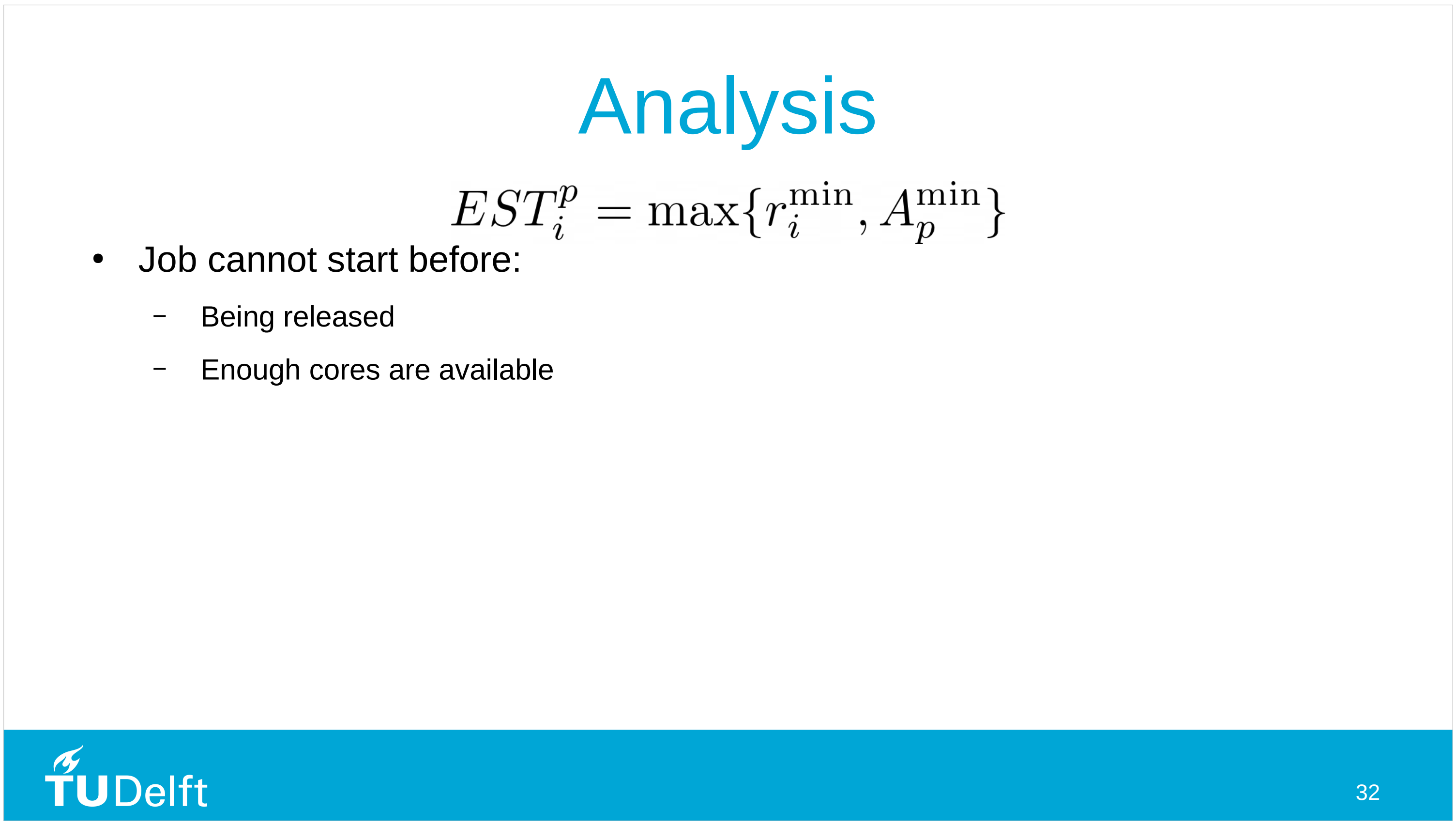
To do the analysis under the fixed priority scheduler we need at least the following terms.

A_min and A_max are the times at which we know that there are possibly for A_min and certainly for A_max p cores. This is the starting point to perform the analysis.

EST and LST are the earliest and latest start time of a job executed with p cores

EFT and LFT are the earliest and latest finishing time of a job executed with p cores

In order for a job to be eligible to be scheduled its earliest start time has to be lower or equal than the latest start time.



Analysis

$$EST_i^p = \max\{r_i^{\min}, A_p^{\min}\}$$

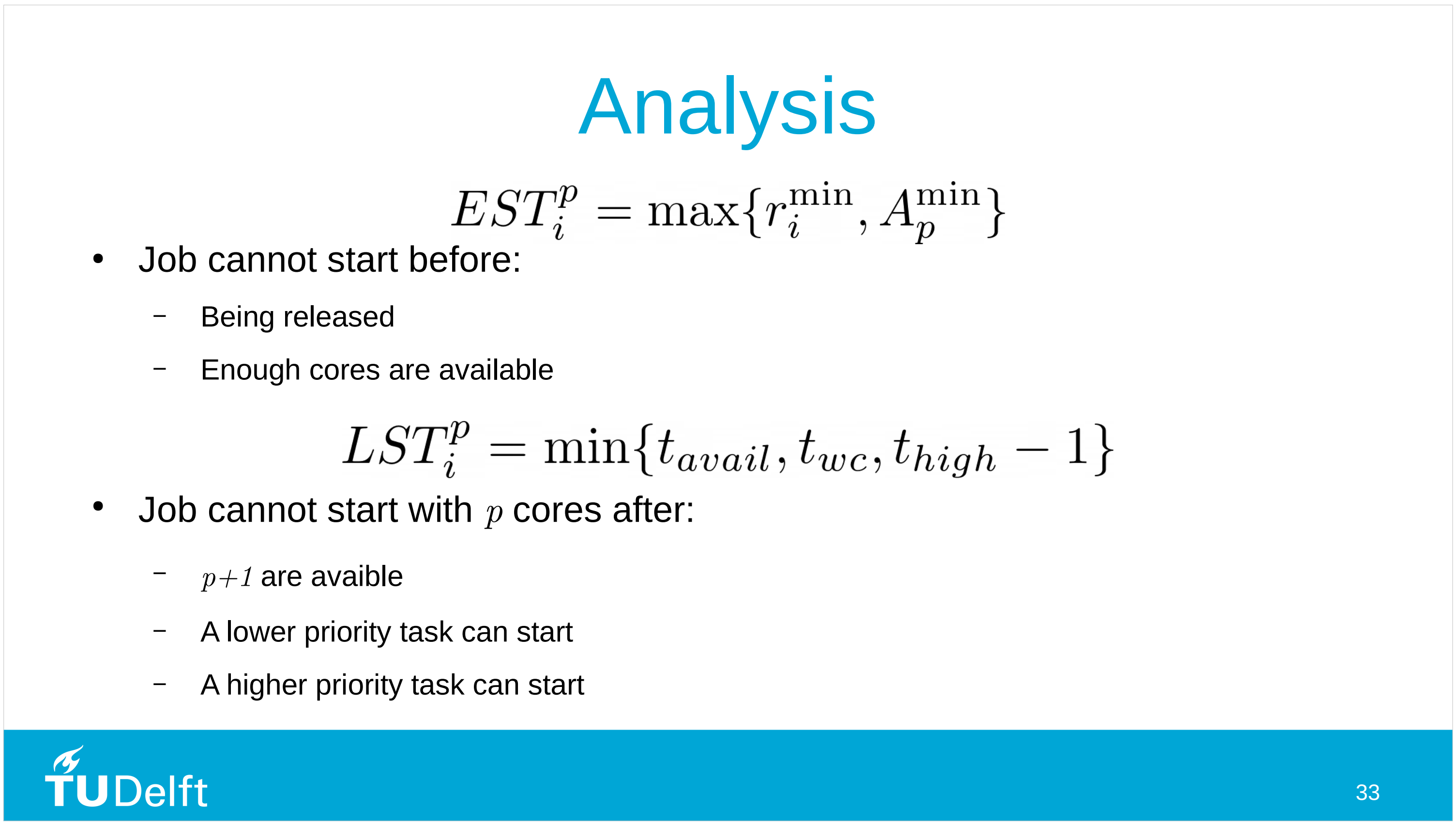
- Job cannot start before:
  - Being released
  - Enough cores are available

TU Delft

To get to the next step we compute the EST and LST of a job.

For the EST we know that a job cannot start before being released and if there are not enough cores to execute the job

<click>For the LST we know that if we want to execute it with p cores with the fixed-priority scheduler there cannot be p+1 cores as otherwise the job would execute with p+1 cores (unless the maximum number of cores is p). Additionally if this job is to be executed it has to be released before or at the same time of a lower priority task as otherwise the lower priority task would be executed. Finally the job has to be released before a higher priority task in order to be scheduled.



Analysis

$$EST_i^p = \max\{r_i^{\min}, A_p^{\min}\}$$

- Job cannot start before:
  - Being released
  - Enough cores are available

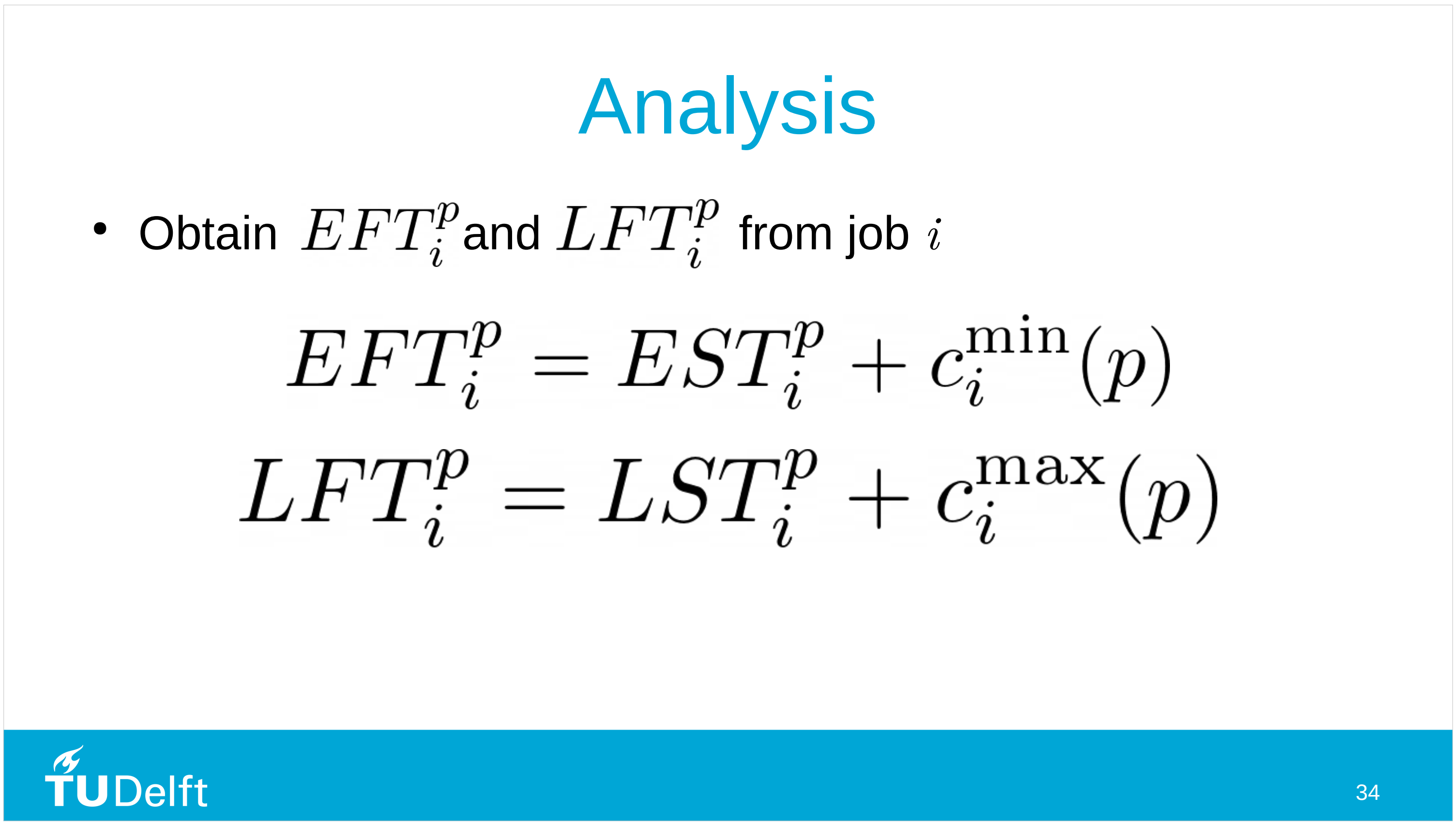
$$LST_i^p = \min\{t_{avail}, t_{wc}, t_{high} - 1\}$$

- Job cannot start with p cores after:
  - $p+1$ are avaible
  - A lower priority task can start
  - A higher priority task can start

To get to the next step we compute the EST and LST of a job.

For the EST we know that a job cannot start before being released and if there are not enough cores to execute the job

<click>For the LST we know that if we want to execute it with p cores with the fixed-priority scheduler there cannot be p+1 cores as otherwise the job would execute with p+1 cores (unless the maximum number of cores is p). Additionally if this job is to be executed it has to be released before or at the same time of a lower priority task as otherwise the lower priority task would be executed. Finally the job has to be released before a higher priority task in order to be scheduled.



Analysis

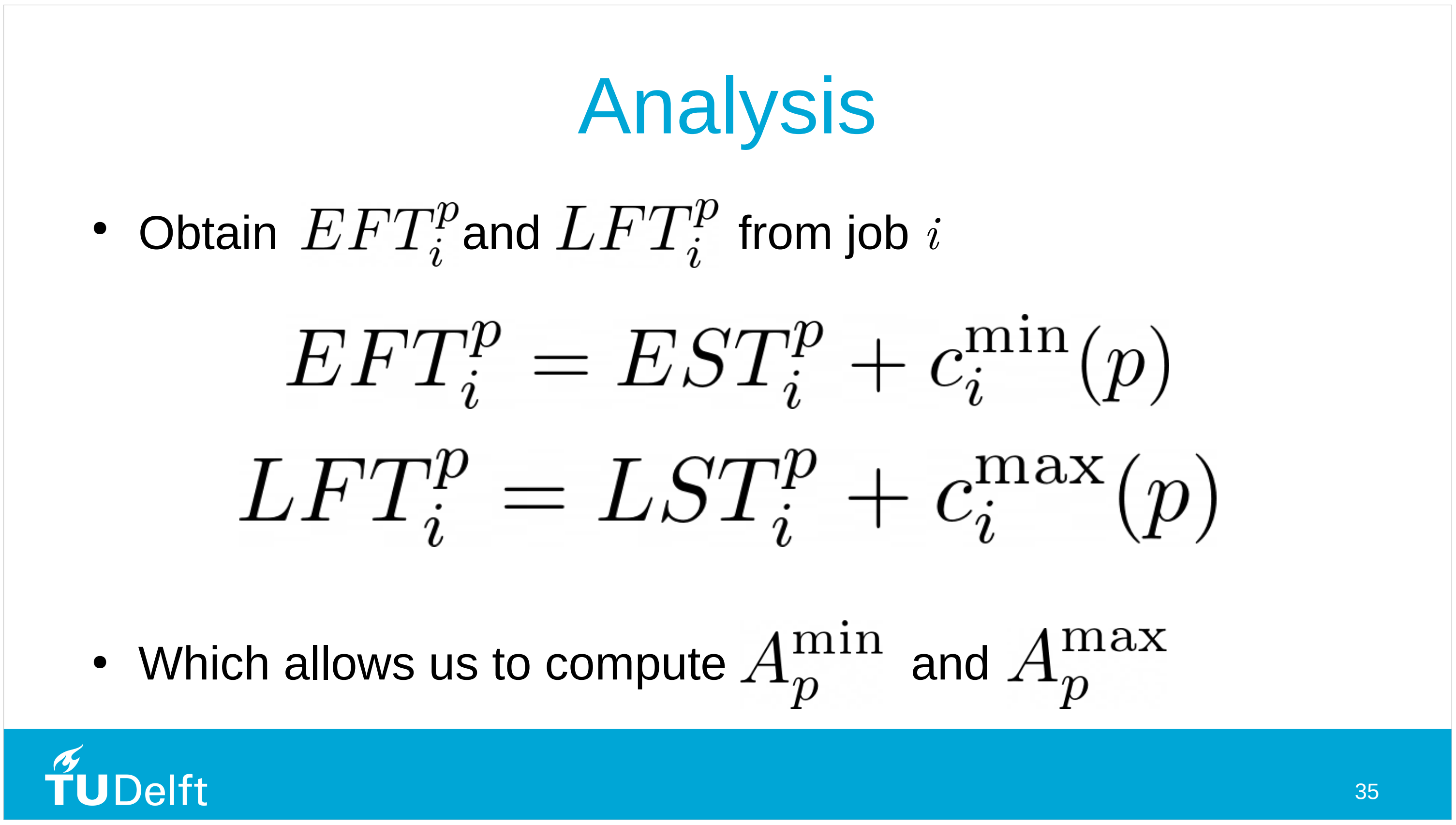
- Obtain EFT_i^p and LFT_i^p from job i

$$EFT_i^p = EST_i^p + c_i^{\min}(p)$$
$$LFT_i^p = LST_i^p + c_i^{\max}(p)$$

TUDelft

Once we have the EST and the LST we can compute the earliest finishing time and latest finishing time just using the minimum and maximum execution time of the job.

Finally we can obtain new availability times for the cores and from here compute the state for other jobs. If there’s a deadline miss we would see it in this step



Analysis

- Obtain EFT_i^p and LFT_i^p from job i

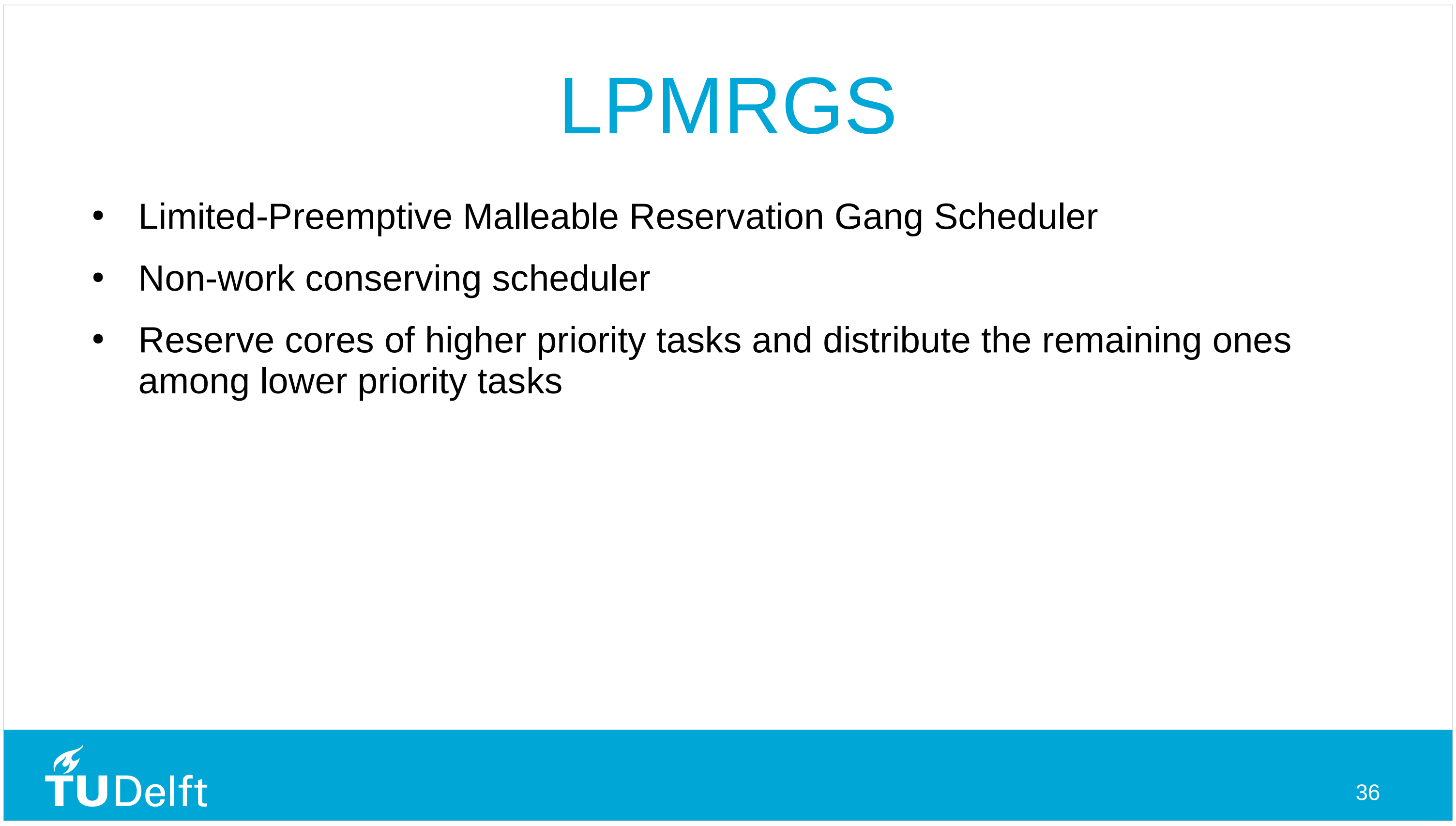
$$EFT_i^p = EST_i^p + c_i^{\min}(p)$$
$$LFT_i^p = LST_i^p + c_i^{\max}(p)$$

- Which allows us to compute $A_p^{\min}$ and $A_p^{\max}$

TUDelft

Once we have the EST and the LST we can compute the earliest finishing time and latest finishing time just using the minimum and maximum execution time of the job.

Finally we can obtain new availability times for the cores and from here compute the state for other jobs. If there’s a deadline miss we would see it in this step



LPMRGS

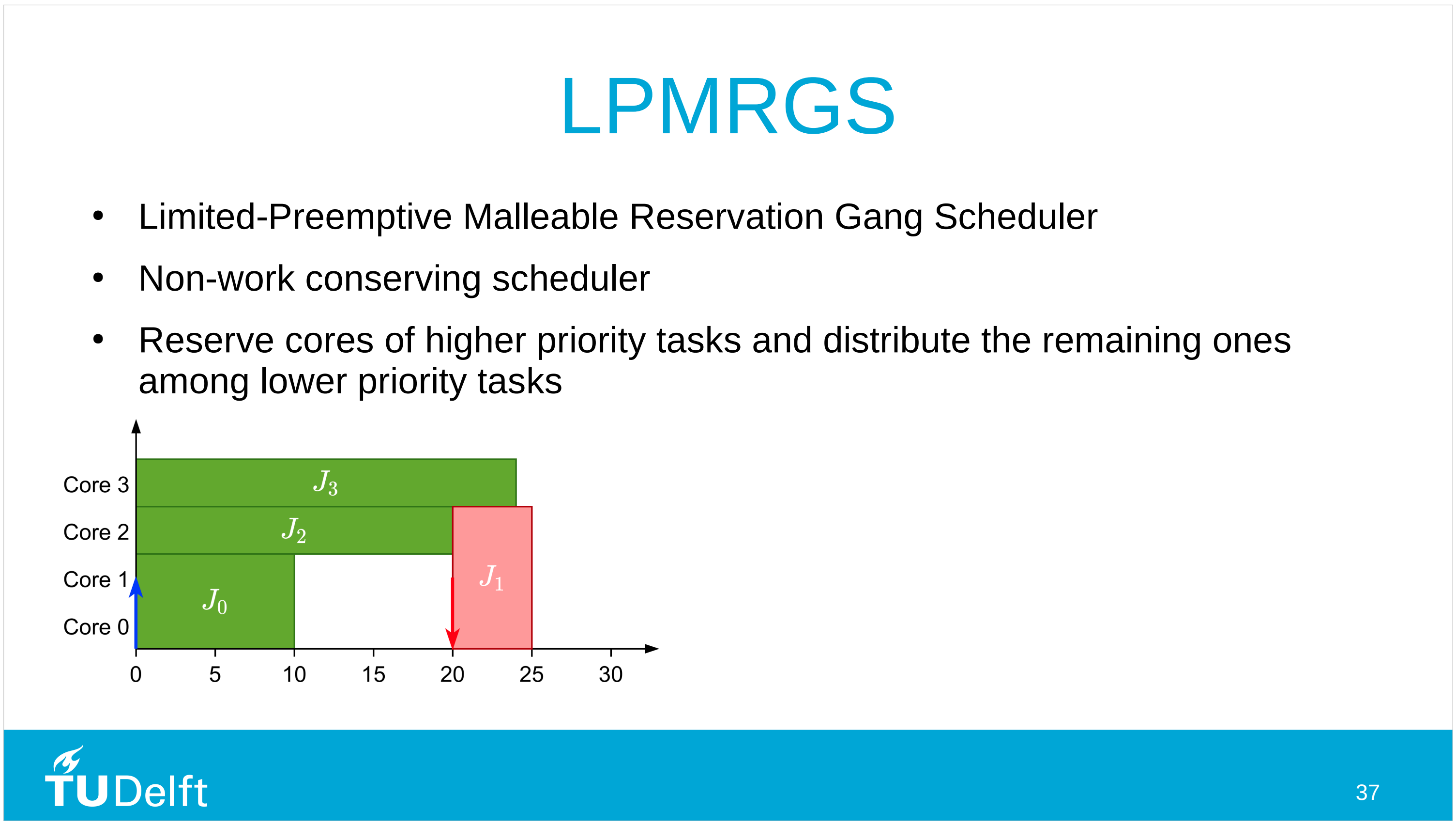
- Limited-Preemptive Malleable Reservation Gang Scheduler
- Non-work conserving scheduler
- Reserve cores of higher priority tasks and distribute the remaining ones among lower priority tasks

TUDelft

Additionally we are working in a Limited-Preemptive Malleable Reservation Gang Scheduler which is a non-work conserving scheduler.

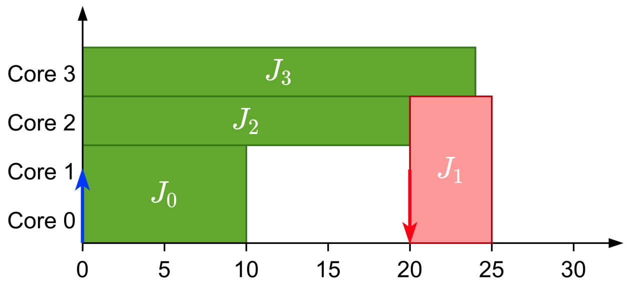
The idea behind is to reserve some cores for higher priority jobs and leave them idle.<click>

Let’s look at this example where the priorities are indicated by the job id and 0 is the highest priority. At time 0 Job 0 has been scheduled leaving 2 cores free but since Job 1 required 3 cores then it has not been scheduled an jobs 2 and 3 have been scheduled instead. Since we know that job 0 has to wait for job 1 to finish we could just leave one core idle reserved for job 1<click> which would allow job 1 to be scheduled immediately after job 0 finishes its execution



LPMRGS

- Limited-Preemptive Malleable Reservation Gang Scheduler
- Non-work conserving scheduler
- Reserve cores of higher priority tasks and distribute the remaining ones among lower priority tasks



TUDelft 37

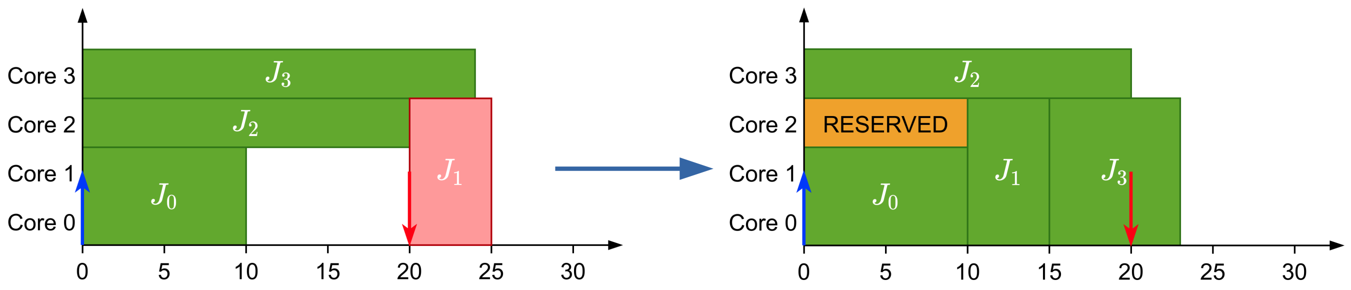
Additionally we are working in a Limited-Preemptive Malleable Reservation Gang Scheduler which is a non-work conserving scheduler.

The idea behind is to reserve some cores for higher priority jobs and leave them idle.<click>

Let’s look at this example where the priorities are indicated by the job id and 0 is the highest priority. At time 0 Job 0 has been scheduled leaving 2 cores free but since Job 1 required 3 cores then it has not been scheduled an jobs 2 and 3 have been scheduled instead. Since we know that job 0 has to wait for job 1 to finish we could just leave one core idle reserved for job 1<click> which would allow job 1 to be scheduled immediately after job 0 finishes its execution

LPMRGS

- Limited-Preemptive Malleable Reservation Gang Scheduler
- Non-work conserving scheduler
- Reserve cores of higher priority tasks and distribute the remaining ones among lower priority tasks



Additionally we are working in a Limited-Preemptive Malleable Reservation Gang Scheduler which is a non-work conserving scheduler.

The idea behind is to reserve some cores for higher priority jobs and leave them idle.<click>

Let’s look at this example where the priorities are indicated by the job id and 0 is the highest priority. At time 0 Job 0 has been scheduled leaving 2 cores free but since Job 1 required 3 cores then it has not been scheduled an jobs 2 and 3 have been scheduled instead. Since we know that job 0 has to wait for job 1 to finish we could just leave one core idle reserved for job 1<click> which would allow job 1 to be scheduled immediately after job 0 finishes its execution

Questions?

TUDelft