# INFO-F404, Real-Time Operating Systems

Joël Goossens

Université libre de Bruxelles

# A) Parallel algorithms

# Introduction I

- In this introduction, we will see that there are limits to parallelizing programs (regardless of whether we have an unlimited amount of processors at our disposal).
- Parallelizing comes at a cost due to overheads related to synchronization, message exchanges, etc.
- We'll begin by defining the concept of **speedup factor**:

## Definition 1 (Speedup factor)

The speedup factor $S(n)$ is a measure of the **relative** performance difference between the sequential and parallel versions of a program. We define it as follows:

$$S(n) = \frac{t_s}{t_p}$$

where $t_s$ denotes the execution time of the uniprocessor (sequential) version and $t_p$ denotes the execution time of the multiprocessor (parallel) version using $n$ processors.

# Introduction II

- When comparing sequential and parallel versions of an algorithm, we usually consider the fastest known sequential algorithm.
- The maximum speedup factor that can be gained using $n$ processors is $n$ (**linear speedup** principle).
- If the speedup factor would be greater than $n$, it would mean that the parallel algorithm could be "simulated" on a single processor, contradicting the assumption that the original sequential algorithm was the fastest known.

# Cost of parallelizing

- Parallelizing has several costs which we must consider:
  - Instants when not all processors are busy. This includes instants where only **one** processor is busy due to the intrinsic sequential nature of a given step or part of the problem.
  - Additional computations in the parallel version.
  - Communication delays.
  - Parallel algorithms are harder to design.
  - etc.

# Maximum speedup — AMDAHL's law

- Let $f$ be the fraction of sequential computations of a parallel program. $(1 - f)$ is thus the fraction of computations which can be parallelized. Then, the total computation time required by the program (assuming $n$ processors are used) is $f \cdot t_s + (1 - f) \cdot \frac{t_s}{n}$.
- Therefore, the speedup factor is:

$$S(n) = \frac{t_s}{f \cdot t_s + (1 - f) \cdot \frac{t_s}{n}} = \frac{n}{1 + (n - 1) \cdot f}$$

- This equation is known as AMDAHL's law (AMDAHL, 1967).
- A corollary of the law is that the **speedup factor has an upper bound**, even with an unlimited number of processors:

$$\lim_{n \to \infty} S(n) = \frac{1}{f}$$

- For example, if only 5% of the computations are sequential, the maximum speedup factor we can hope for is **20**, regardless of the number of processors.

# *Rank Sort*, sequential version

- During a **Rank Sort**, we compute the number of items smaller than each item. This number is then the position (**rank**) of the given item in the sorted array.
- The sequential version of this algorithm has a time complexity of $O(n^2)$, making it a poor sequential sorting algorithm:

```
for (i=0; i<n; i++){ //for each number
  x = 0;
  for (j=0; j<n; j++)
    if (a[i]>a[j]) x++;
  b[x] = a[i];
}
```

- The algorithm above assumes the array only holds unique values. Handling of duplicate values is left as an exercise.

# *Rank Sort* with $n$ processors

- Each processor can determine the rank of a single item in $O(n)$ time:

```
for (i=0; i<n; i++){ //for each number in parallel
  x = 0;
  for (j=0; j<n; j++)
    if (a[i]>a[j]) x++;
  b[x] = a[i];
}
```

- The parallel version thus has a time complexity of $O(n)$, which is better than any sequential comparison sort.

# Bitonic sort I

These slides are based on reference [4].

## Definition 2 (Bitonic sequence)

A sequence $(a_1, a_2, \ldots, a_{2k})$ is said to be **bitonic** iff:

- Either there is an integer $j, 1 \leqslant j \leqslant 2k$, such that

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_j \geqslant a_{j+1} \geqslant a_{j+2} \geqslant a_{2k}$$

- Or the sequence does not initially satisfy the previous condition, but can be shifted cyclically until the condition **is** satisfied.

## Definition 3 (Compare-and-swap)

The compare-and-swap instruction receives a distinct pair of elements as input and produces an ordered pair (in either increasing or decreasing order) as output in one time unit (will be used in time complexity analysis).

# Bitonic sort II

## Example 4

The following sequence: 3 5 8 9 7 4 2 1 is bitonic.

We assume that the length of the sequence to be sorted is an integer power of 2.

## Property 5

Let $(a_1, a_2, \ldots, a_{2k})$ be a bitonic sequence, and let $d_i = \min(a_i, a_{k+i})$ and $e_i = \max(a_i, a_{k+i})$, for $i = 1, 2, \ldots, k$. The following properties hold:

- The sequences $(d_1, d_2, \ldots, d_k)$ and $(e_1, e_2, \ldots, e_k)$ are both bitonic
- $\max(d_1, d_2, \ldots, d_k) \leqslant \min(e_1, e_2, \ldots, e_k)$

# Sorting a Bitonic Sequence

Given a bitonic sequence $(a_1, a_2, \ldots, a_{2k})$, it can be sorted into a sequence $(c_1, c_2, \ldots, c_{2k})$, arranged in nondecreasing order, by the following algorithm $\text{Merge}_{2k}$ using nondecreasing compare-and-swaps:

Step 1 The two sequences $(d_1, d_2, \ldots, d_k)$ and $(e_1, e_2, \ldots, e_k)$ are produced.

Step 2 These two bitonic sequences are sorted independently and recursively, each by a call to $\text{Merge}_k$.

Step 3 Sorted sequences $(d'_1, \ldots, d'_k)$ and $(e'_1, \ldots, e'_k)$ are concatenated and returned.

Remarks:

- the two sequences can be sorted independently since no element of $(d_1, d_2, \ldots, d_k)$ is larger than any element of $(e_1, e_2, \ldots, e_k)$;
- the recursion terminates when $k = 2$, using a single compare-and-swap.

# Sorting an Arbitrary Sequence

Algorithm $\text{Merge}_{2k}$ assumes that the input sequence is bitonic.
Given an arbitrary sequence $(a_1, a_2, \ldots, a_n)$, consider the $n/2$ compare-and-swaps for the pairs $(a_1, a_2), (a_3, a_4), \ldots, (a_{n-1}, a_n)$. For odd-numbered compare-and-swaps, place the smallest value first. For even-numbered compare-and-swaps, place the largest value first.
At the end of the first stage, $n/4$ bitonic sequences of length 4 are obtained. These sequences can be sorted using $\text{Merge}_4$. Odd-numbered instances get sorted by nondecreasing order, whereas even-numbered instances get sorted by nonincreasing order. This yields $n/8$ bitonic sequences, each of length 8.
The process repeats until a single bitonic sequence of length $n$ is obtained.

# Computational Complexity of the Bitonic Sort

Considering $2k = 2^i$, the time to complete $\text{Merge}_{2k}$ is given by the following recurrence:

$$\begin{aligned} d(2) &= 1 \\ d(2^i) &= 1 + d(2^{i-1}) \end{aligned}$$

whose solution is $d(2^i) = i$. Consequently, the total computational complexity is:

$$\sum_{i=1}^{\log n} d(2^i) = \sum_{i=1}^{\log n} i = \frac{(1 + \log n) \log n}{2}$$

- The computational complexity of the Bitonic Sort is $\mathcal{O}((\log n)^2)$.
- The maximal number of compare-and-swaps required is $\mathcal{O}(n(\log n)^2)$ (see [4] for details).

# Parallelizing schedulability tests

- We will now study parallelizing of a tool we introduced during the lectures on real-time scheduling.
- The vast majority of schedulability tests work by **simulating** the system in a feasibility interval and checking whether any deadline is missed.
- However, simulation is mostly a sequential notion.
- We will attempt to identify conditions under which a simulation may be parallelized.
- These conditions (work-conservation, determinism and partial memorylessness) are quite general and hold for most popular uniprocessor scheduling algorithms.

# Assumptions

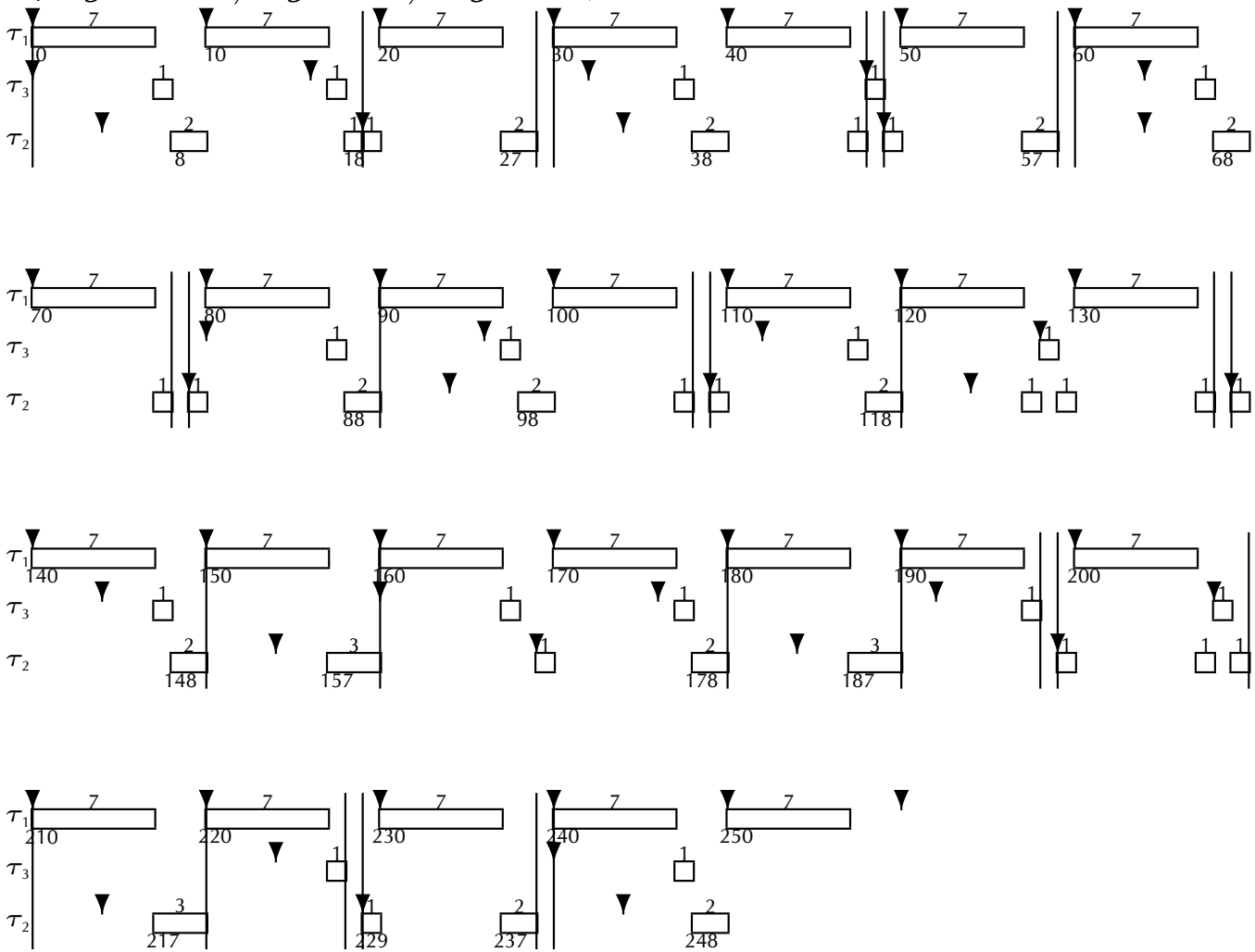
- We consider **work-conserving** schedulers. This implies that the processor is never left idle as long as jobs are ready to run.
- We consider **uniprocessor** schedulers.
- We consider **deterministic** schedulers. Given an arbitrary set of jobs, the algorithm generates a **unique** schedule.

# Idle instants

- As a reminder, $x \in \mathbb{N}$ is an idle instant if all jobs that arrived strictly before instant $x$ are completed before or at instant $x$.
- We consider **partially memoryless** schedulers. Let $i_t$ denote the latest idle instant strictly before instant $t$. A scheduler is partially memoryless if the decisions taken at instant $t$ only depend on the schedule for instants in $[i_t, t)$ and the system state at instant $t$.
- Let $[t_0, t_k)$ be the feasibility interval for a given simulation and let $t_1, t_2, \ldots, t_{k-1}$ be distinct idle instants in the schedule. If the scheduler is deterministic, partially memoryless and work-conserving, then each of the $k$ intervals $[t_0, t_1)$, $[t_1, t_2)$, $\ldots$, $[t_{k-1}, t_k)$ can be simulated in **parallel**.

# Example

3 periodic tasks ($T_1 = 10, C_1 = 7, O_1 = 0; T_2 = 15, C_2 = 1, O_2 = 4; T_3 = 16, C_3 = 3, O_3 = 0$)



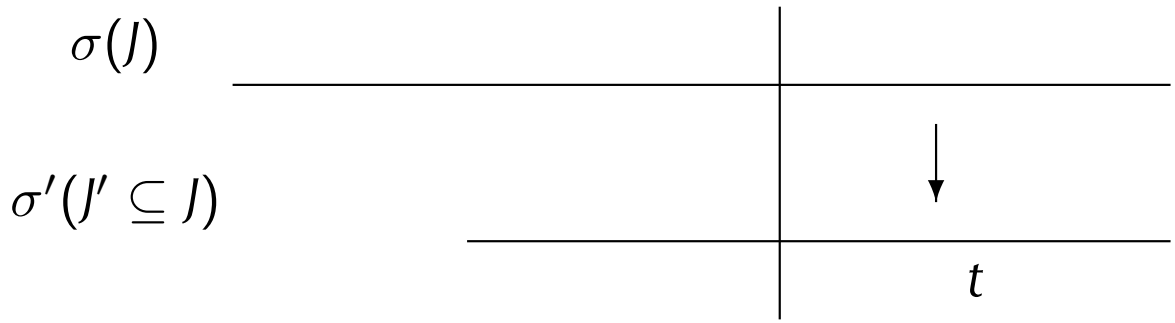
# Problem

- It's easy to determine idle instants for work-conserving schedulers but in a **sequential** manner.
- As such, how can we hope to speedup execution through parallelization?

## Lemma 6

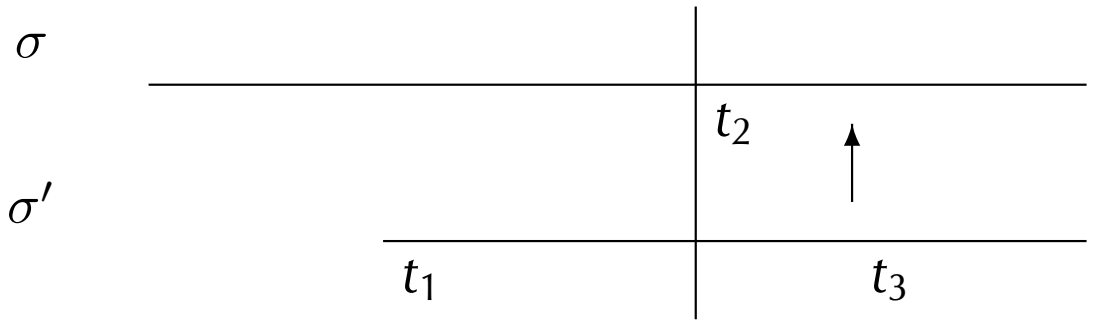
*Let J be an arbitrary set of jobs and let $\sigma$ be the schedule produced by a scheduler A. Let us assume that there is an idle instant t in $\sigma$. Let $J'$ be a subset of J ($J' \subseteq J$) and let $\sigma'$ be the schedule obtained with an arbitrary but work-conserving scheduler B. Then, instant t is also idle in $\sigma'$.*

(Proof: see uniprocessor chapter.)



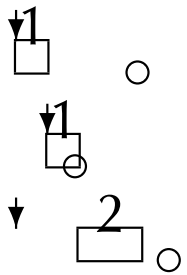
## Lemma 7 ([1])

*Let $J$ be an arbitrary set of jobs and let $\sigma$ be the schedule produced by a scheduler $A$. Let $\sigma'$ be the schedule produced by the same scheduler but by only considering jobs arriving at or after instant $t_1$. Let $t_2$ be the first idle instant strictly after $t_1$ in schedule $\sigma$. If a deadline is missed at instant $t_3$ in $\sigma'$ where $t_3 \geqslant t_2$, then a deadline is missed in the initial schedule $\sigma$ at instant $t_3$. Thus, system $J$ is not schedulable.*



# Reasonable schedulers I

- If a scheduler that successfully schedules a set of jobs $J$ can also successfully schedule any subset of $J$, the scheduler is said to be **reasonable**.
- We've seen that popular uniprocessor schedulers are work-conserving and deterministic. It can also be shown that they are reasonable.
- Do note that some schedulers may not be. For example, non-preemptive schedulers are usually not reasonable:

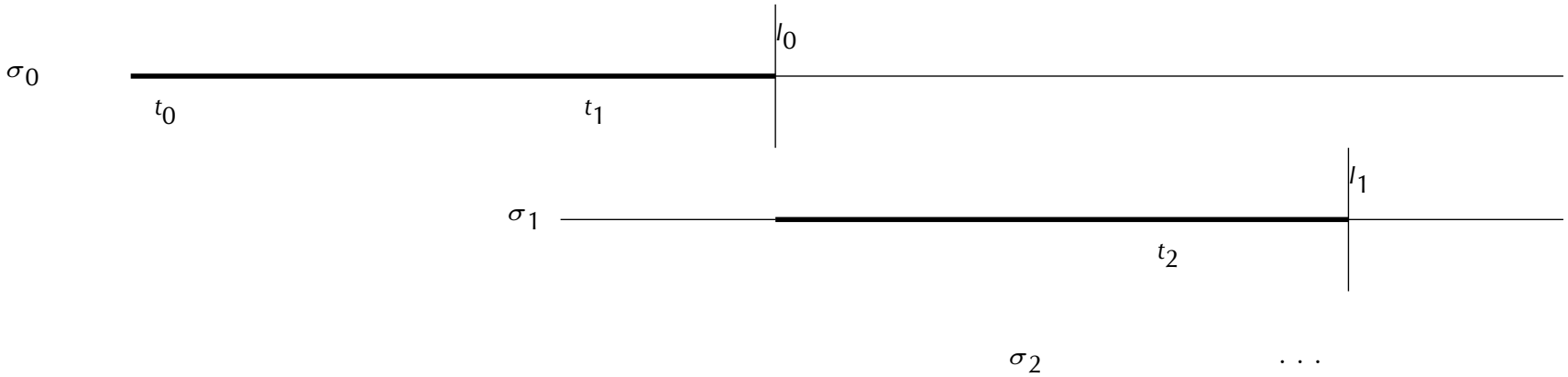



- For reasonable schedulers, we have a stronger property than Lemma 7:

  **If a deadline is missed in $\sigma'$, then a deadline is missed in the initial schedule $\sigma$. The system is thus unschedulable.**

# Lemma 7 in practice

- Lemma 7 will allow us to parallelize our simulation on feasibility interval $[t_0, t_k)$ over $k$ processors for deterministic, work-conserving and partially memoryless schedulers. They need not necessarily be reasonable.
- Let's illustrate the main idea:



# More formally

- **Theorem 8**
  *Let $J$ be an arbitrary set of jobs and let $\sigma$ be the schedule produced by a work-conserving, deterministic and partially memoryless scheduler. Let $t_0$ be the first instant where a job of $J$ appears in the system. Let $t_1, t_2, \ldots$ be instants such that $t_0 \leqslant t_1 \leqslant t_2 \leqslant \cdots$. Let $\sigma_j$ ($j \geqslant 0$) be the schedule obtained by only considering jobs of $J$ that arrive at or after instant $t_j$. Let $I_j$ ($j \geqslant 0$) be the first idle instant after instant $t_{j+1}$ in $\sigma_j$. Let $\lambda_j$ be defined as follows:*

$$\begin{aligned} \lambda_0 &= t_0 \\ \lambda_j &= \max_{0 \leqslant \ell < j} \{I_\ell\}, \quad \text{for } j > 0 \end{aligned}$$

*We then have that for all $j \geqslant 0$,*

$$t \geqslant \lambda_j \Rightarrow (\sigma_j(t) = \sigma(t))$$

# Parallelizing the simulation

- $k$ processors $\pi_0, \pi_1, \ldots, \pi_{k-1}$
- We pick $(t_0), t_1, \ldots, t_{k-1}, (t_k)$
- Processor $\pi_j$ $(0 \leqslant j < k)$ will simulate the system by only considering jobs arriving at or after instant $t_j$.
- The simulation on $\pi_j$ may terminate (for feasible systems) after having simulated an idle instant after $t_{j+1}$, which is instant $I_j$ by definition.
- The system is unfeasible if and only if $\exists \pi_j \cdot 0 \leqslant j < k$ that simulates a deadline miss in interval $[\lambda_j, I_j)$.
- Note that if the scheduler is not reasonable and that we miss a deadline on processor $\pi_j$ before instant $\lambda_j$, we cannot conclude that the system is unfeasible!
- $\pi_j$ $(j > 0)$ does not initially know $\lambda_j$. Therefore, $\pi_j$ must keep track of the **last** missed deadline (if there is one at all).

# Pseudo-algorithm [2] I

1. *Initially, the values of $t_j$ and $t_{j+1}$ are communicated to processor $\pi_j$*
2. *In addition, the parameters of all jobs of the real-time system that are generated at or after instant $t_j$ must be communicated to processor $\pi_j$ – note that for many task models (such as periodic tasks) this information may be determined on line by $\pi_j$ during its generation of simulation $\gamma_j$, if the parameters of the periodic tasks that comprise the real-time system are communicated to $\pi_j$ beforehand.*
3. *$\pi_j$ simulates the behavior of the (deterministic, partially memoryless and work-conserving) scheduling algorithm on its set of jobs until an idle instant $I_j$ at or after $t_{j+1}$ is identified during the simulation. If feasibility-analysis is the goal, then if deadline misses occur during this simulation, $\pi_j$ records the time at which the latest deadline miss occurred but continues the simulation – the exact scheduling rule used for scheduling jobs that have missed deadlines is not important, other than that it continue to conform to the property of being work-conserving. If on-line admission control is the goal, then $\pi_j$ writes to the file $\mathcal{F}_j$ all the idle times identified during this simulation.*

# Pseudo-algorithm [2] II

4. *Processor $\pi_0$ knows the value of $\lambda_0$ beforehand (since $\lambda_0 = t_0$ by definition); for all $j > 0$, $\pi_j$ will receive the value of $\lambda_j$ from $\pi_{j-1}$.*
5. *Once $\pi_j$ knows $\lambda_j$ and $I_j$, it computes $\lambda_{j+1} = \{\lambda_j, I_j\}$, and communicates this value to $\pi_{j+1}$.*

# Parallelizing a more specific problem I

- Scheduling of asynchronous periodic tasks under constrained deadlines.
- A task $\tau_i$ is characterized by $T_i$ (its period), $D_i$ (its relative deadline), $C_i$ (its worst-case execution time) and $O_i$ (its offset).
- $[t_0 = \min\{O_i\}, t_k = \max\{O_i\} + 2 \cdot \text{lcm}\{T_i\})$ is a feasibility interval for EDF.
- We can prove the following property:

## Lemma 9

*$\pi_j$ ($j < k - 1$) finishes its simulation at instant $t_{j+1} + L^m$ at the latest, where $L^m$ is the length of the longest busy period.*

- $L^m$ corresponds to the first busy period in the synchronous configuration:

$$L^m = \sum_{i=1}^{n} \left\lceil \frac{L^m}{T_i} \right\rceil \times C_i,$$

- Research has shown that $L^m \ll P$ ([3])

# Parallelizing a more specific problem II

- Therefore, we can establish an upper bound on the time complexity of the parallel version:

## Theorem 10 ([1])

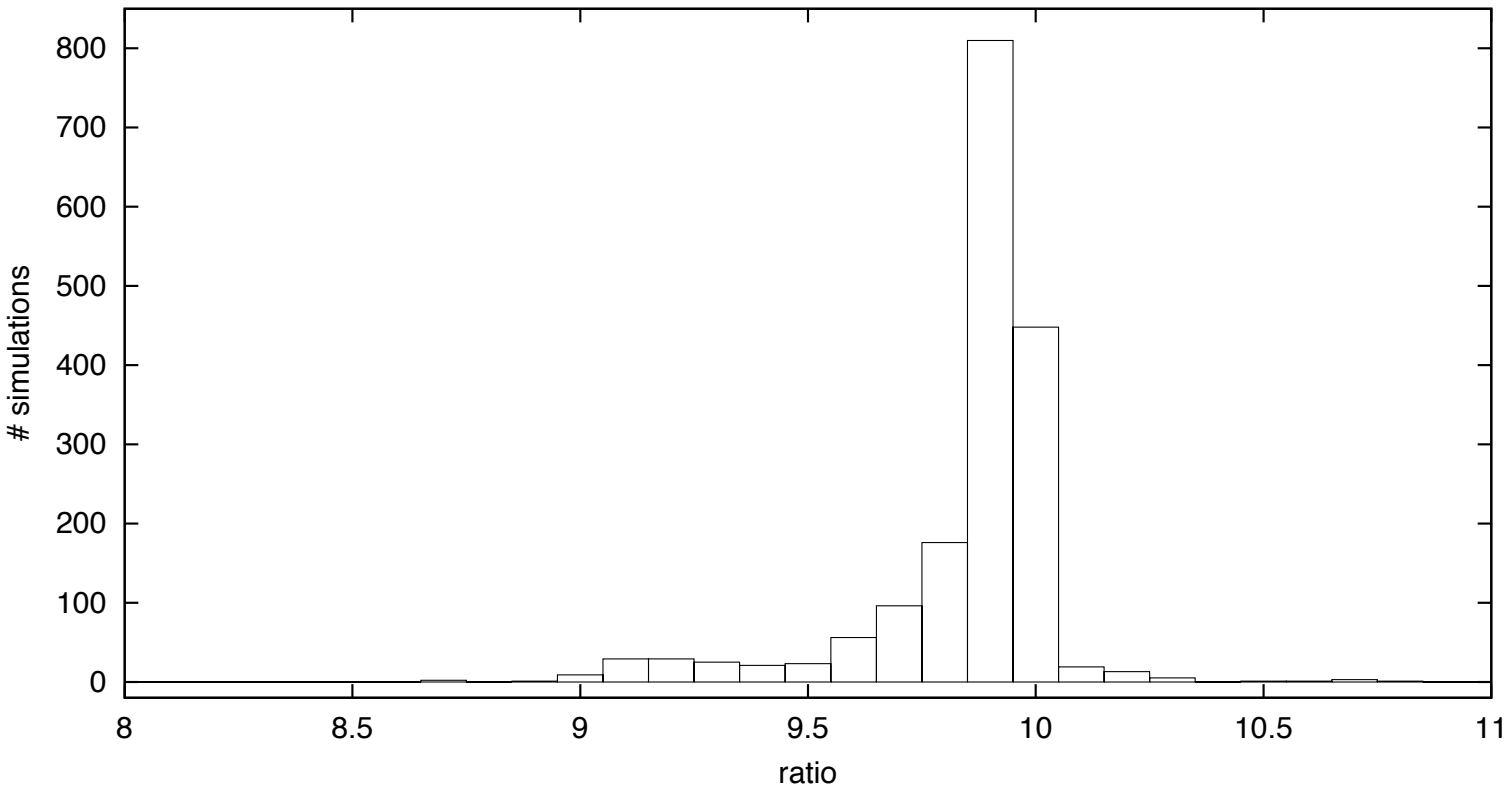
*The maximal time complexity of the parallel feasibility test for asynchronous periodic systems under constrained deadlines for any work-conserving, deterministic and partially memoryless scheduler is* $O\left(L^m + \max\left\{t_{j+1} - t_j \mid j = 0, \ldots, k-1\right\}\right)$

- The speedup factor yielded by parallelization is proportional to $\frac{P}{L^m}$.
- If the number of processors is bounded (i.e. is exactly $k$), the maximal time complexity becomes $O(\frac{P}{k} + L^m)$.

# Experimental evaluation

- Pseudo-random generation of asynchronous periodic task sets with constrained deadlines.
- $C_i \in [1, 20], D_i \in [2, 170], T_i \in [3, 670]$
- Order of magnitude of $P$ was $10^9$.

# Speedup distribution using 10 processors — $\frac{t_{\text{mono}}}{t_{\text{multi}}}$ ratio



# MPI algorithms for **reasonable** schedulers I

- Algorithm executed by processor $\pi_{k-1}$

```
Begin
        feasible ← Verification γ_{k-1}; {décrit supra}
        MPIBarrier();
        MPIIprobe();
        If (message(s))
        {
                MPIRecv();
                feasible ← false;
        }
        Return feasible;
End.
```

# MPI algorithms for **reasonable** schedulers II

- Algorithm executed by processor $\pi_i \quad (0 < i < k)$
  Begin
  Verification $\gamma_i$;
  `MPIBarrier().`; End.
- Verification algorithm $\gamma_j$
  Input: $n, T_i, C_i, D_i, O_i, t_j, t_{j+1}, j$

  Local variables:
  Natural $l_j$; {the first idle point after time $t_{j+1}$}
  Boolean *simulationfails*;
  Natural $t$; {the current time}

  Begin
  $t \leftarrow t_j$;
  *simulationfails* $\leftarrow$ false;
  $l_j \leftarrow t_k$;

# MPI algorithms for **reasonable** schedulers III

While $((\neg simulationfails) \wedge (t < I_j))$
{
    Schedule until the next event;
    {which implies the updating of $t$ and $I_j$}
    If (deadline miss)
    {
        $simulationfails \leftarrow true$;
        {stops the current simulation}
        `MPISend();` {stops all (running) simulations}
    }

# MPI algorithms for **reasonable** schedulers IV

```
                MPIIprobe();
                If (message(s))
                {
                        MPIRecv();
                        simulationfails ← true;
                        {stops the current simulation}
                }
        } {While}
        Return ¬simulationfails
End.
```

[1] Goossens, J., and Baruah, S.
Multiprocessor algorithms for uniprocessor feasibility analysis.
In *Proceedings of the seventh International Conference on Real-Time Computing Systems and Applications* (Cheju Island, (South Korea), December 2000), IEEE Computer Society Press. ISBN 0-7695-0930-0.

[2] Goossens, J., and Baruah, S. K.
Parallelizing uniprocessor schedule generation.
Technical report, 2002.

[3] Goossens, J., and Devillers, R.
Feasibility intervals for the deadline driven scheduler with arbitrary deadlines.
In *Proceedings of the sixth International Conference on Real-time Computing Systems and Applications* (Hong Kong, China, December 1999), IEEE Computer Society Press, pp. 54–61. ISBN 0-7695-0306-3.

[4] Padua, D., Ed.
*Encyclopedia of Parallel Computing*, vol. 1 A–D.
Springer, 2011.