Department of Mathematics, University of Padua

Ravenscar-EDF

Comparative Benchmarking of an EDF Variant of a Ravenscar Runtime

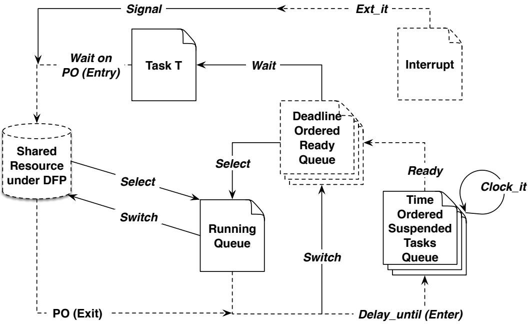
Ada Europe 2017
22nd Int'l Conference on Reliable Software Technologies

Paolo Carletto: `carletto.paolo@gmail.com`
Tullio Vardanega: `tullio.vardanega@math.unipd.it`

June 13, 2017

The RM-to-EDF Transformation Process

Turning Priorities into Deadlines



1. **Task Dispatching Policy**: from *"FIFO Within Priorities"* to *"EDF"*[1]
2. **Locking Policy**: from IPCP to DFP[2]

---

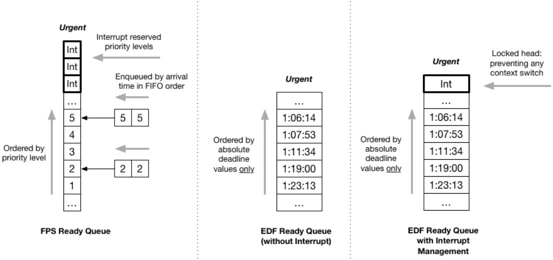
[1] A. Burns, An EDF Runtime Profile based on Ravenscar. Ada Lett. 33, 1 (June 2013)

[2] A. Burns and A. Wellings. The Deadline Floor Protocol and Ada. Ada Lett. 36, 1 (July 2016)

The RM-to-EDF Transformation Process

Implementation Challenges

Interrupt handling intrinsically assumes priorities, which – in principle – do not belong in an EDF system



Urgent

Int
Int
Int
...
5
4
3
2
1
...

Interrupt reserved priority levels

Enqueued by arrival time in FIFO order

5 5

2 2

Ordered by priority level

FPS Ready Queue

Urgent

...
1:06:14
1:07:53
1:11:34
1:19:00
1:23:13
...

Ordered by absolute deadline values only

EDF Ready Queue (without Interrupt)

Urgent

Int
...
1:06:14
1:07:53
1:11:34
1:19:00
1:23:13
...

Locked head: preventing any context switch

Ordered by absolute deadline values only

EDF Ready Queue with Interrupt Management

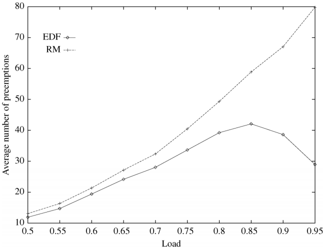
- Our solution reserves a fictitious position at the top of the ready queue for the current interrupt handler
    - If an interrupt handler is active, that position is used and the deadline-based part of the queue is frozen
    - If no interrupt is running, that position is not in use and cannot be contended

Evaluation Results

Buttazzo's Analysis

Buttazzo claimed EDF better than RM (FPS) in many respects

- Lower runtime overhead
    - Less preemptions
- Easier analysis
- More robust under overloads
    - Transient
    - Permanent

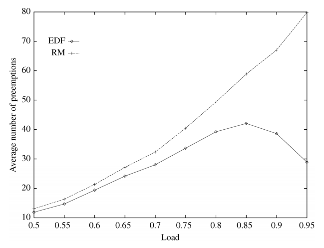


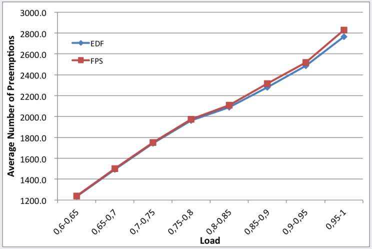
Evaluation Results

Beyond Buttazzo's Results

What is weak in Buttazzo's analysis?

- Task cardinality too small (10-30 tasks) to be significant
- Overload analysis confined to specific cases and not sufficiently general
- Different preemption behavior observed under 100%





- Lack of practical implementation and analysis of resource sharing protocols

Evaluation Results

Highest Schedulable Utilization

Which tasksets achieved the highest schedulable utilization in each runtime variant?

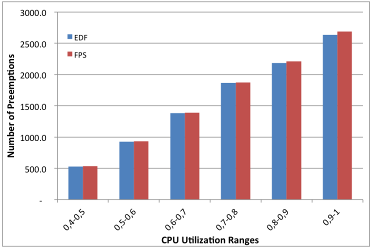
Taskset Type	Task Types	Delta Schedulable Utilization	Max CPU Load	EDF			FPS		
				RC	DM	PR	RC	DM	PR
Constrained	Short & Mid	**2,89%**	105,50%	30.714	0	3.637	29.850	415	6.202
Implicit	Mid Only	**3,72%**	102,63%	18.691	0	837	18.021	673	2.040
Constrained	All	**0,05%**	104,06%	24.398	0	5.131	24.409	0	5.211
Implicit	All	**5,22%**	100,85%	24.935	953	6.309	26.236	0	5.715

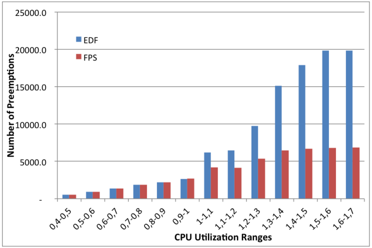
- **RC**: count of regular completions
- **DM**: count of deadline misses
- **PR**: count of preemptions

Evaluation Results

Runtime Overhead

Do the less preemptions and context switches that EDF incurs justify the higher costs of its scheduling operations?

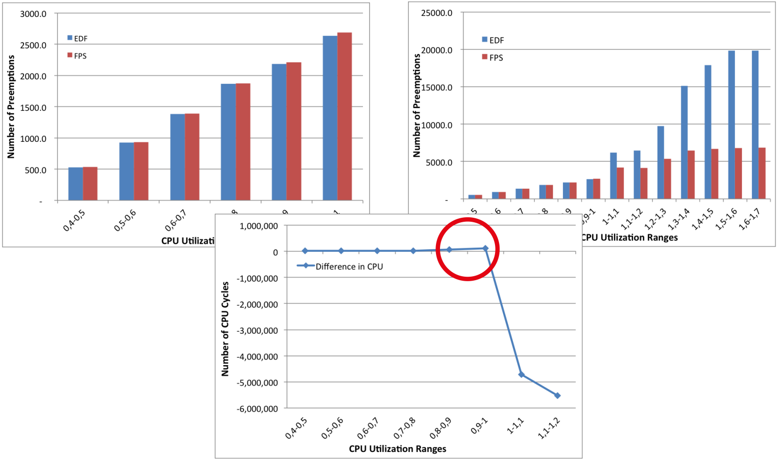




Evaluation Results

Runtime Overhead

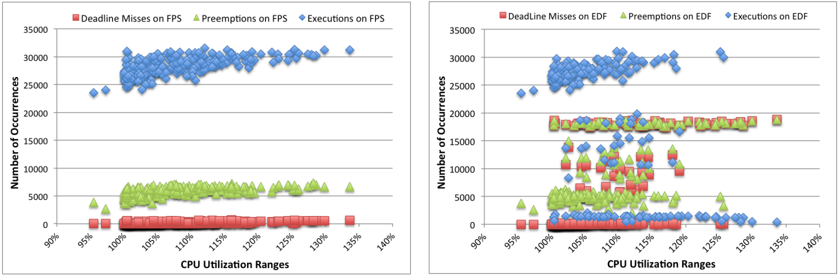
Do the less preemptions and context switches that EDF incurs justify the higher costs of its scheduling operations?



Evaluation Results

Resilience to Overload Situations

What happens to EDF and FPS under overload conditions, when the CPU utilization exceeds 100%?

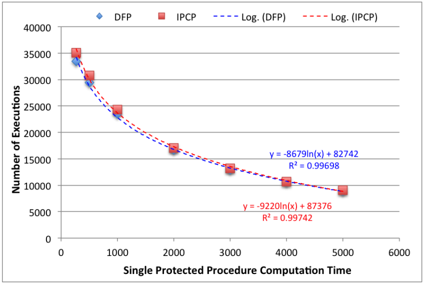


- FPS presents a linear behavior
- EDF's behaviour varies dramatically depending on the nature of the overload situation
    - Transient vs permanent

Evaluation Results

Locking Policy

How does DFP perform compared to IPCP?



- It presents a logarithmic converging progression as the computation time of the protected procedure increases
- DFP incurs more cumulative overhead than IPCP
  - Due to the need to read the clock in checking absolute deadlines

Future Work

Multilayered Scheduling

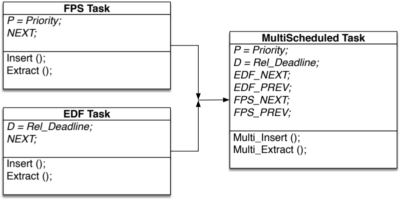
How can we take benefit of the best of both?

- EDF generates a feasible schedule (if any exists) within 100% CPU utilization
- FPS has more resilience beyond 100% CPU utilization

The Solution

A co-existence of both algorithms should yield the best of both worlds: EDF "becomes" FPS above 100% load

- A double linkedlist could offer a quick switch mechanism
- It should be based on a threshold value computed dynamically by the runtime on the idle time



FPS Task

P = Priority;
NEXT;

Insert ();
Extract ();

EDF Task

D = Rel_Deadline;
NEXT;

Insert ();
Extract ();

MultiScheduled Task

P = Priority;
D = Rel_Deadline;
EDF_NEXT;
EDF_PREV;
FPS_NEXT;
FPS_PREV;

Multi_Insert ();
Multi_Extract ();

A bareboard runtime lib for time-predictable parallelism

Davide Compagnin (PhD candidate),
Tullio Vardanega
University of Padova

Moral

- When you seek *sustainable time-composable* parallelism, mind what you abstract away of the (manycore) processor hardware
- Implementation experience suggests that you should hide *much less* than used to be with concurrency

Kalray MPPA-256

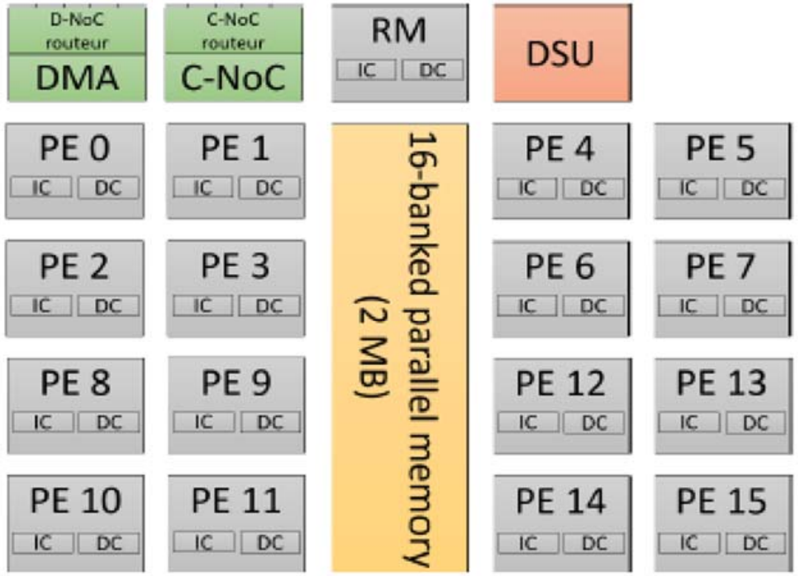


FIGURE 2.1: Kalray's compute cluster

- 288-core single chip
  - 16 17-core compute clusters
  - 4 I/O subsystems (2D torus)
- Each cluster includes 17 cores
  - 16 for general-purpose computing
  - 1 for communication and core scheduling ops
- 2MB RAM per cluster, in 16 128KB-memory banks, grouped pairwise for 8 core pairs
  - Divided in left-side and right-side bank
  - Memory address mapping interleaved or blocked

NodeOS

- Kalray's lightweight POSIX-API runtime for *thread-level parallelism* in compute clusters
  - Asymmetric: the resource manager core processes all (synchronous, blocking) kernel calls (proxied from the compute cores, and FIFO queued) and services the NoC interfaces
- One thread per core, one process per cluster
  - Neither process- nor thread-level scheduling (no preemption and migration) are supposed to occur

Pthreads unfit for parallelism /1

- The POSIX primitives perform multiple data cache invalidations and write-buffer purging ops to assure cache coherency across cores
- Too memory-heavy for embedded parallelism (mostly owing to the execution stack)
  - The context switch overhead off preemptive scheduling annuls the parallel speed-up
- Pthreads can only be static placeholders pinned to cores, serving *tasks*, i.e. parallel opportunities

Pthreads unfit for parallelism /2

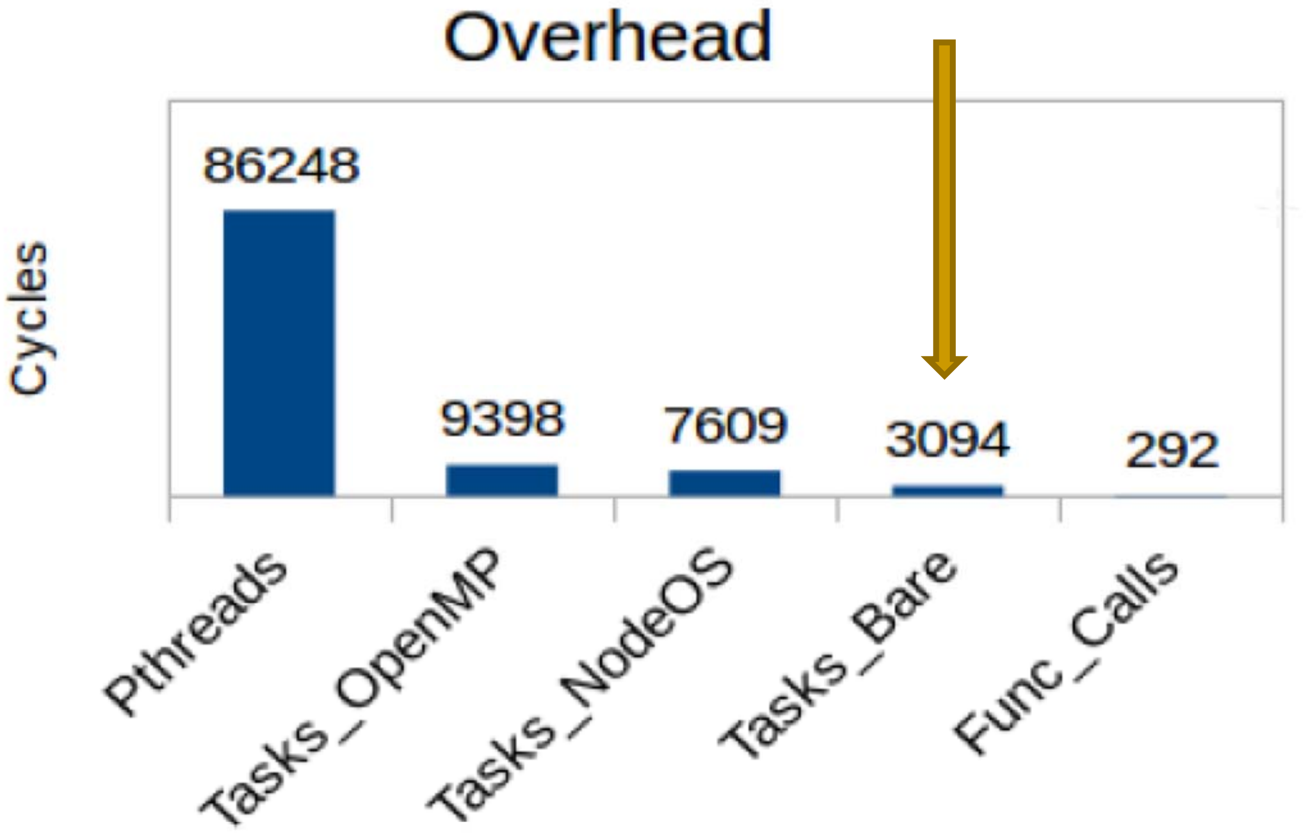


FIGURE 2.9: Pthreads and lightweight tasks overhead comparison

Our runtime library /1

- An execution model that supports *lightweight tasks* to allow exposing the potential parallelism of applications *efficiently*
- An application-level runtime environment that implements dynamic, load-balanced task scheduling on top of threads
- Applications seen as DAGs

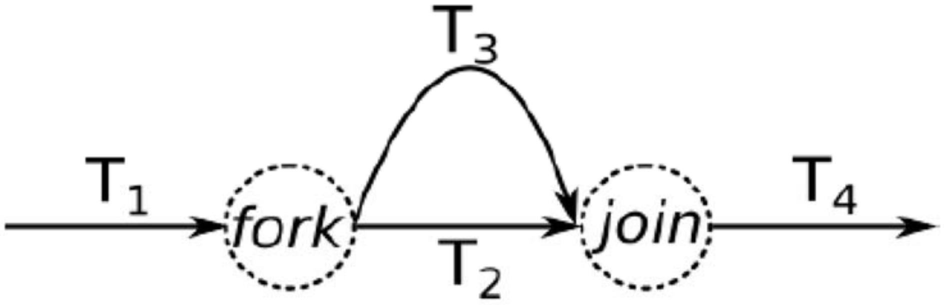


FIGURE 2.2: A fork/join DAG

Runtime architecture

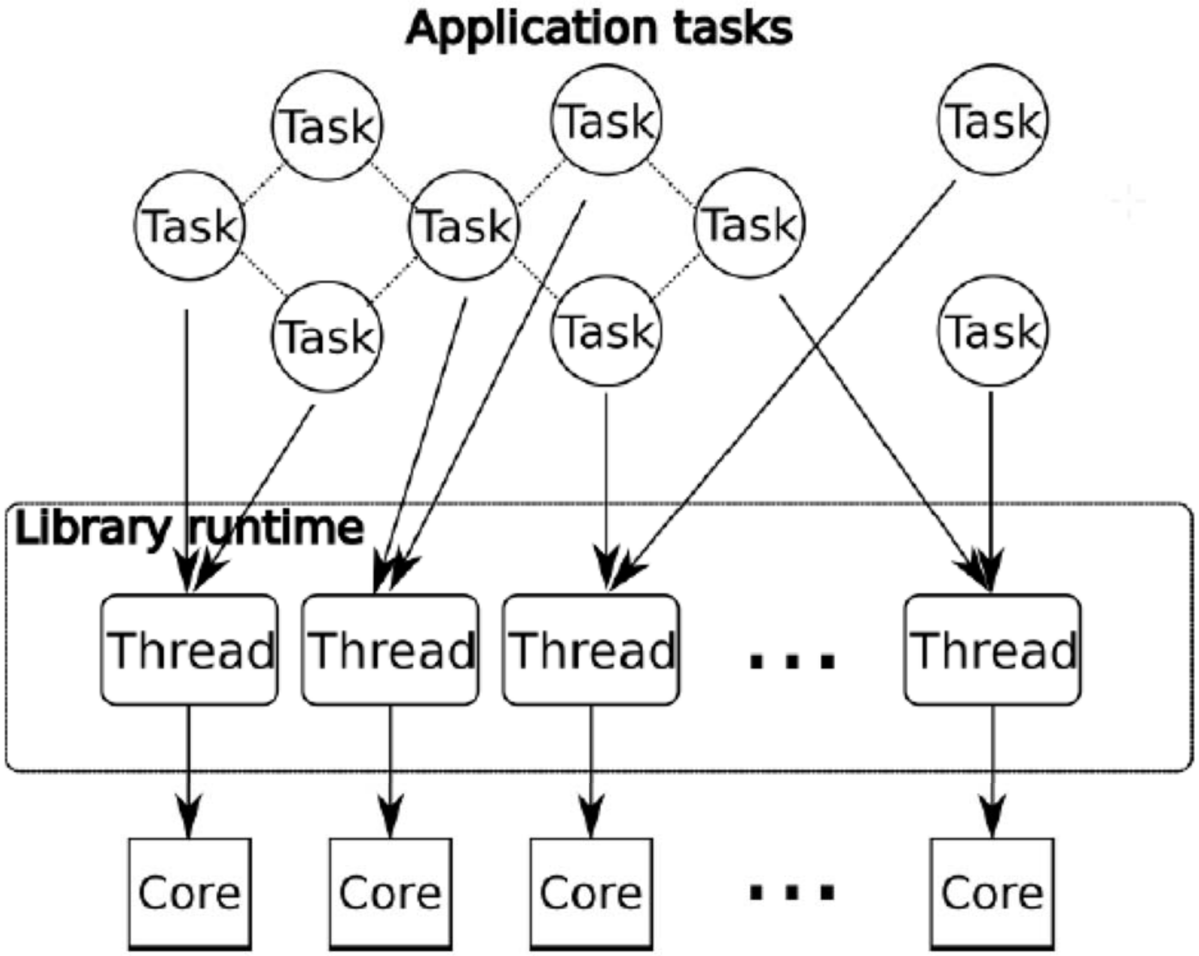


FIGURE 2.3: Execution model

Our runtime library /2

- DAGs model parallel computation
  - Edges denote sequential strands of computation
  - Nodes denote fork and join operations
- Suspension is costly and *should be avoided*
  - Invert control-flow dependencies and convert the program to a *continuation-passing style*
- The computation always makes progress performing a *tail-recursive* function call
  - No return to the caller, but to a “continuation” that represents the remainder of the computation

Continuations /1

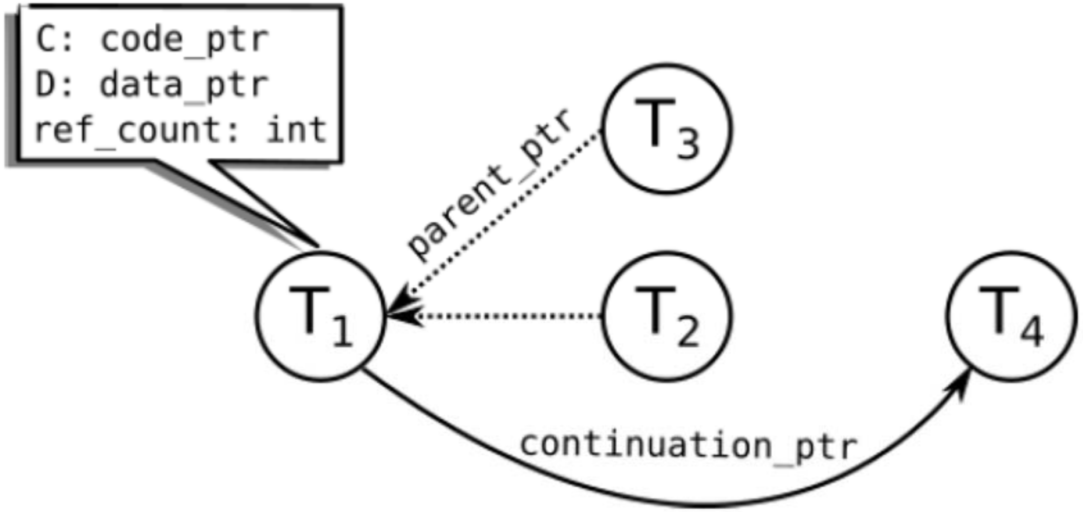


FIGURE 2.5: A task-based implementation of fork/join parallelism

- The completion of T2 **and** T3 triggers the execution of T4 (their continuation)
- The continuation task T4 is seen as part of T1
  - And it inherits T1's possible ancestor
  - Children tasks return to their parent effectively by sending return values to the continuation

Continuations /2

- Tasks never suspend
  - Their execution is deferred before starting
- This does away with the nesting of stack frames, and makes the execution of tasks completely *asynchronous*
- This model needs a task pool that stores the tasks that need execution, which neatly allows for *load balancing*

Execution model /1

- Tasks run to completion
  - Hence, there are no blocking, yielding, suspension, or other interfering events
  - Much benefit on temporal and spatial locality
- The runtime is stack-less
  - All tasks that execute within the context of the same executor may share its stack
- The runtime complexity is minimum

Execution model /2

- The schedule loop exits when all tasks have been executed
  - But checking whether the task pool is empty *may not be sufficient*
  - Residual tasks may be still executing with an empty task pool and they can (still) originate a further subtree of tasks
- We check completion of the root of DAG
  - Its completion corresponds to the termination of the computation

Load balancing /1

- *Work-sharing* is work-conserving
  - No executor can be idle as there are ready tasks
- But it is not very efficient to implement
  - The *push mode* feeds one executor at a time
  - The *pull model* requires queue locking, which serializes scheduling decisions and becomes a scalability bottleneck
- It simply does not scale

Load balancing /2

- *Work-stealing* uses a dequeue per executor
  - Double-ended local LIFO queue
  - Pushing and popping on the tail (serialized)
- When the local pool becomes empty, the executor steals from a victim
  - Stealing removes the task at head of the victim's deque (FIFO, to minimize access conflicts)
  - Random victim selection propagates work well
- Lesser contention among cores, more data locality, better load balancing

Load balancing /3

- *Work-seeking* uses cooperative distribution of work between busy and idle executors
  - When the executor empties its local queue, it seeks work from busy executors
  - Busy executors regularly check for work-seeking executors and, when they find one, they *synchronously push* a task into their queue
- The idle executor suspends on empty (local) queue and resumes as soon as the queue is no longer empty

Which is best?