Task Parallelism

Prof. Guido Araujo

Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Programming models

- BSC OpenSs
- OpenMP 4.x
  - Intel IOMP
  - GCC GOMP
  - Unicamp MTSP

```
#pragma omp task in(v[i-1]) out(v[i])
fun1(&v[i-1], &v[i])
```

Task Parallelism

The OpenMP 4.0 task programming model

```
for (int i=1, j=1;  i<N;  i++)  {

      #pragma omp task in(v[i-1]) out(v[i])
       fun1(&v[i-1],  &v[i]);

       for (int k=0;  k<i;  k++,  j++)  {

          #pragma omp task in(v[i]) out(u[j])
             fun2(&v[i],   &u[j]);
       }

       fun3( 3 * i );
}
```

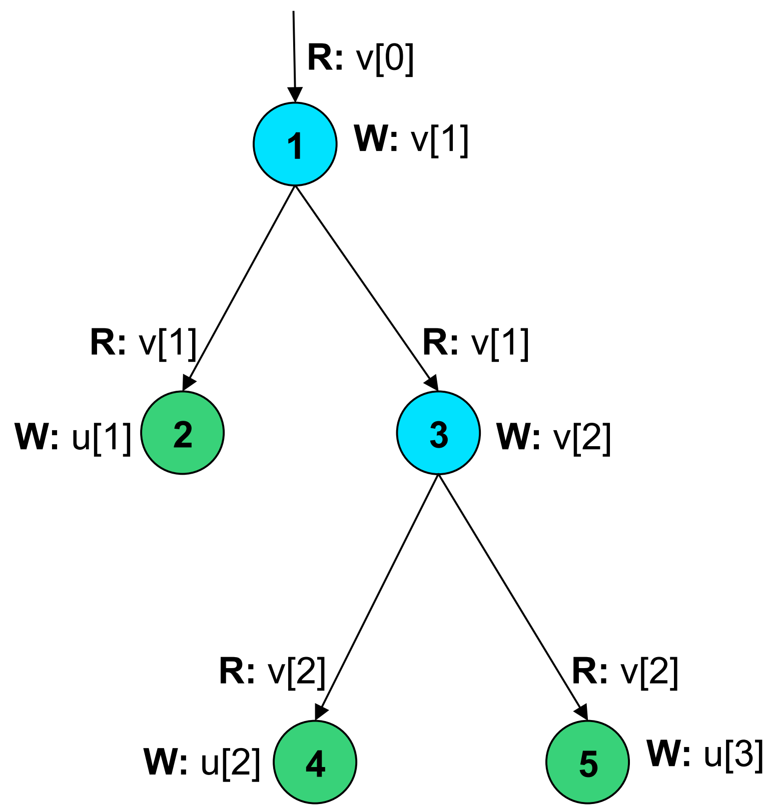
Create a task!

Create a task!

Create NO tasks!

Task Parallelism

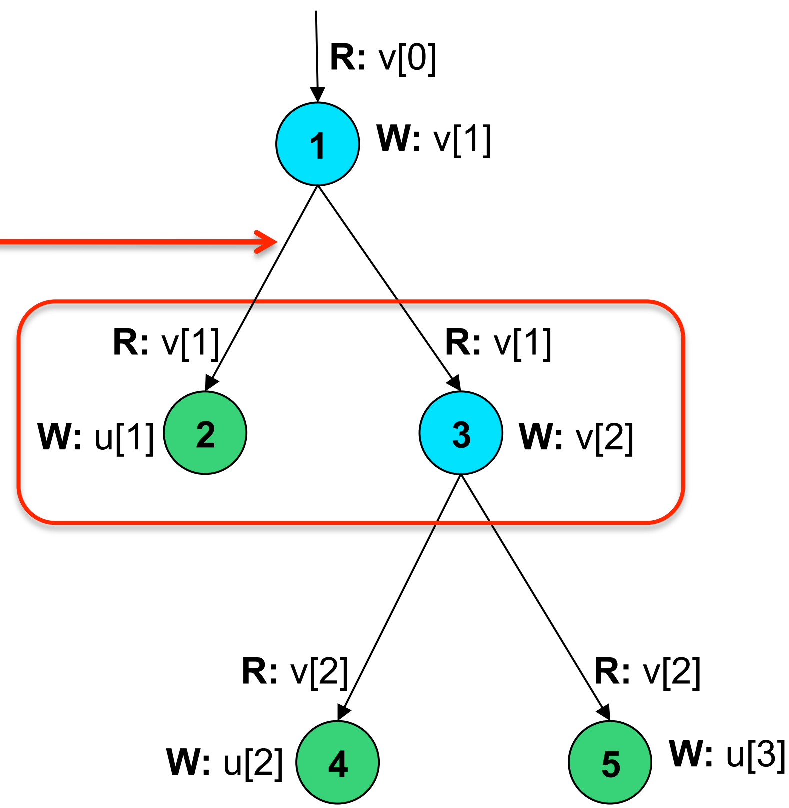
```
for (int i=1, j=1;  i<N;  i++)  {

    #pragma task in(v) out(v)
     fun1(&v[i-1],  &v[i]);

     for (int k=0;  k<i;  k++,  j++)  {

       #pragma task in(v) out(u)
       fun2(&v[i],   &u[j]);
     }

     fun3( 3 * i );
}
```



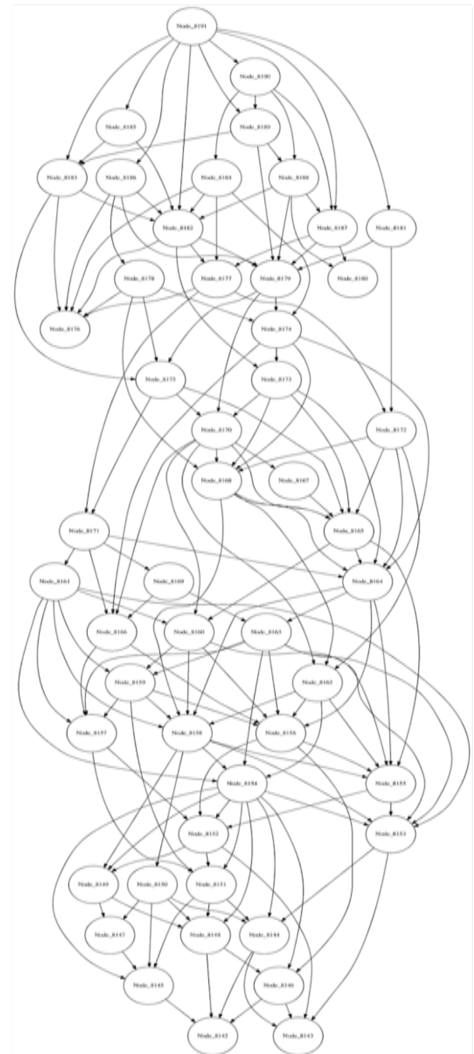
Task Parallelism Advantages

No need to communicate data. Just detect the dependency

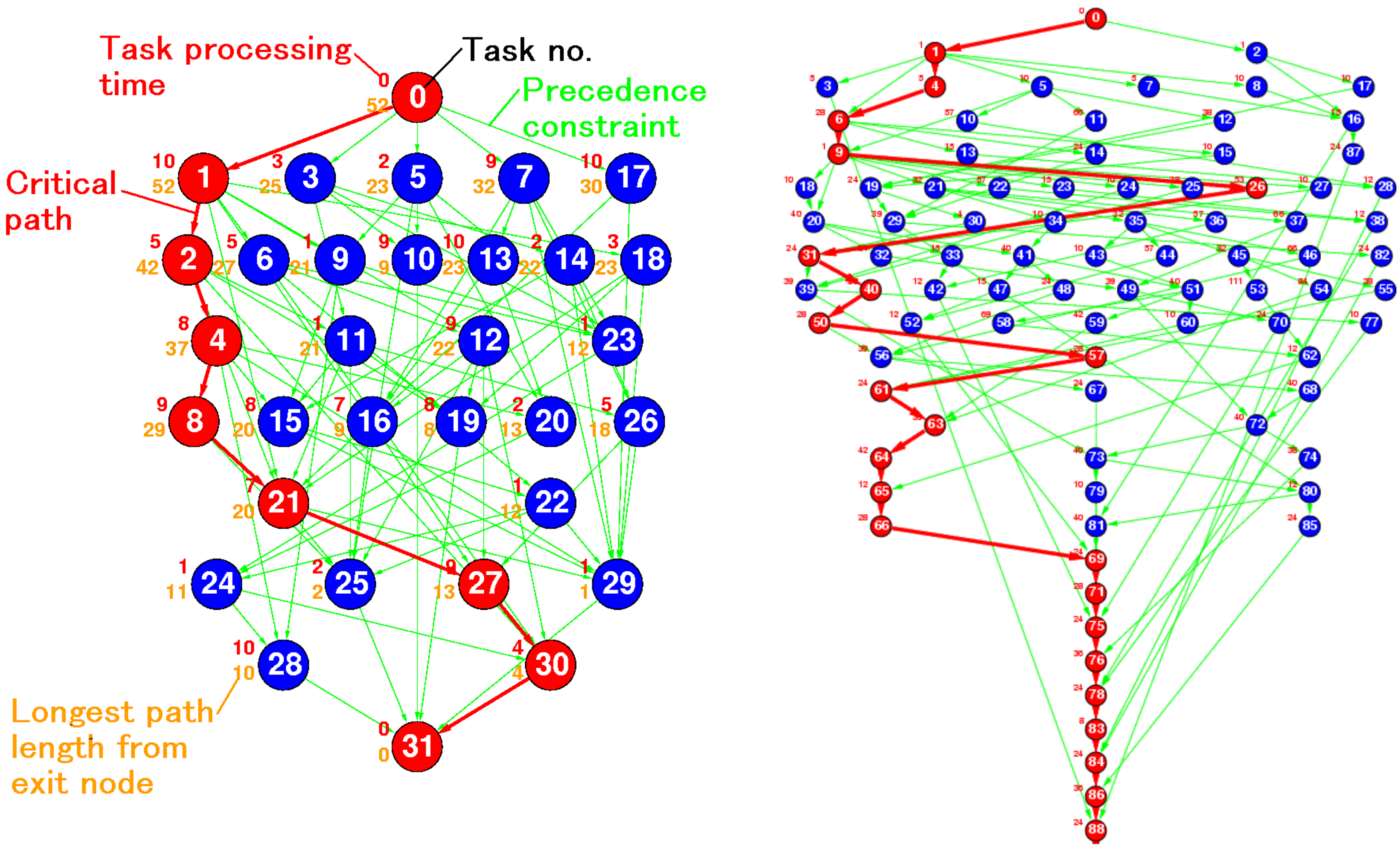
Parallelism between interations of different loops



But graphs are not always that simple!



But graphs are not always that simple!

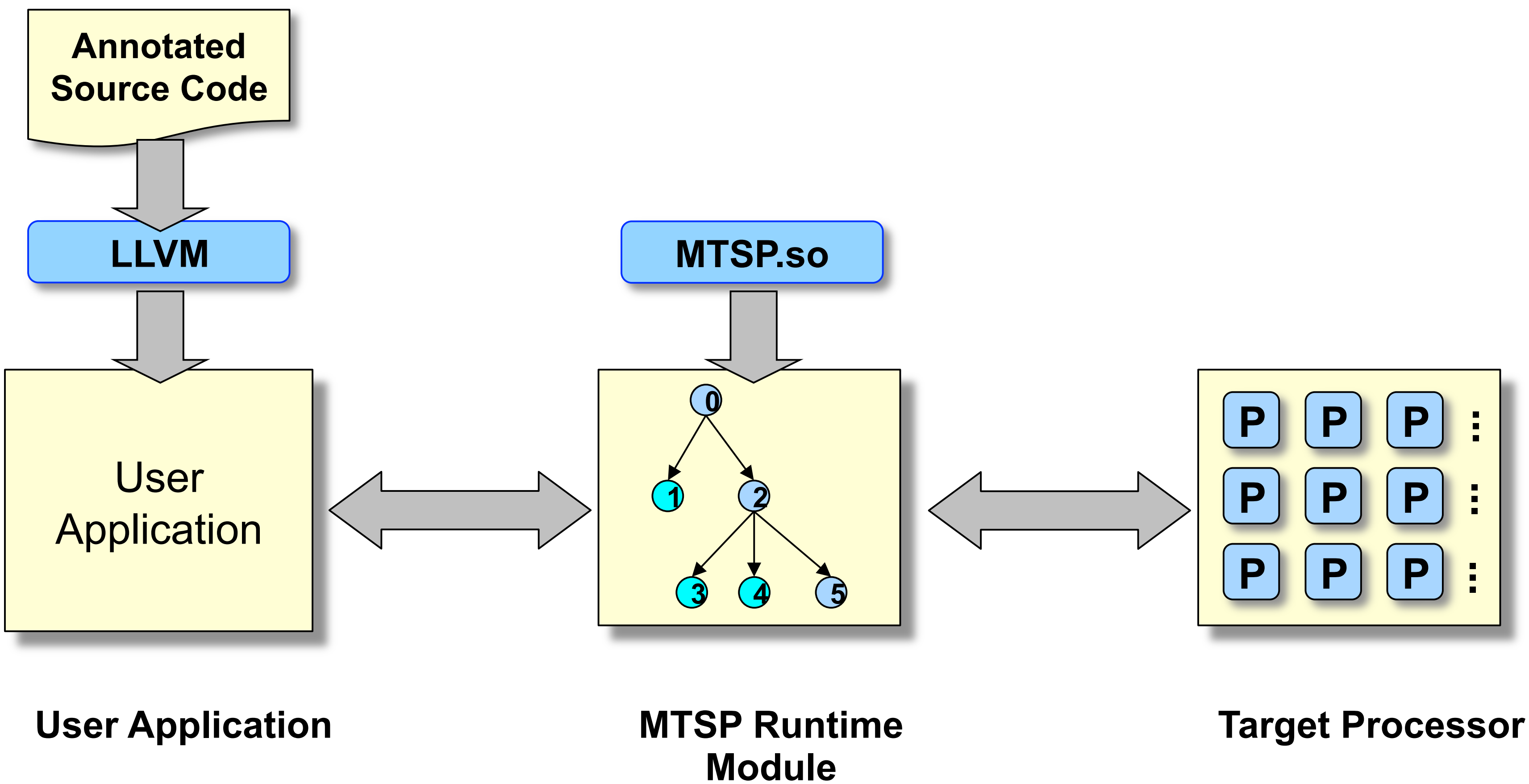


Task Parallelism

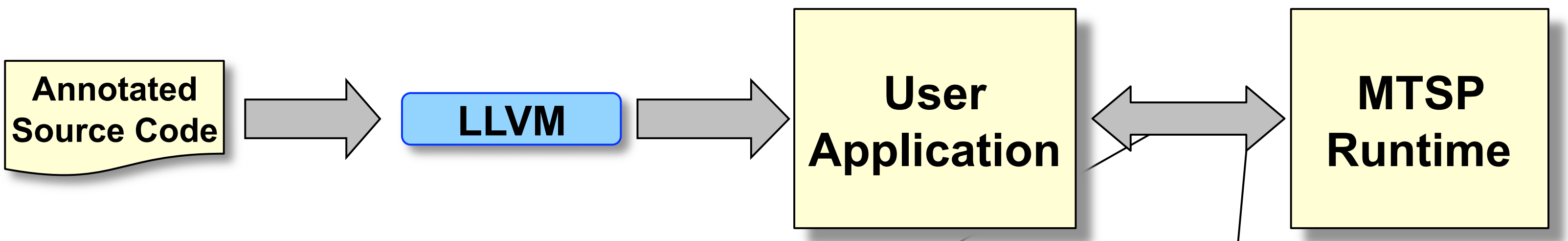
- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Runtime design

The MSTP runtime



The CLang OMP Interface



/// Used to create a pool of threads

/// and initiate the system

void kmpc_fork_call(...);

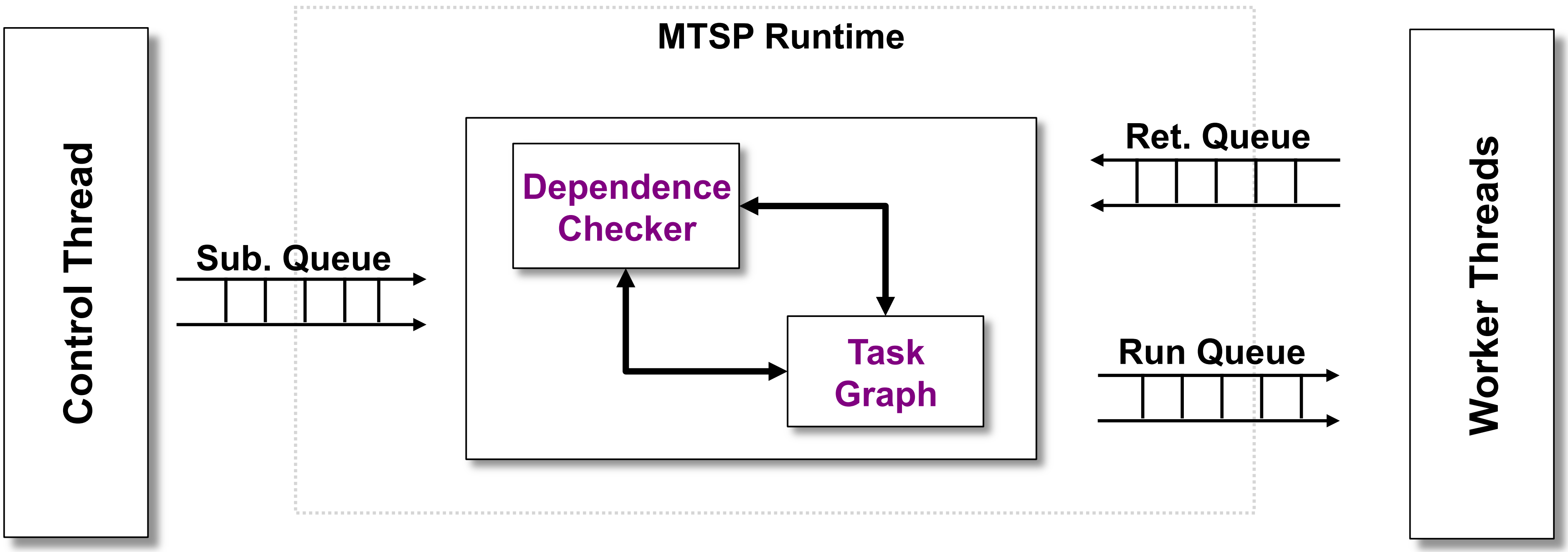
/// Used for task allocation

kmp_task* **kmpc_omp_task_alloc**(...);

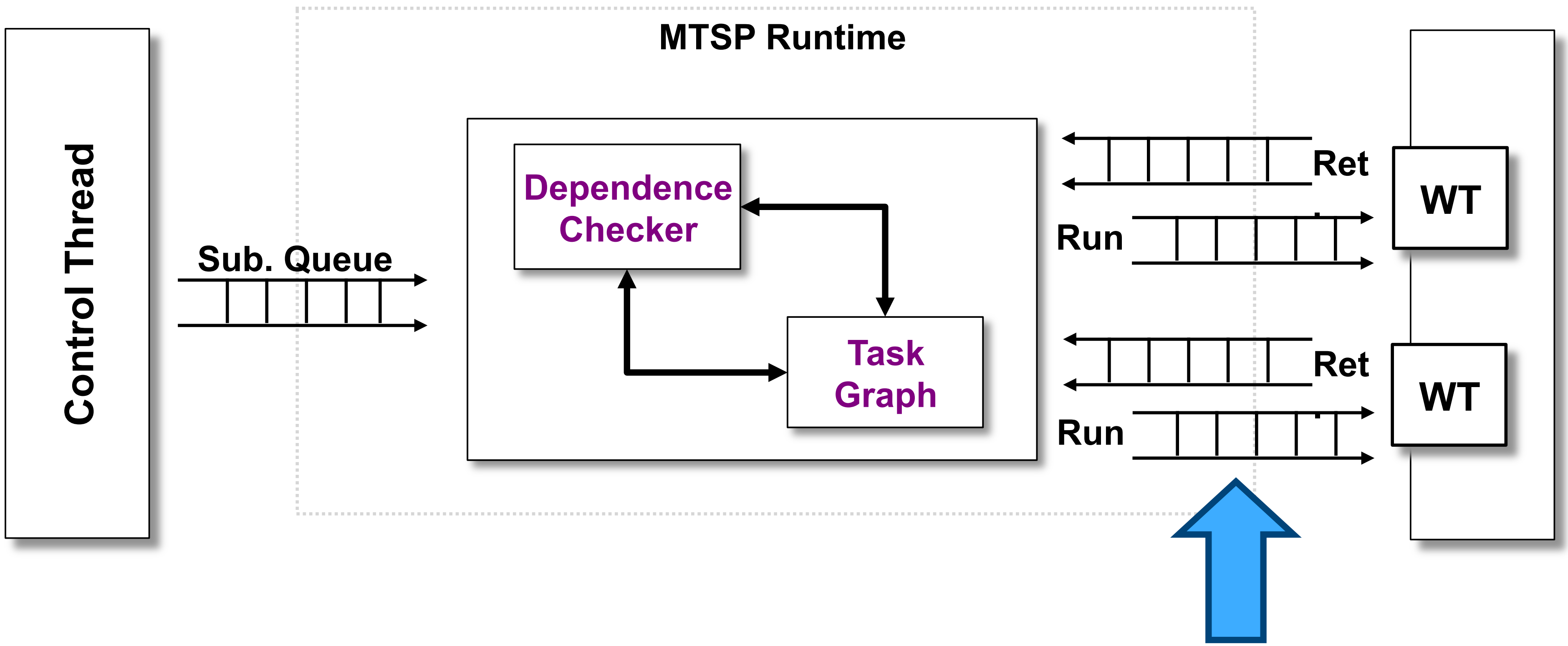
/// Used to add a task to the runtime

***kmp_int32* kmpc_omp_task_with_deps(...);**

MTSP Overview



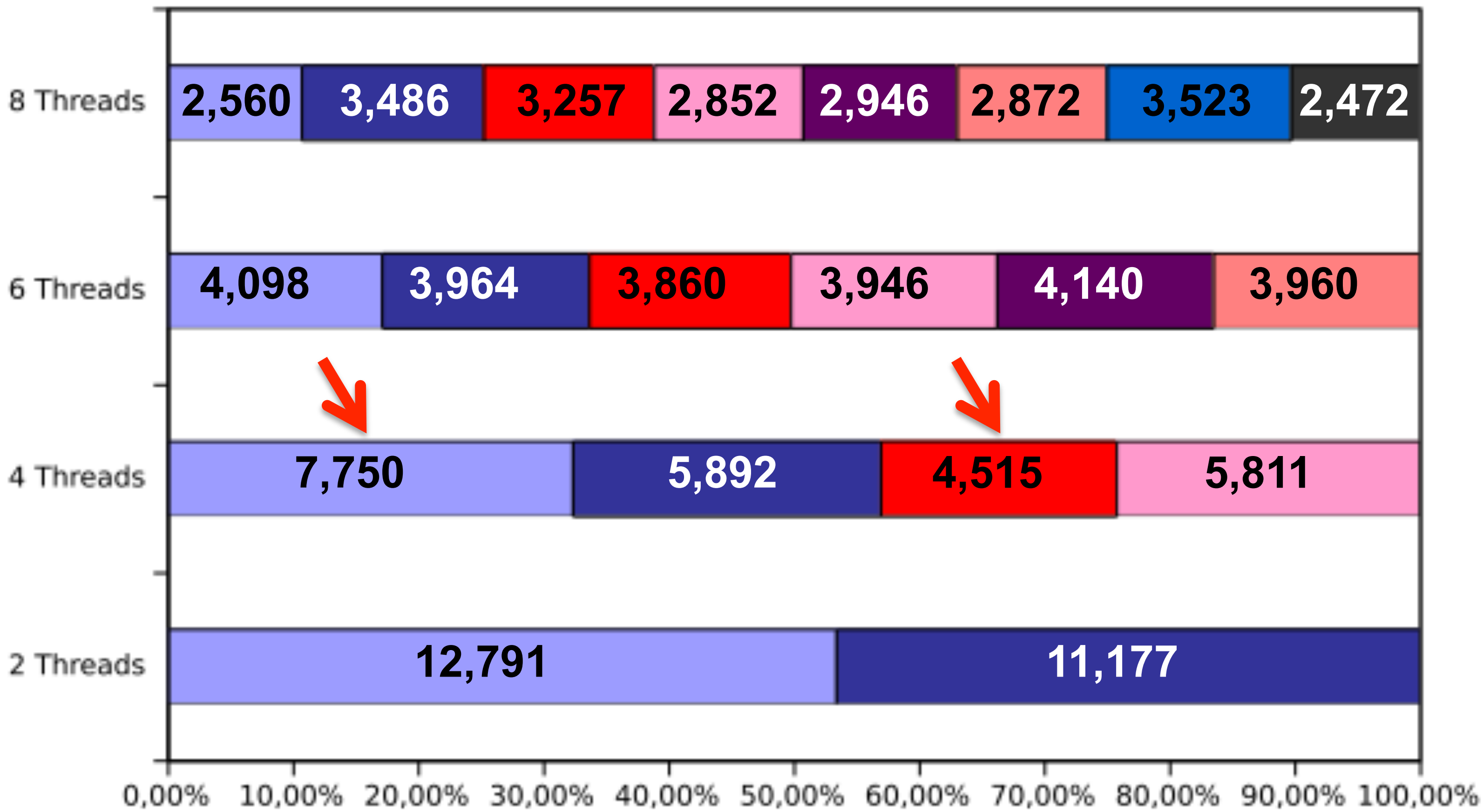
Distributed Queues



Each core has its own run/ret queues.

Evaluating Workload

Problem: Load Balancing

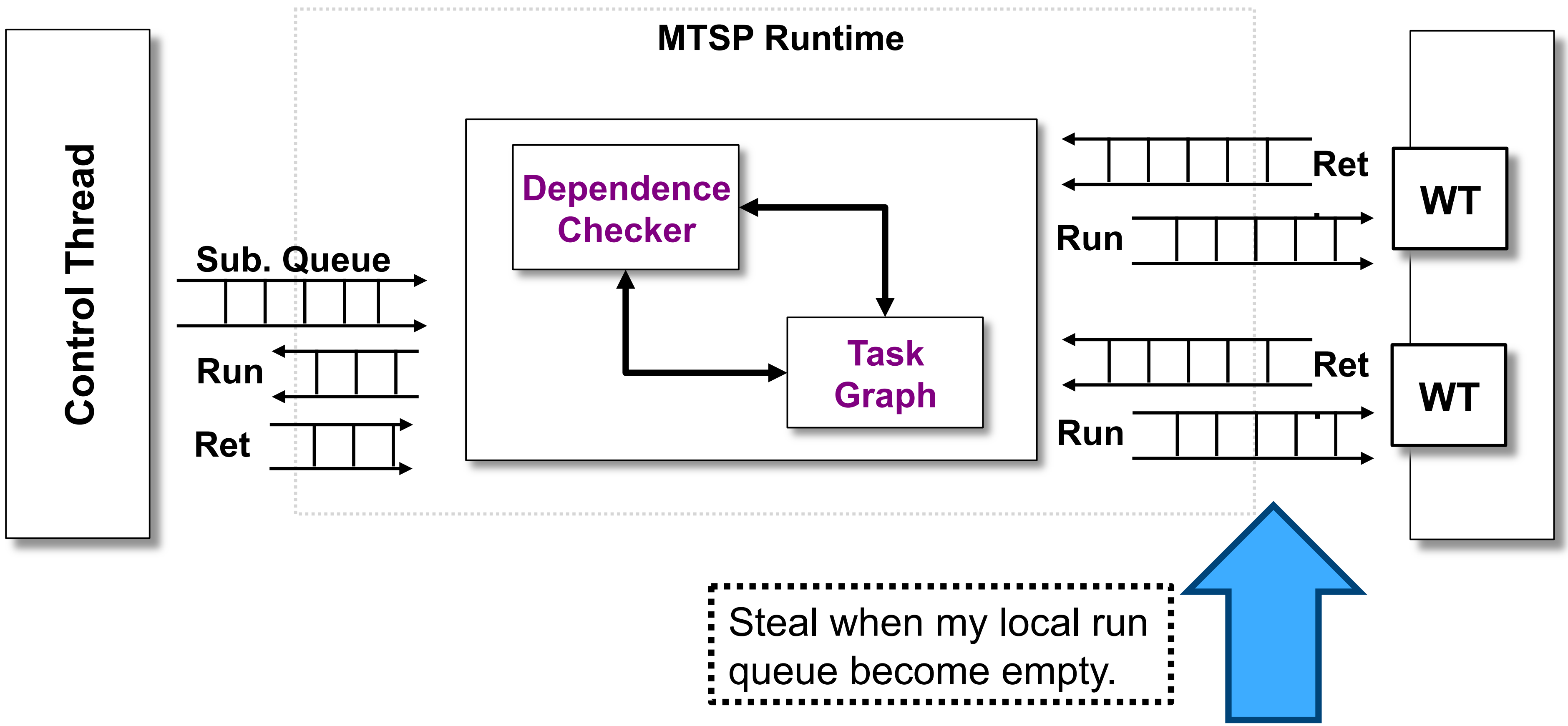


SparseLu -n 64 -m 64

Task Parallelism

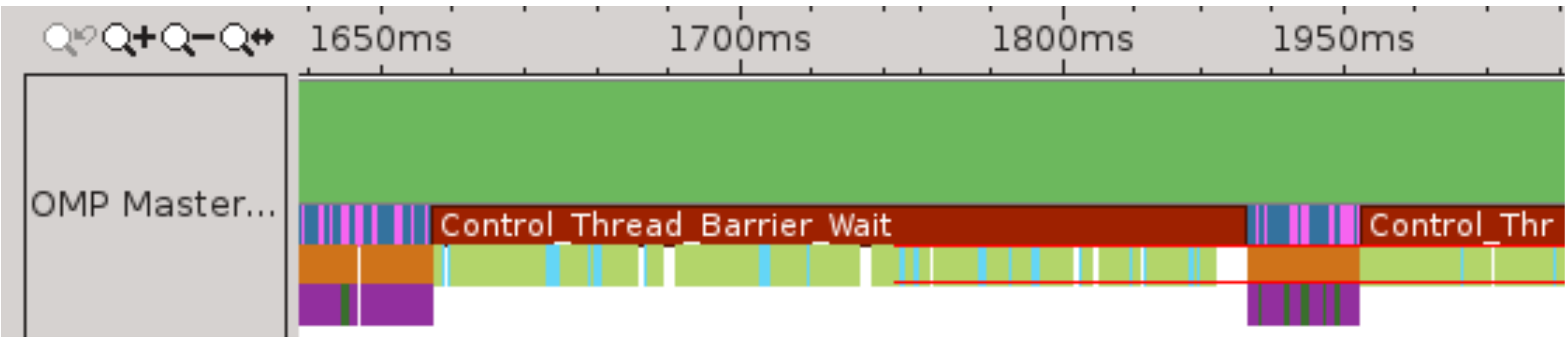
- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

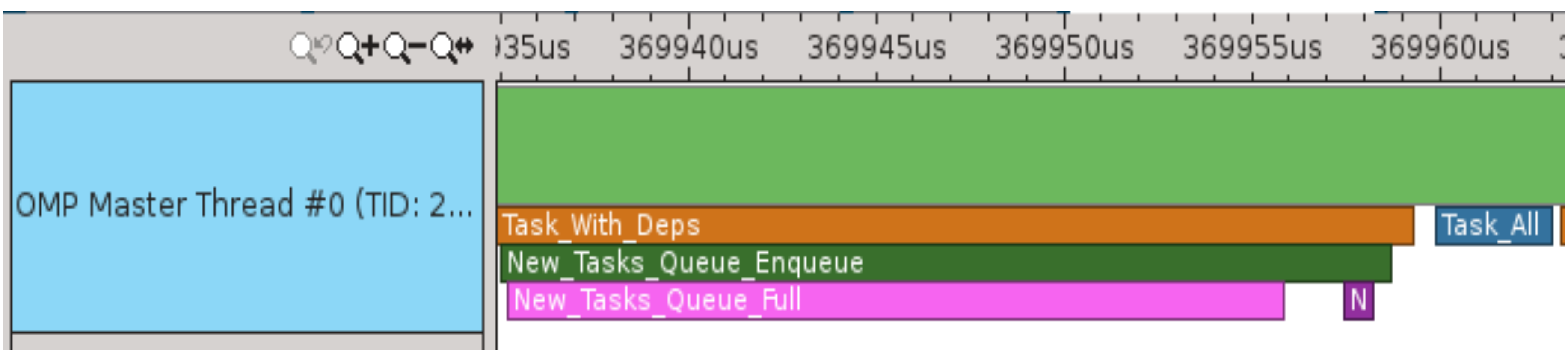
Work Stealing among Worker Threads



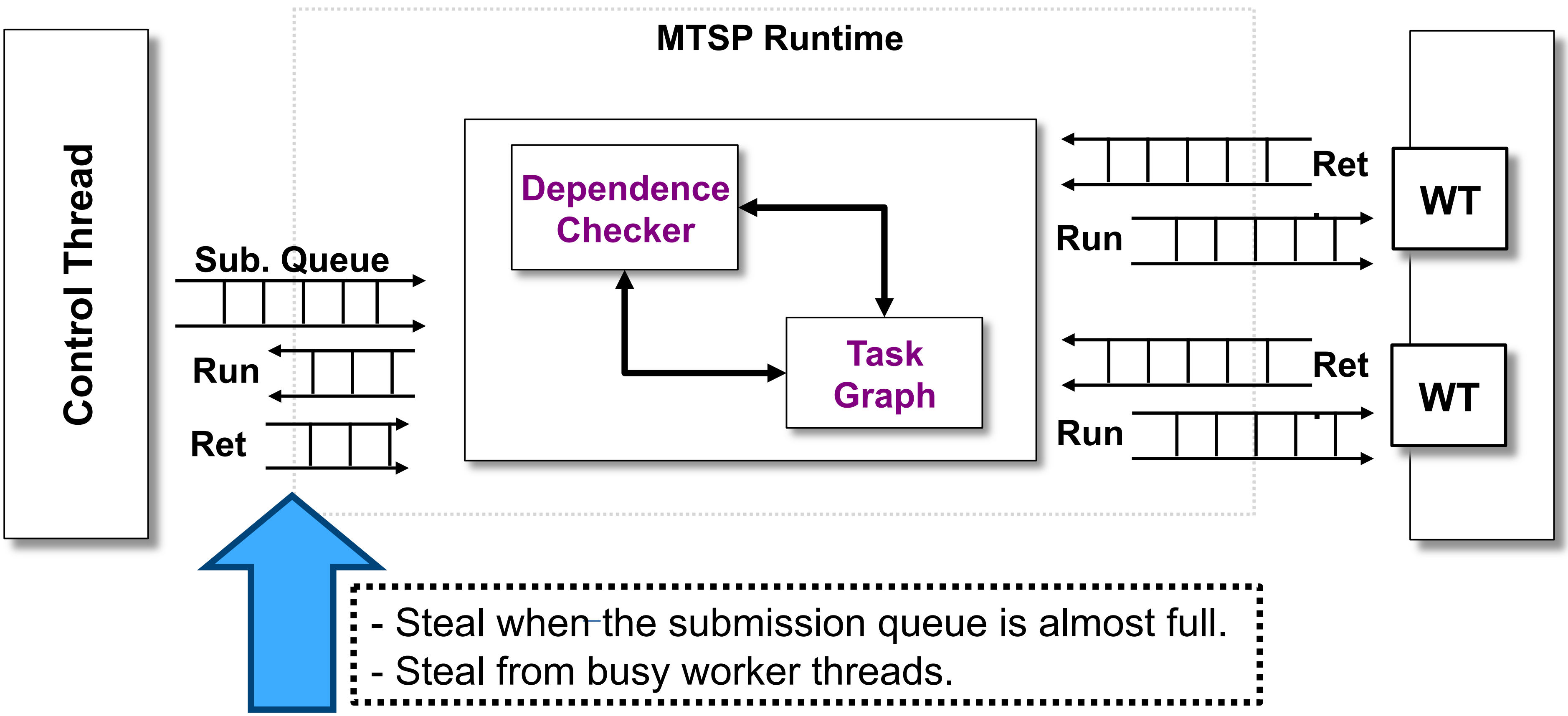
Control thread waiting

Solution: Work Stealing to control thread



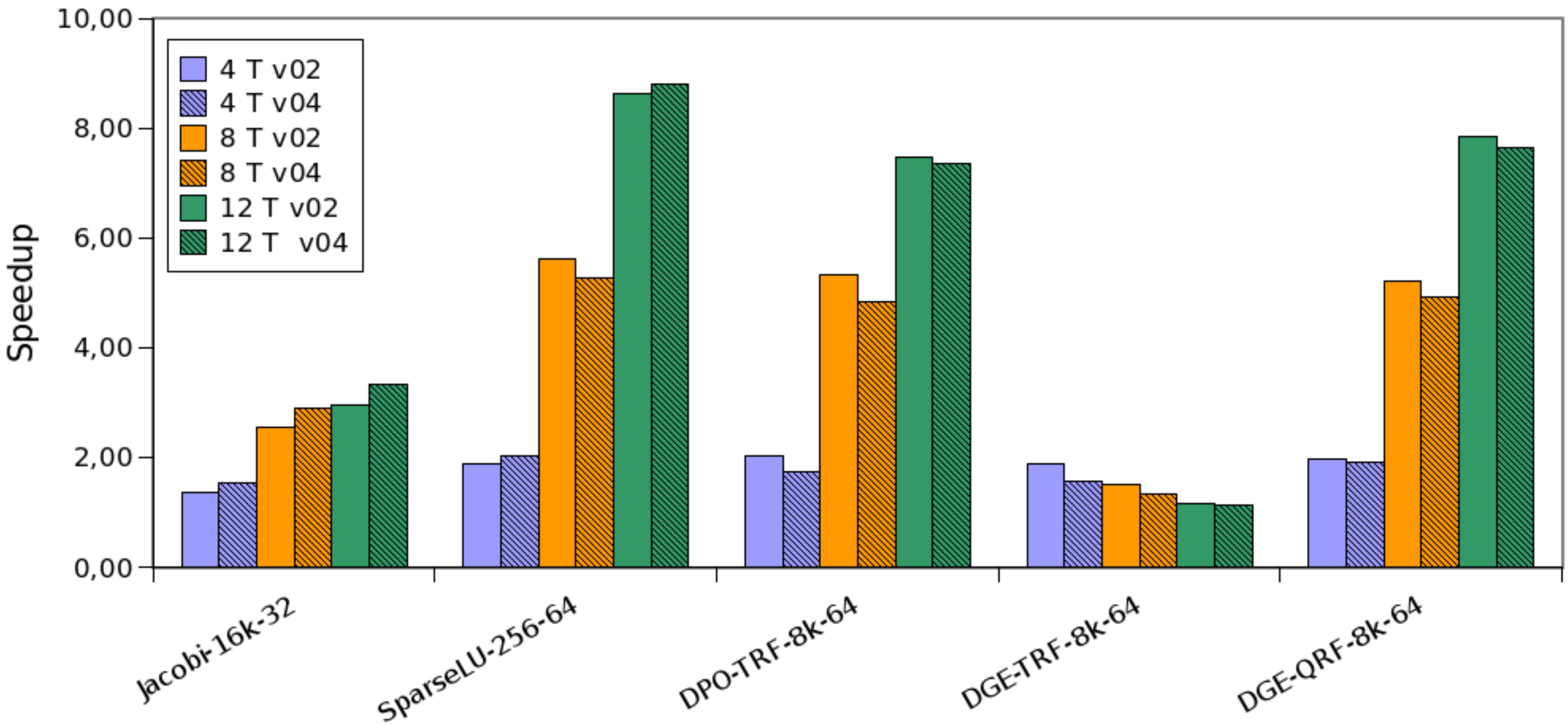


Work Stealing for the Control Thread



- Steal when the submission queue is almost full.
- Steal from busy worker threads.

Control Thread Stealing



Take away: for a larger number of threads performance improves!

Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Efficient Task Execution

Efficient Task Execution

Manager Cost	= 1 second
Task Size	= 1.5 seconds

Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds

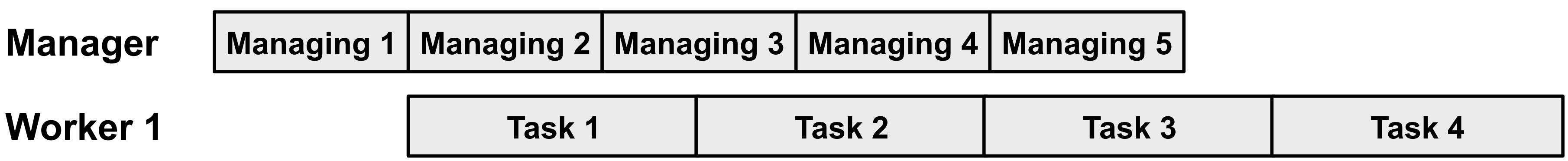
1 Worker Thread

Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds

1 Worker Thread

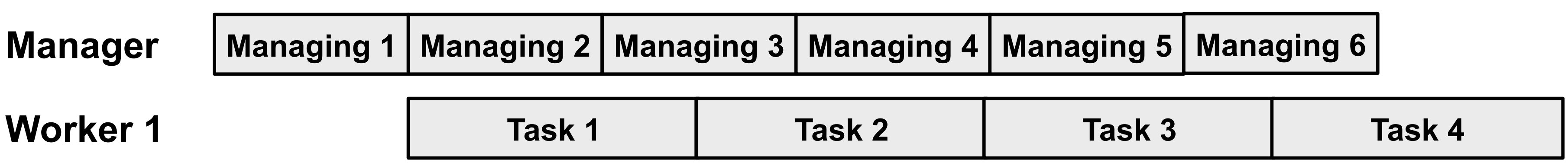


Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds

1 Worker Thread



One task is dispatched every 1 second.

One task is consumed every 1.5 seconds.

No Worker Thread is Idle

Efficient Task Execution

Manager Cost	= 1 second
Task Size	= 1.5 seconds

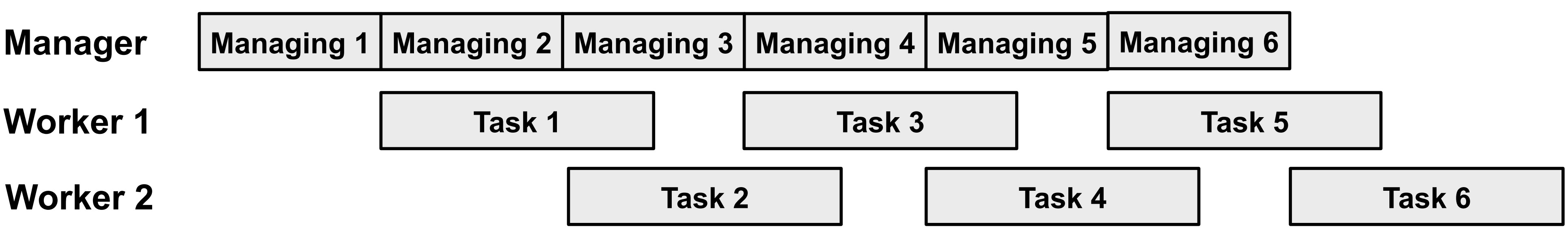
2 Worker Threads

Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds

2 Worker Threads



Two tasks are dispatched every 2 seconds.

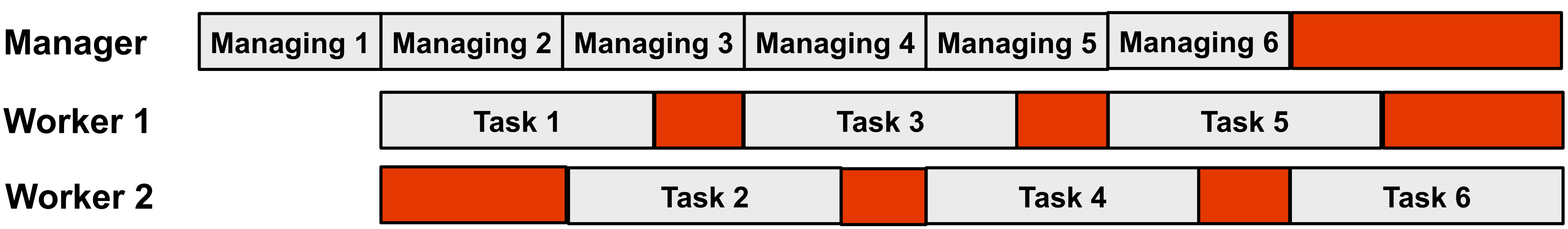
One task isconsumed every 1.0 seconds.

Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds

2 Worker Threads



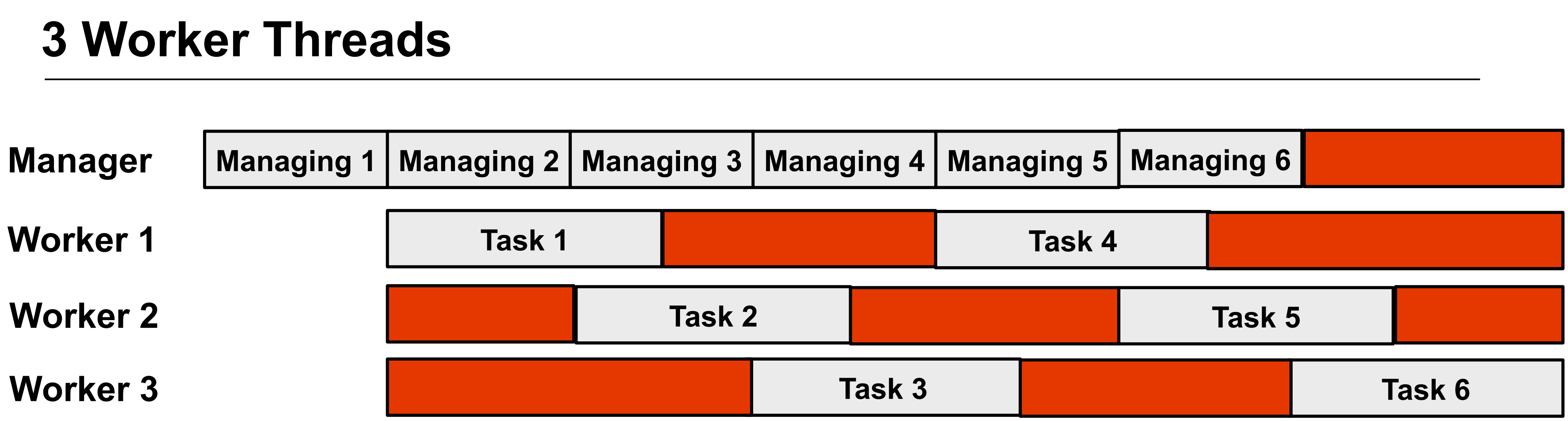
Two tasks are dispatched every 2 seconds.

Two tasks are consumed every 1.5 seconds.

Efficient Task Execution

Manager Cost = 1 second

Task Size = 1.5 seconds



Efficient Task Execution

Manager Cost = **0.5 second**

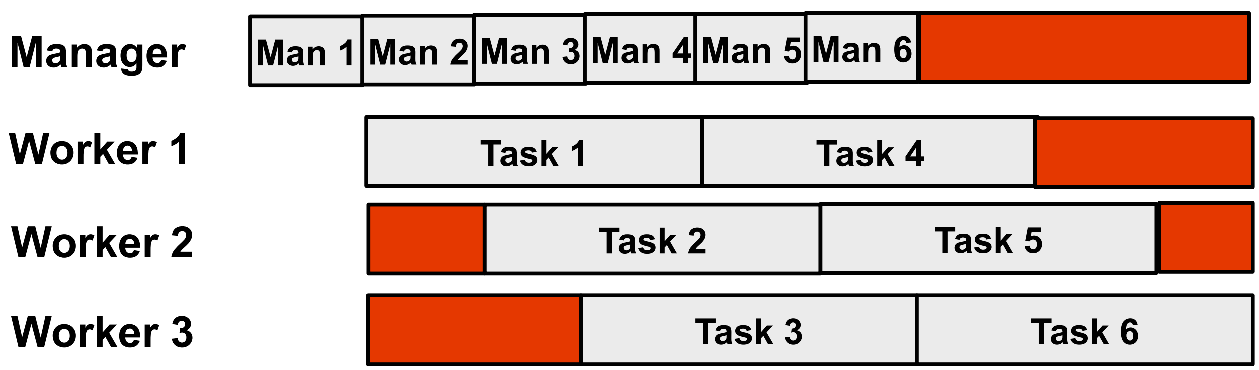
Task Size = 1.5 seconds

Efficient Task Execution

Manager Cost = **0.5 second**

Task Size = 1.5 seconds

3 Worker Threads



Efficient Task Execution

Rationale:

Task management throughput remained constant.

Task consumption throughput increased

Should activate only the needed number of workers

The Implementation

- **Task Size** → TSize
- **Runtime Cost** → RCost
- **Max Number of Workers** → ?

The Implementation

- **Task Size** ⟶ **TSize**
- **Runtime Cost** ⟶ **RCost**
- **Max Number of Workers** ⟶ **?**

The maximum number of workers for a given *TSize* and *RCost* is:

$$MaxWorkers = \frac{TSize}{RCost}$$

That is, the maximum number of tasks the runtime can dispatch during the execution of another task.

Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Runtime overhead

Task Data	SparseLU (-64)			
Task ID	402830	402860	4028a0	4028e0
Task cycles	246287	429212	382085	649154
Runtime cycles	9896	19457	25522	33167
#Tasks	64	1024	1024	21856

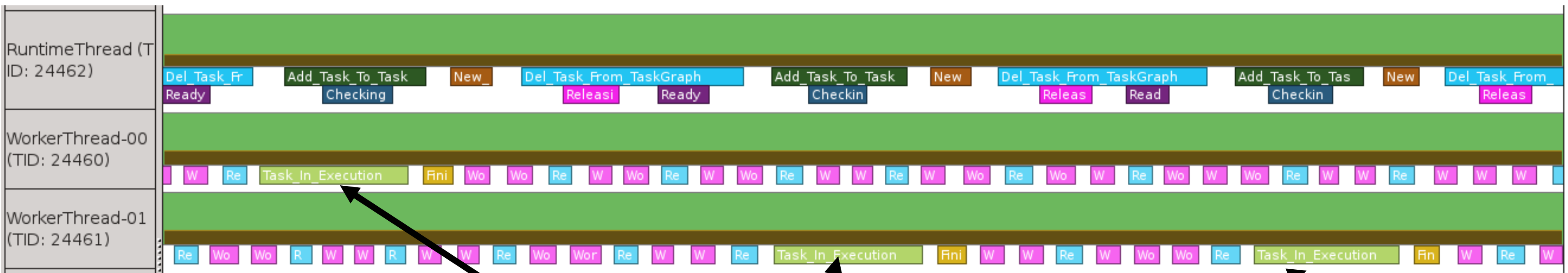
Large tasks

Task Data	Jacobi (-32)			
Task ID	401ab0	401ea0	403040	403170
Task cycles	35736	100728	1682	6742
Runtime cycles	497	637	15234	38907
#Tasks	512	1024	4096	4096

Small tasks with many dependences

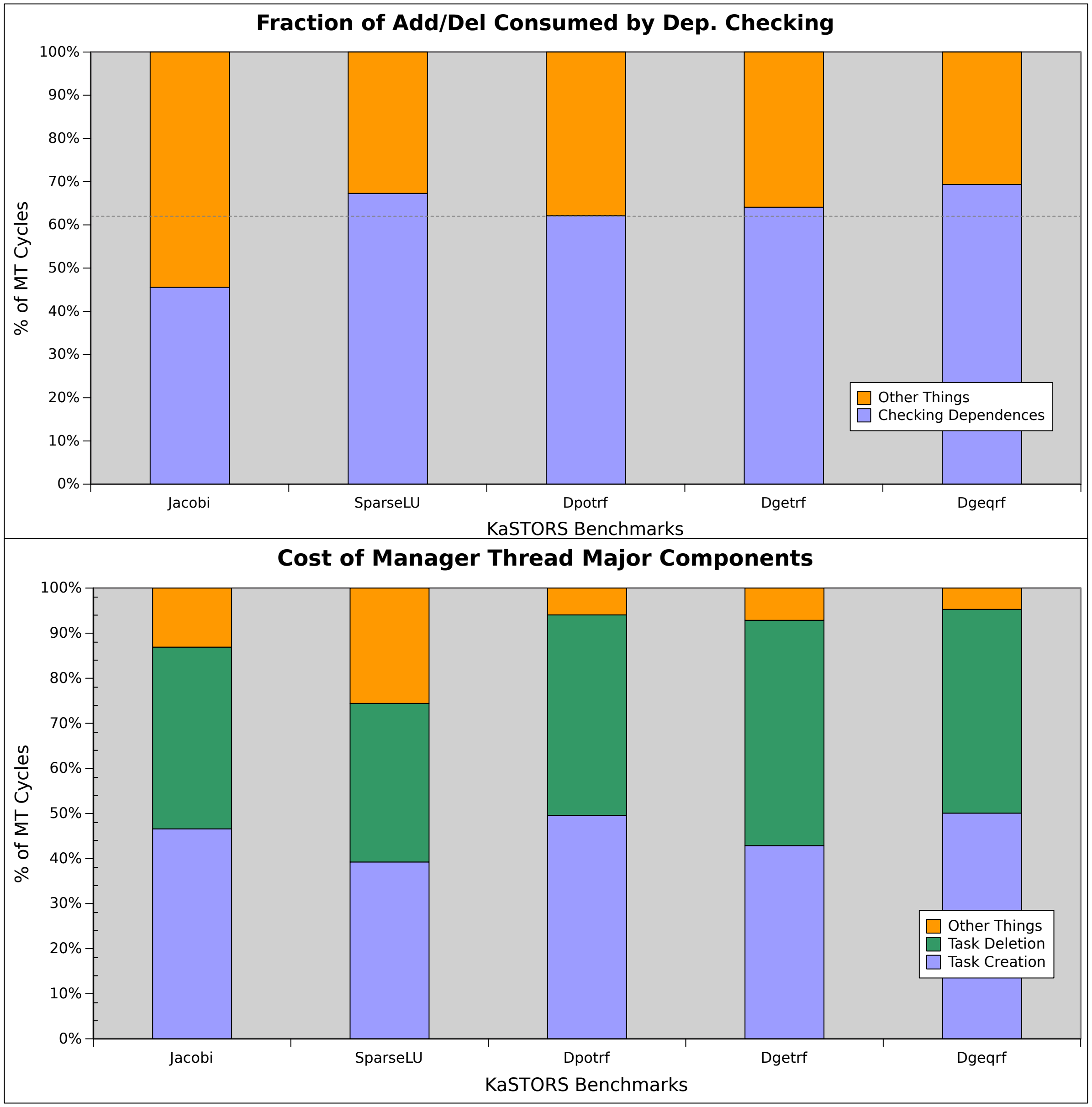
Profiling the MTSP Runtime

What are the major sources of overhead?



Task Size → <u>2,000 Add & Mul</u>

MTSP Overhead Split

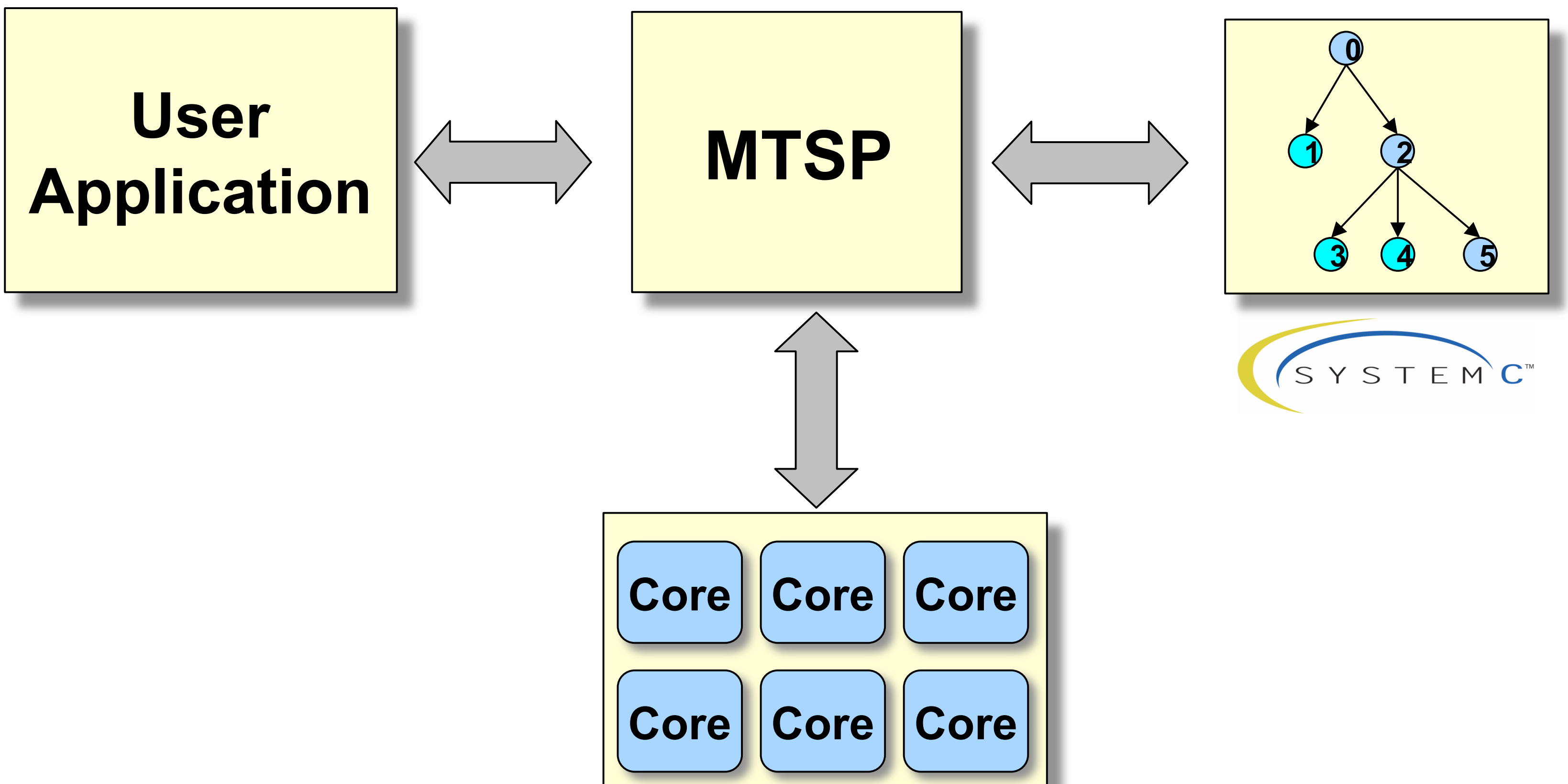


Need to reduce the runtime overhead!

What if we use hardware support?

Solution: Hardware-enabled task management

Task Graph Accelerator (TGA)



Task Parallelism

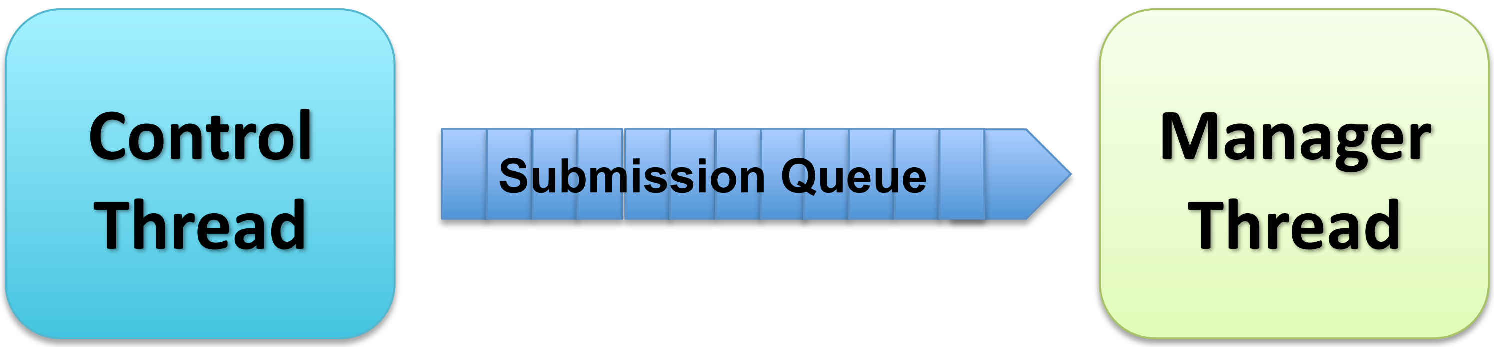
- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Research
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Task Coalescing

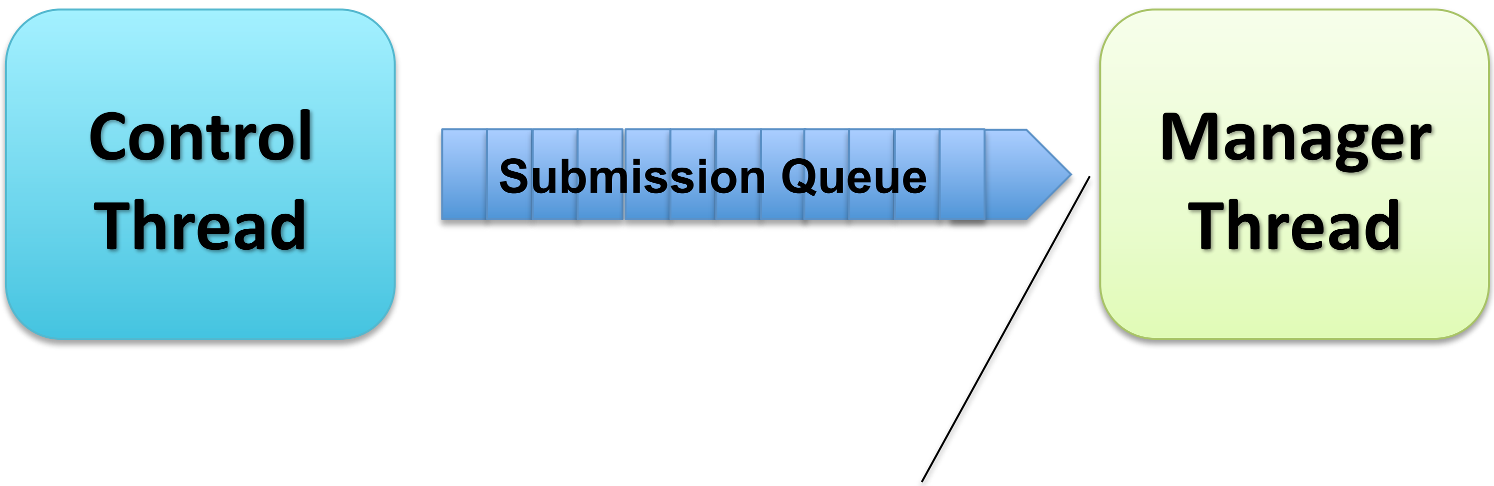
- What is it?
  - Group together subsequent tasks from the submission queue.

- Why do it?
  - Reduce the average time spent with task management.

Task Coalescing



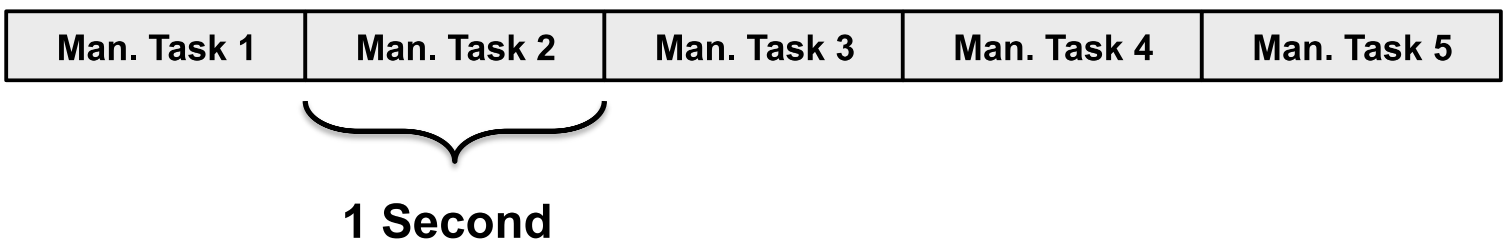
Task Coalescing




```
while (current_task() == prev_task())
      concatenate_tasks();
```

Task Coalescing

Manager Thread



Individual task management.

Task Coalescing

Manager Thread

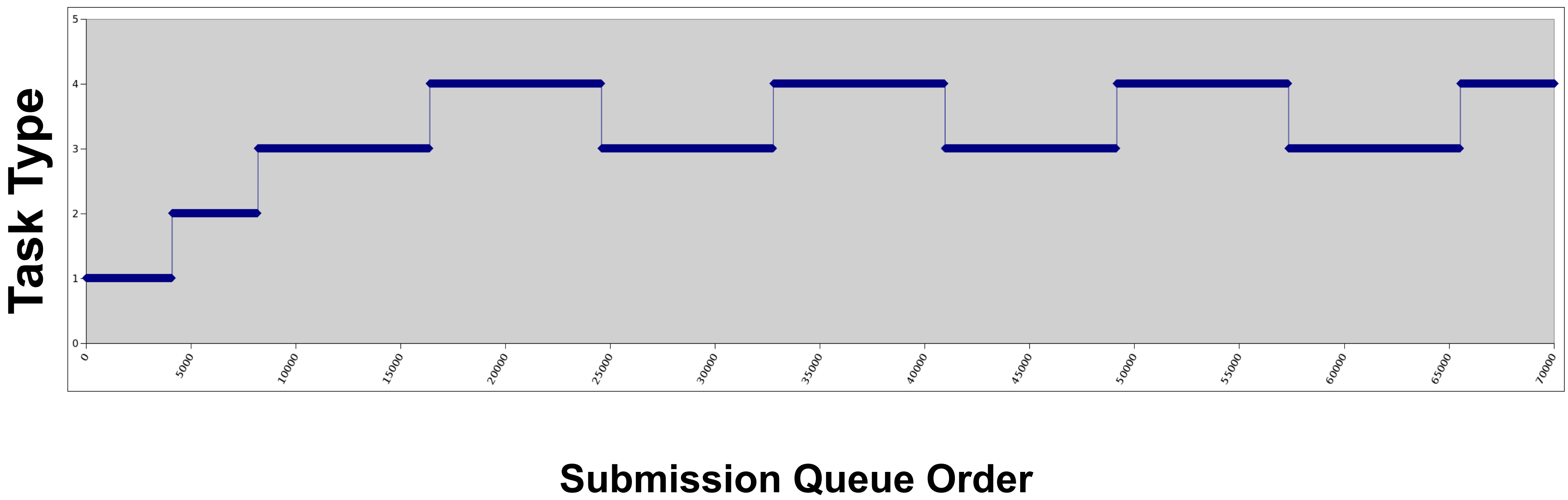
Task 1 & 2	**Task 3 & 4**	**Task 5 & 6**	**Task 7 & 8**	**Task 9 & 10**

1 Second...

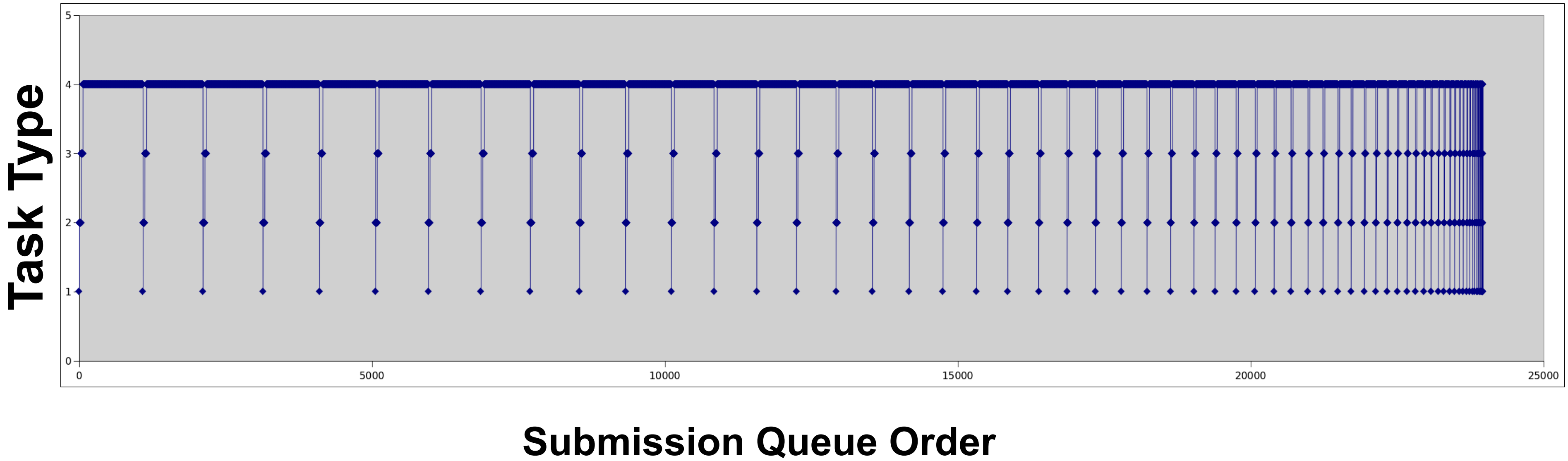
but 2 Tasks

Coalesced task management.

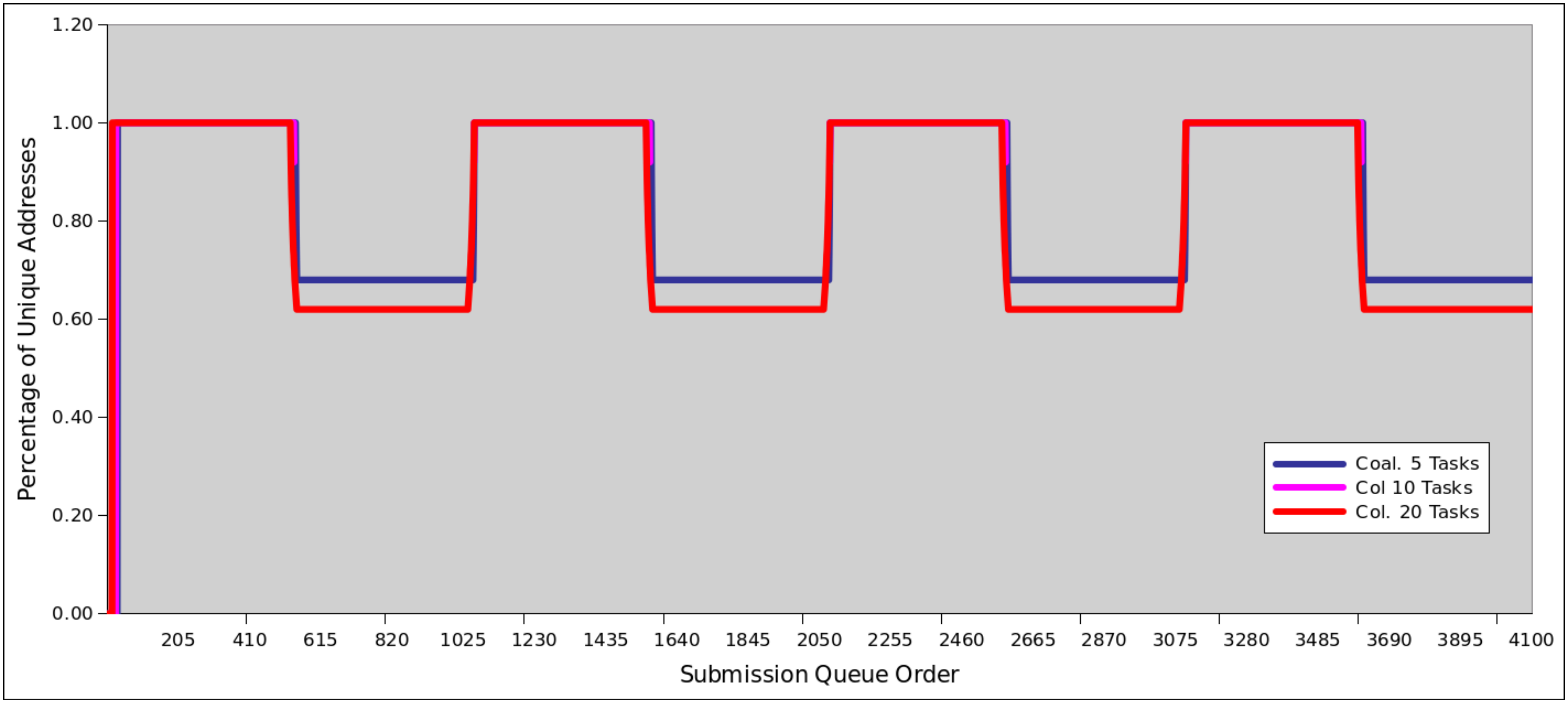
Task Submission Pattern (Jacobi)



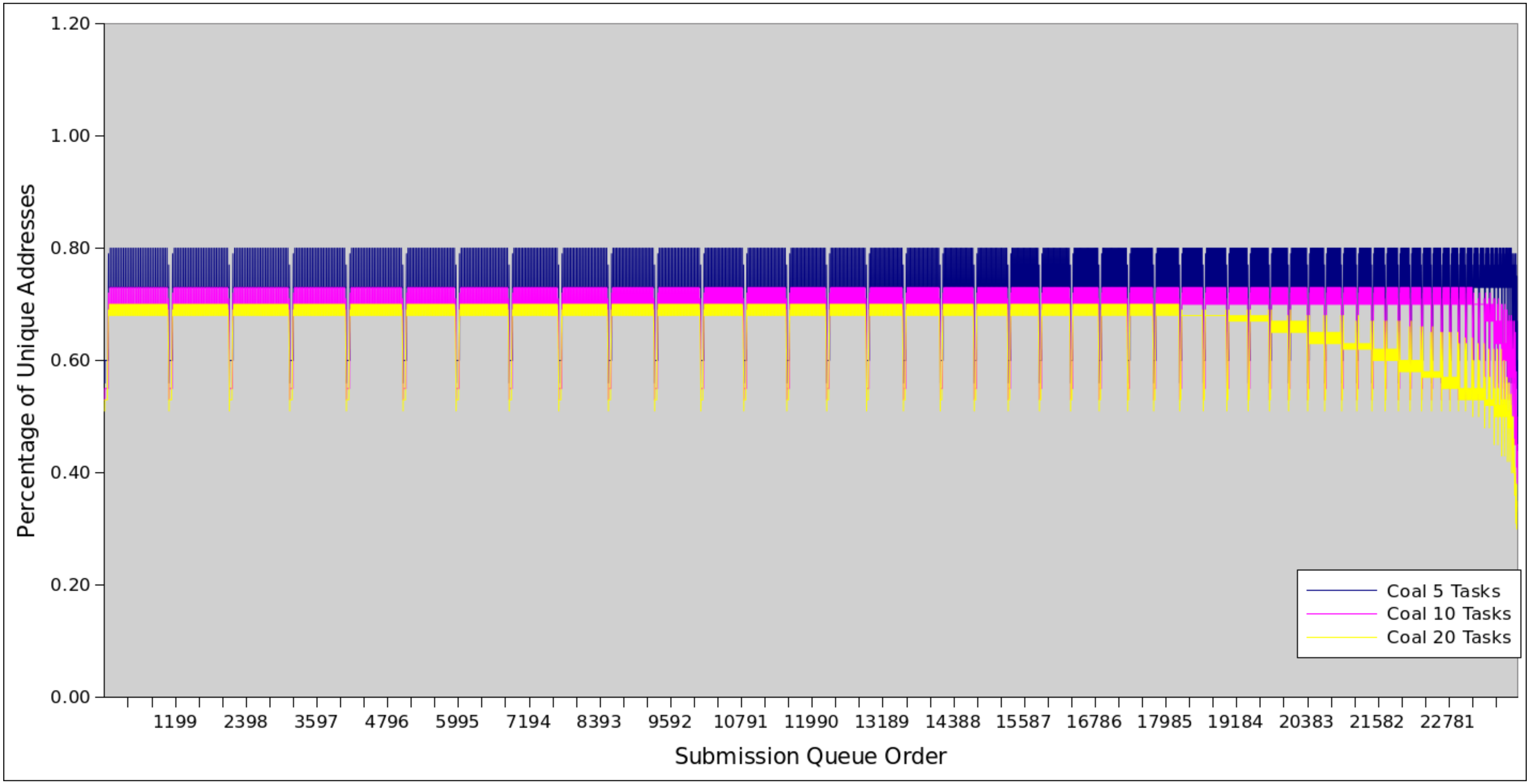
Task Submission Pattern (SparseLU)



Parameter Sharing Among Tasks



Parameter Sharing Among Tasks



Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

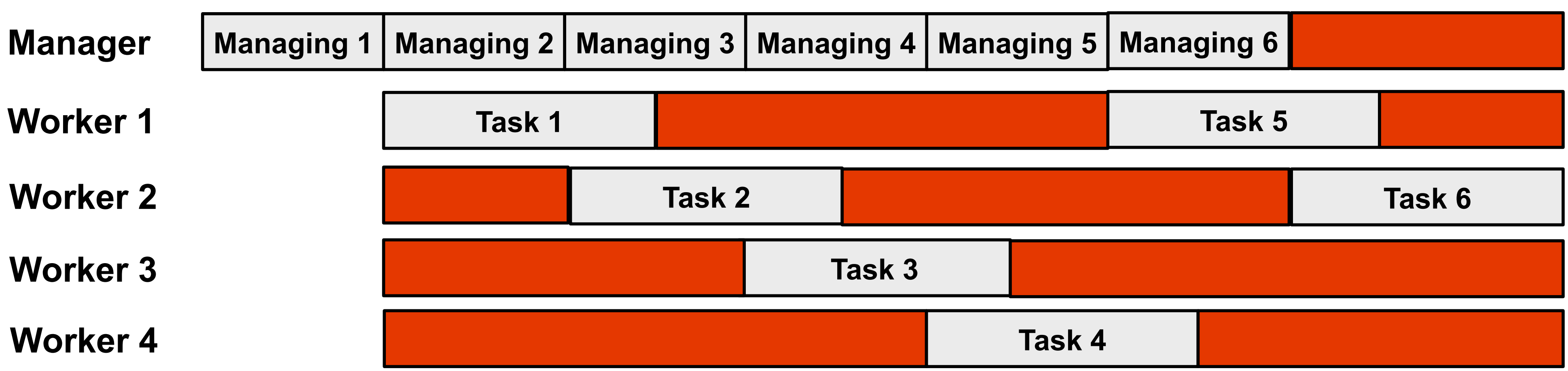
The Idea

Manager Cost	= 1 second	← **Difficult to improve!**
Task Size	= 1.5 seconds	← **Cannot control.**

The Idea

Manager Cost = 1 second ← **Difficult to improve!**

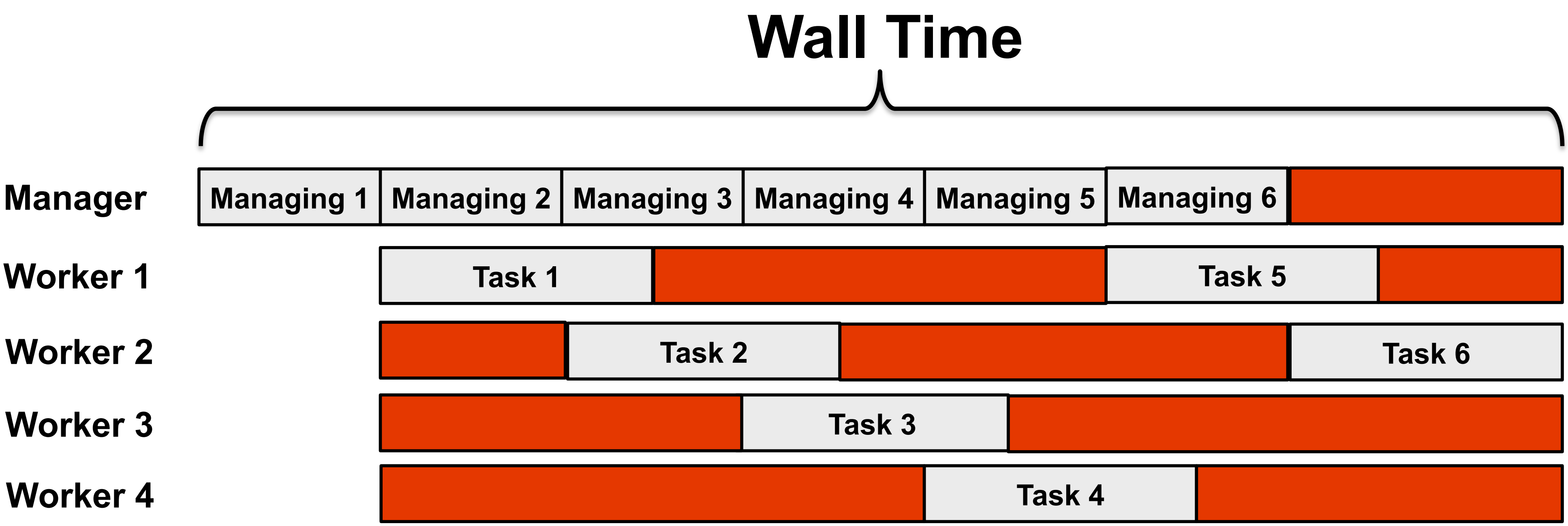
Task Size = 1.5 seconds ← **Cannot control.**



The Idea

Manager Cost = 1 second ← **Difficult to improve!**

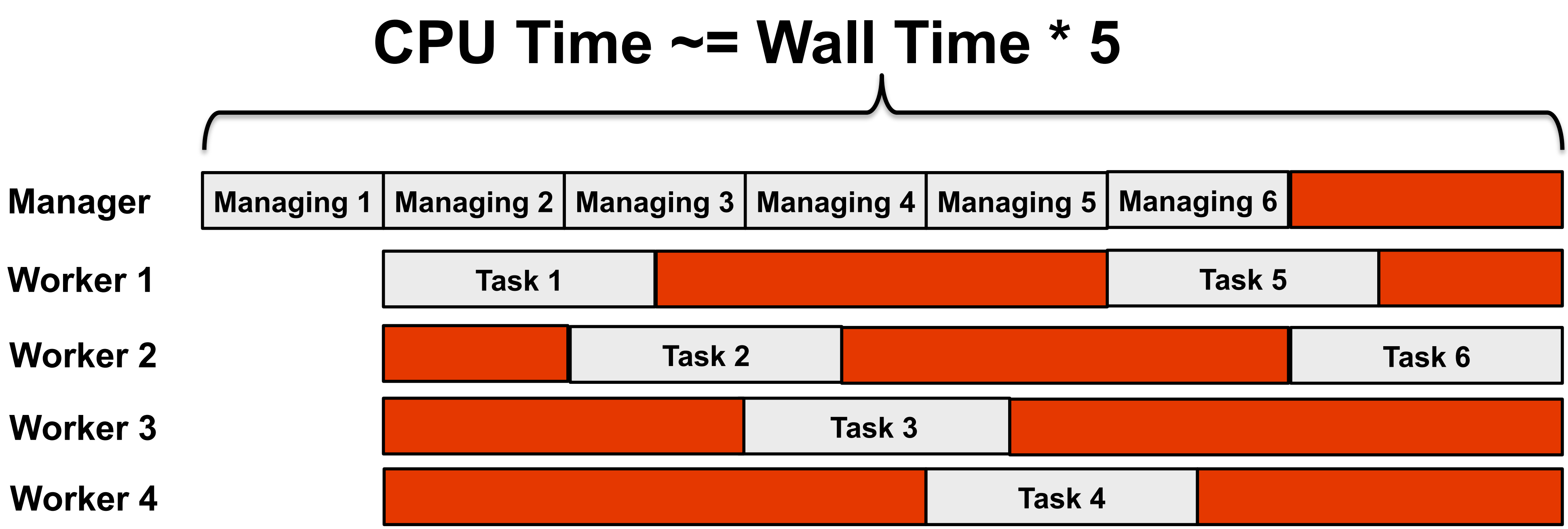
Task Size = 1.5 seconds ← **Cannot control.**



The Idea

Manager Cost = 1 second ← **Difficult to improve!**

Task Size = 1.5 seconds ← **Cannot control.**

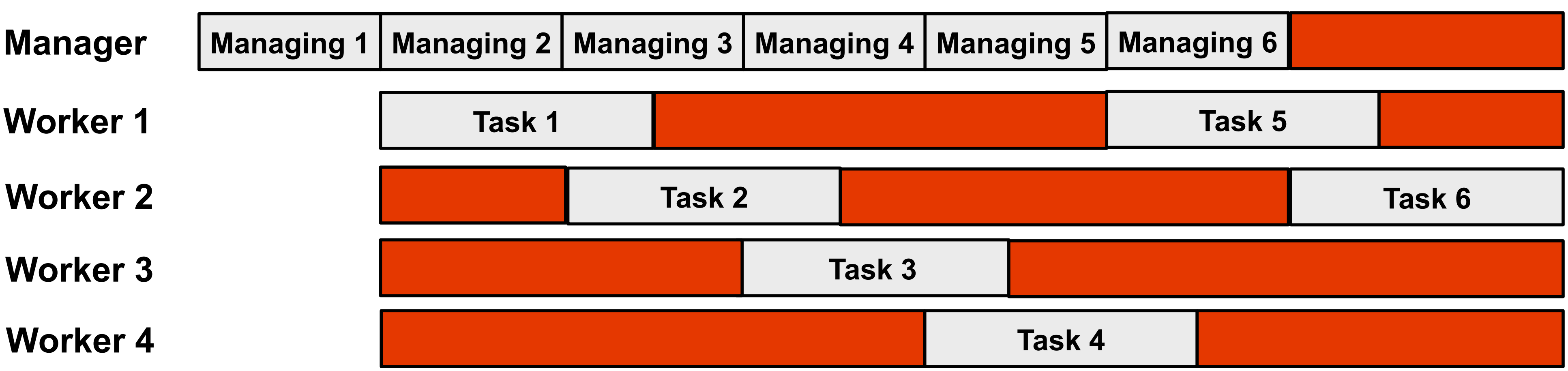


The Idea

Manager Cost = 1 second ← **Difficult to improve!**

Task Size = 1.5 seconds ← **Cannot control.**

Can we reduce CPU Time without degrading Wall Time?



The Implementation

- Always use a small number of threads!

The Implementation

- ~~Always use a small number of threads!~~

The Implementation

- ~~Always use a small number of threads!~~
- Programs have phases...

The Implementation

- ~~Always use a small number of threads!~~
- Programs have phases...
- Dynamically compute the instantaneous maximum number of workers supported!

Remembering

- **Task Size** → **TSize**
- **Runtime Cost** → **RCost**
- **Max Number of Workers** → **?**

The maximum number of workers for a given *TSize* and *RCost* is:

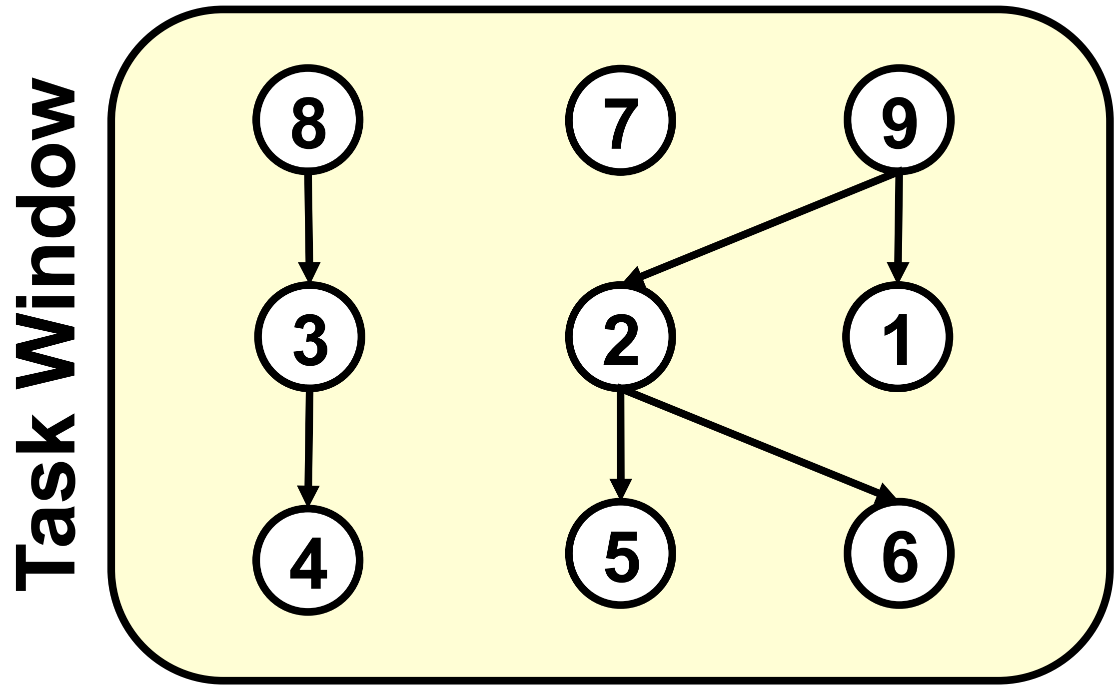
$$MaxWorkers = \frac{TSize}{RCost}$$

That is, the maximum number of tasks the runtime can dispatch during the execution of another task.

Task Parallelism

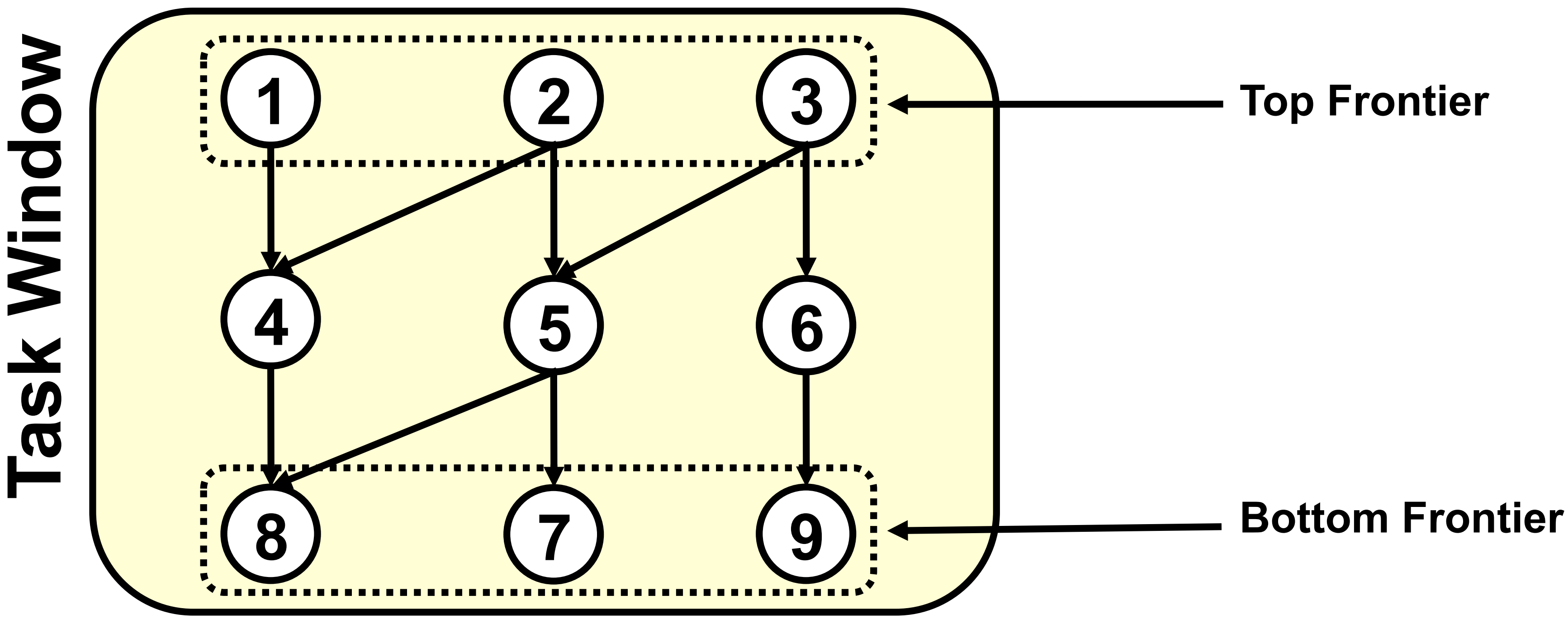
- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Critical Path Prediction and QoS

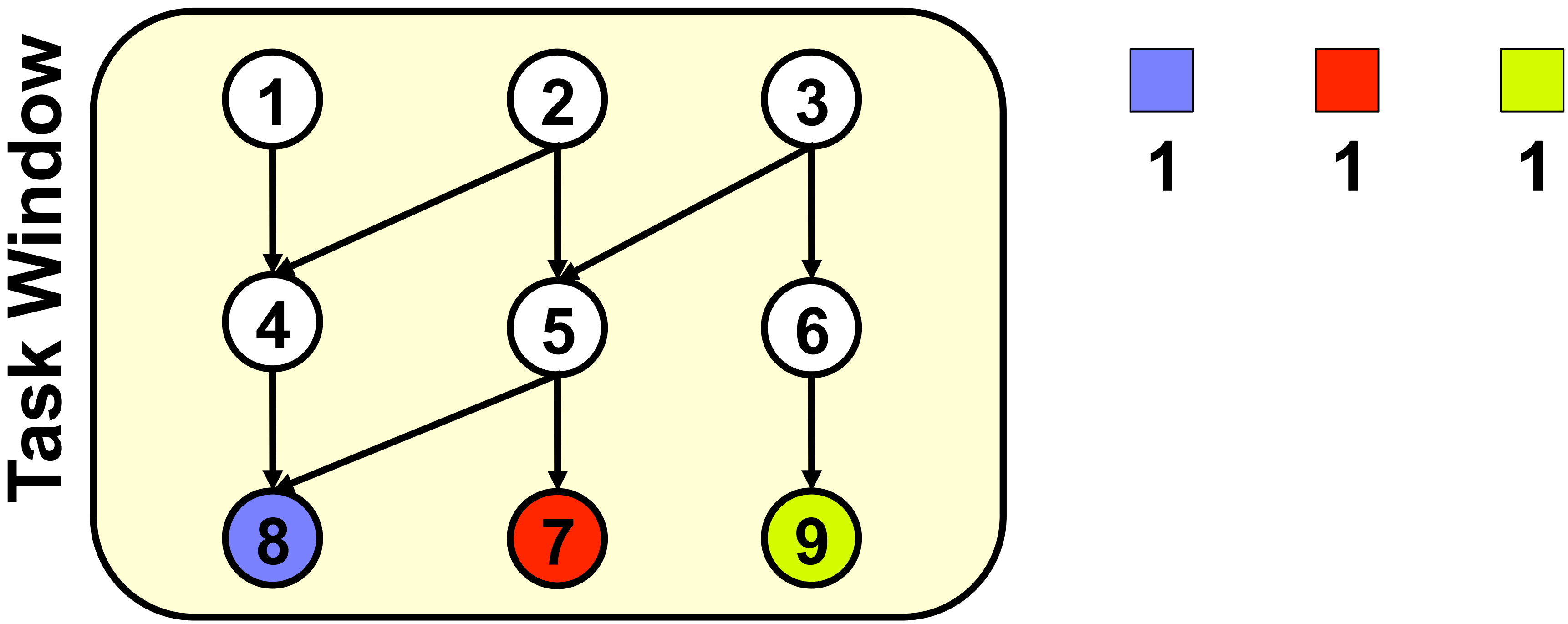


Which node should execute first? 7, 8 or 9?

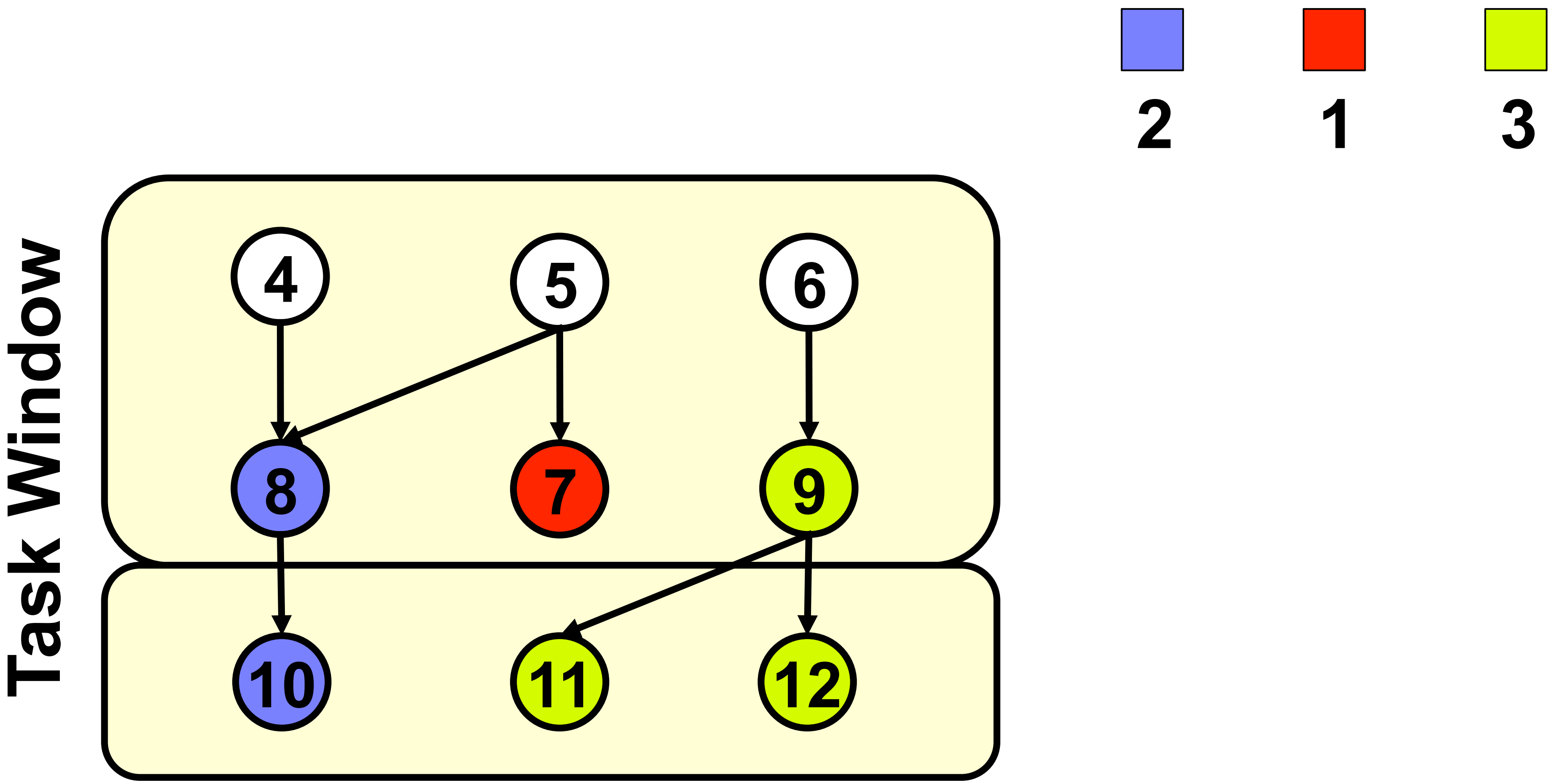
Critical Path Prediction and QoS



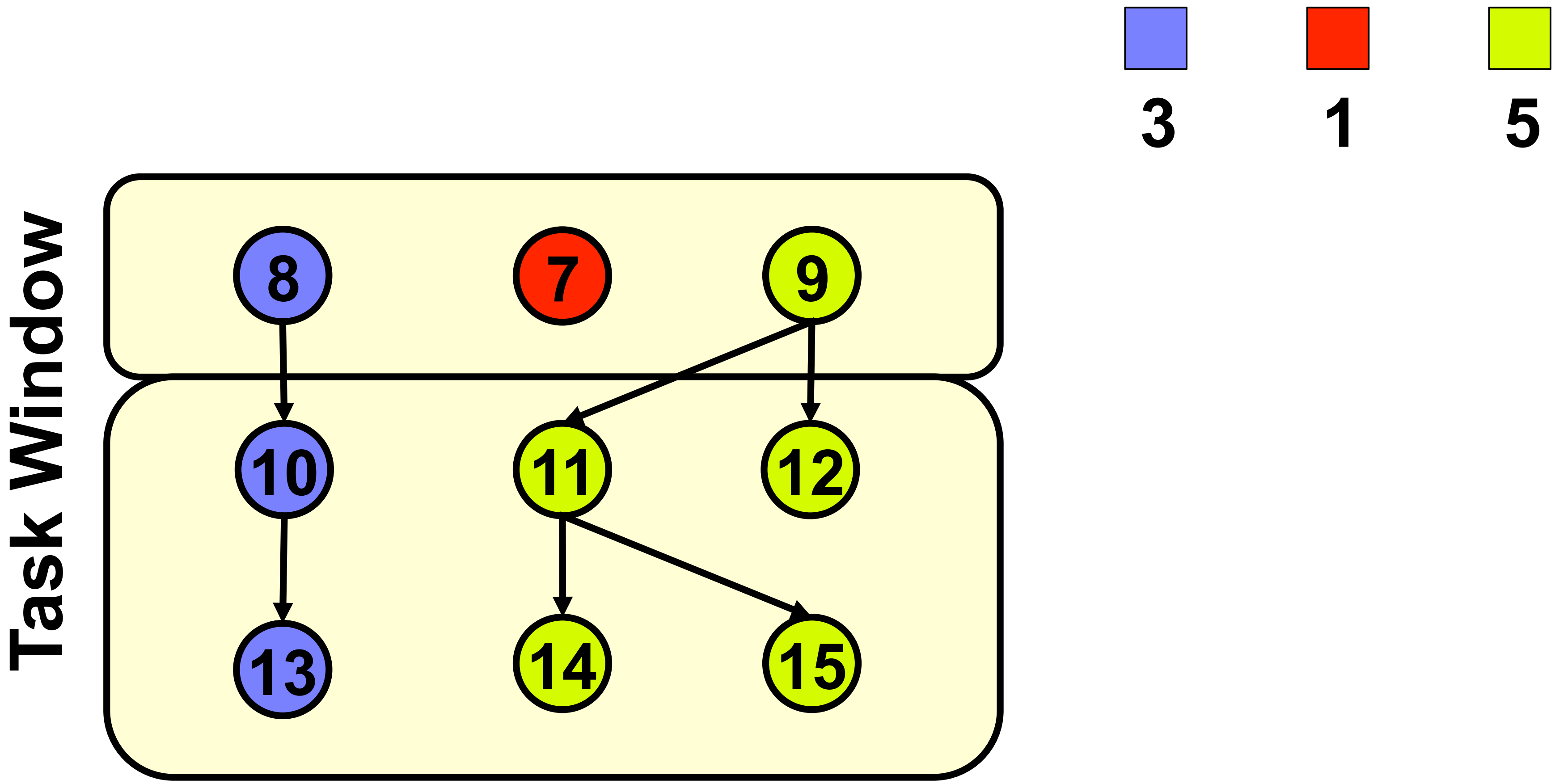
Critical Path Prediction and QoS



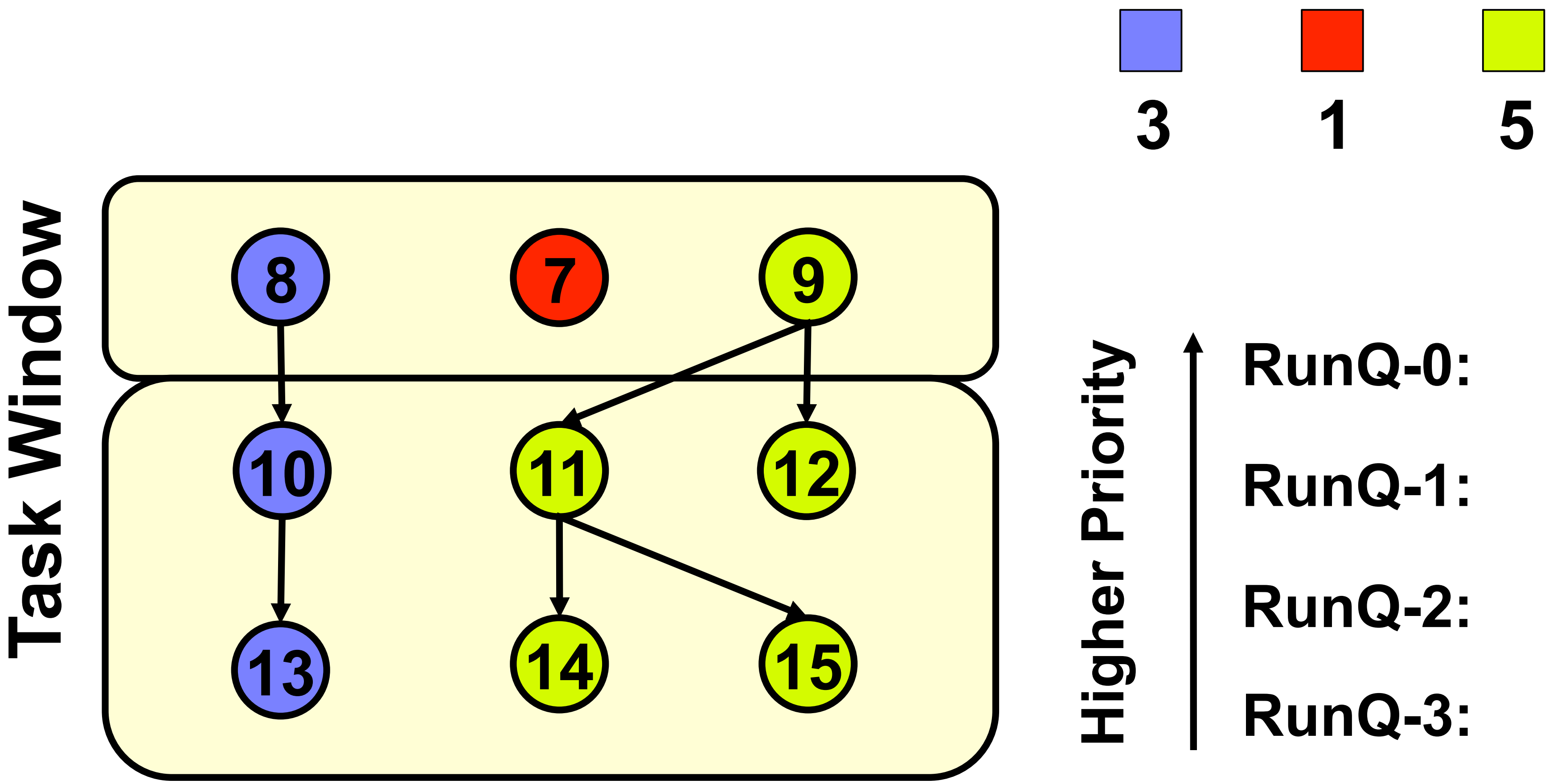
Critical Path Prediction and QoS



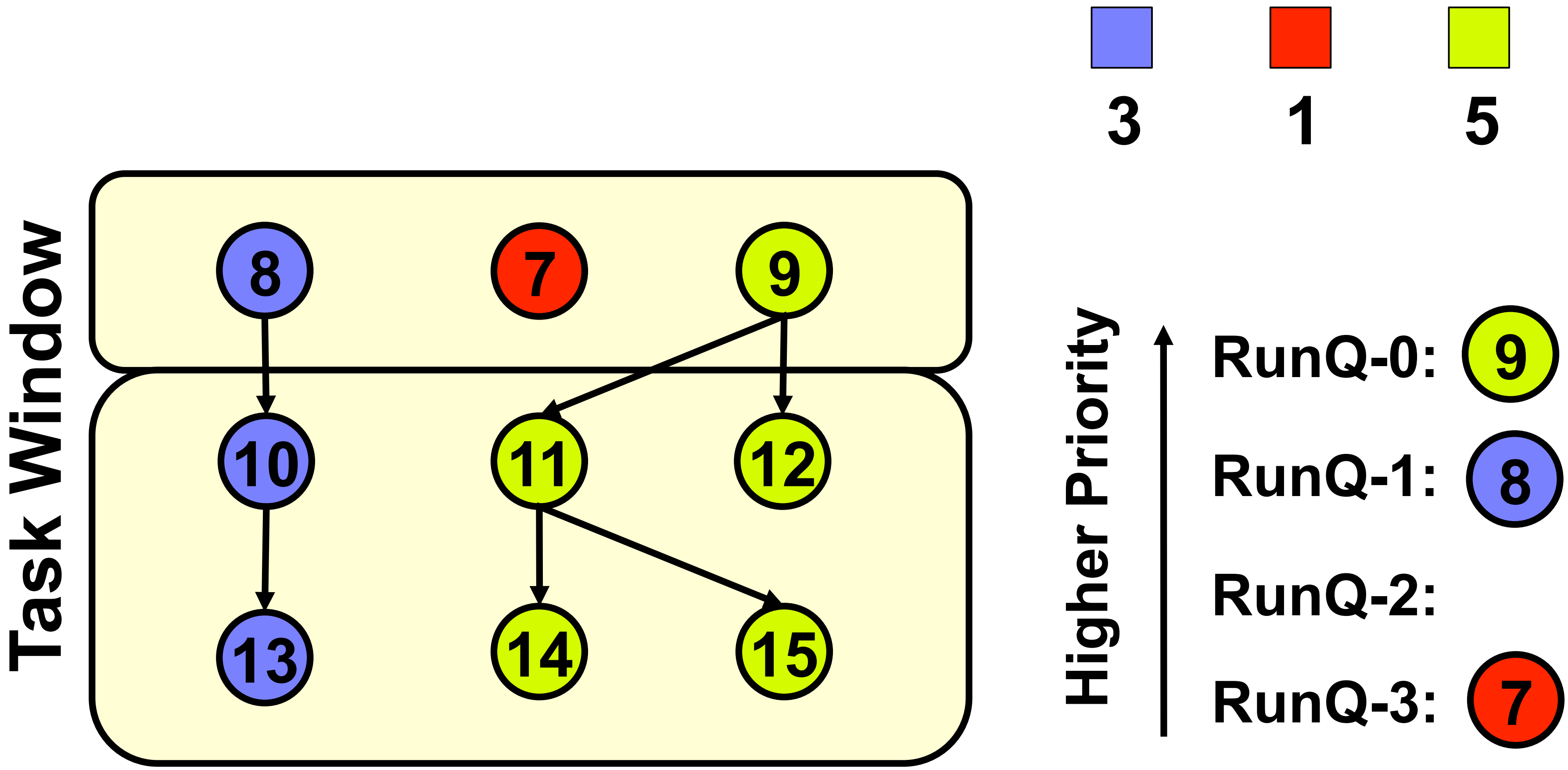
Critical Path Prediction and QoS



Critical Path Prediction and QoS



Critical Path Prediction and QoS



Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Optimization
  - Task coalescing
  - Dynamic worker deactivation
  - Workload balancing
  - Dependency analysis
- Example

Programming Support

```
#pragma omp parallel for check
for (i=0; i < N; i++) {
   a[i] = fun(&a[i]);
}
```

What if fun writes to a[i+1] ?!

- Programmers make mistakes
- Have to support loop-carried detection
- New check clause to OMP
- IWOMP 2014

Support Correctness

```
for (i=0; i < N; i++) {
   #pragma omp task in(u[i]) out(v[i]) check
   fun(&u[i], &v[i]);
}
```

What if fun writes to u[i] ?!

- We intend to do the same for tasks
- Use escape analysis techniques

Support Performance

```
for (i=0; i < N; i++) {
  #pragma omp task in(u[i]) out(v[i]) check
  fun(&u[i], &v[i]);
}
```

- Does it pay off?
  - Runtime should estimate the speed-up
  - Evaluate task duration, runtime overhead and # worker threads
  - Develop heuristics to give hints to the programmer
  - It could latter be used for automatic annotation

Task Parallelism

- Programming model
- Runtime design
- Task stealing
- Runtime performance
- Runtime overhead
- Research
  - Task coalescing
  - Dynamic worker deactivation
  - Dependency analysis
- Example