Fault tolerance, malleability and migration for divide-and-conquer applications on the Grid

Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal

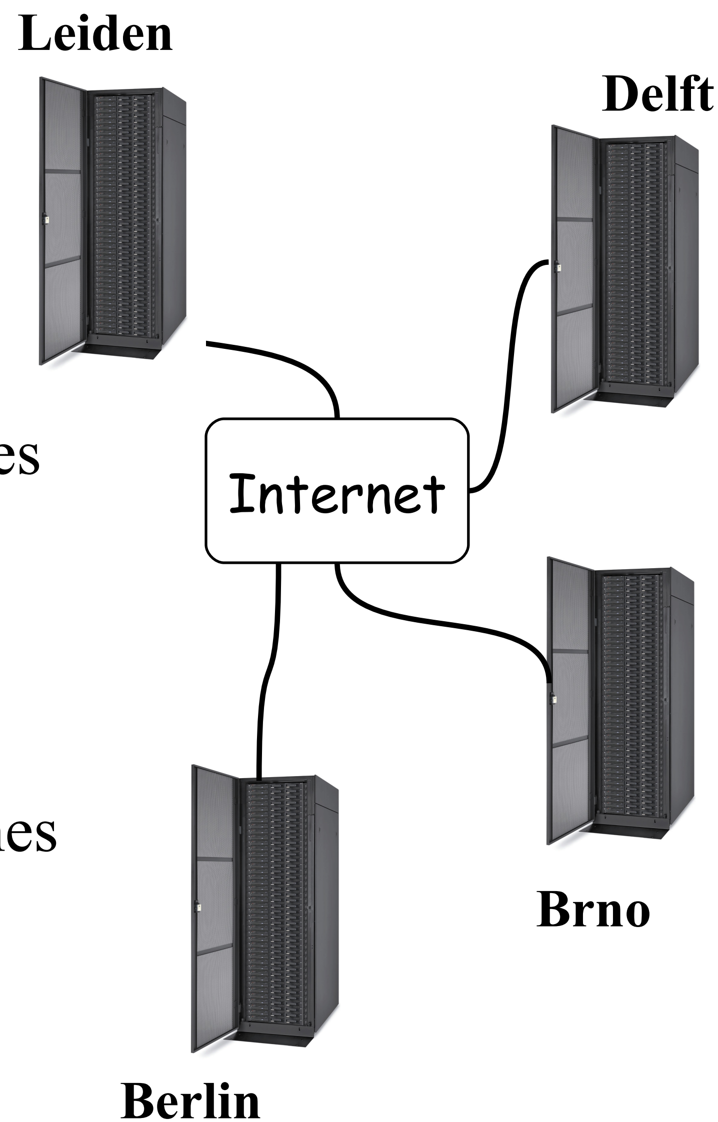
vl·e

vrije Universiteit

Ibis

Distributed supercomputing

- Parallel processing on geographically distributed computing systems (grids)
- Needed:
  - Fault-tolerance: survive node crashes (also entire clusters)
  - Malleability: add or remove machines at runtime
  - Migration: move a running application to another set of machines (comes for free with malleability)
- We focus on divide-and-conquer applications



Outline

- The **Ibis** grid programming environment
- **Satin**: a divide-and-conquer framework
- Fault-tolerance, malleability and migration in Satin
- Performance evaluation

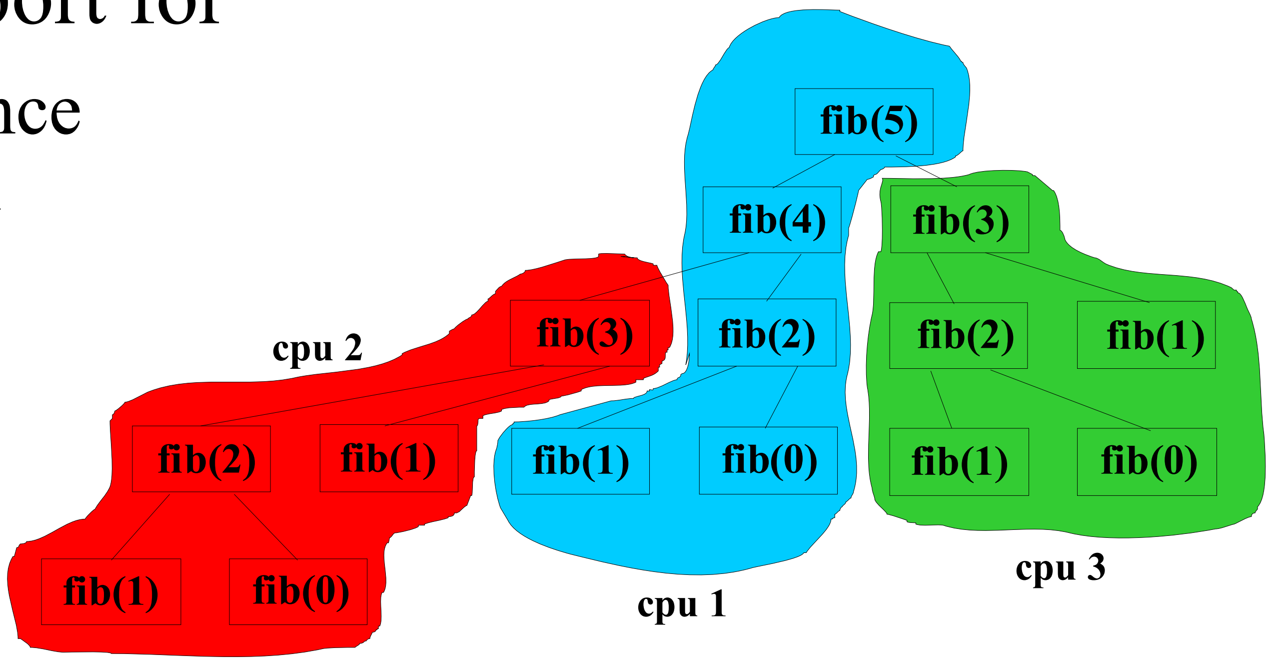
The Ibis system

- Java-centric => portability
  - „write once, run anywhere"
- Efficient communication
  - Efficient pure Java implementation
  - Optimized solutions for special cases with native code
- High level programming models:
  - Divide & Conquer (Satin)
  - Remote Method Invocation (RMI)
  - Replicated Method Invocation (RepMI)
  - Group Method Invocation (GMI)

http://www.cs.vu.nl/ibis/

Satin: divide-and-conquer on the Grid

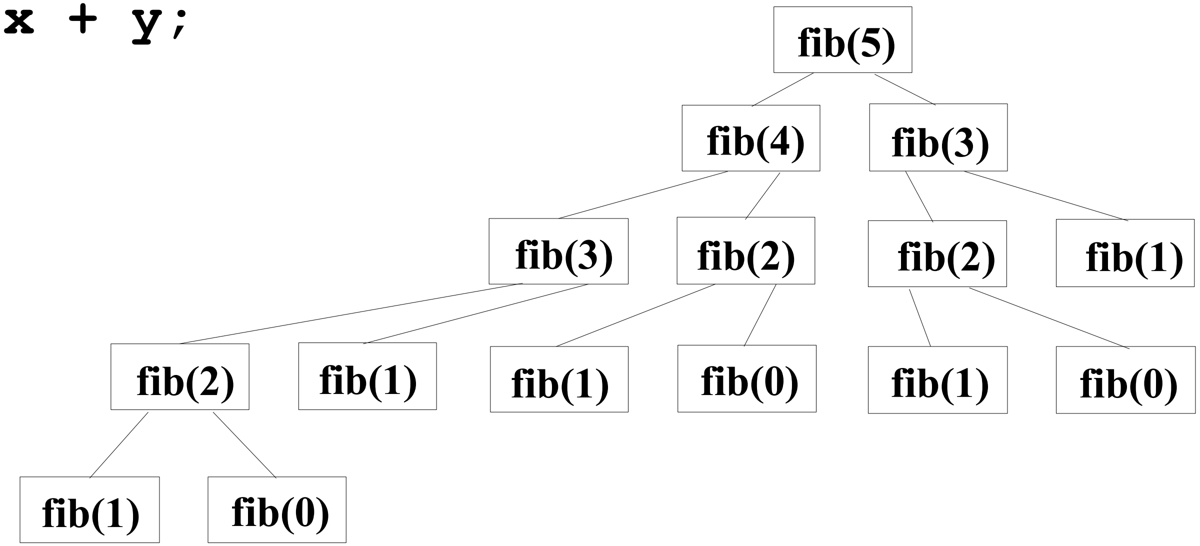
- Effective paradigm for Grid applications (hierarchical)
- Satin: Grid-friendly load balancing (aware of cluster hierarchy)
- Missing support for
  - Fault tolerance
  - Malleability
  - Migration



Example: Fibonacci

```
class Fib {
   int fib (int n) {
      if (n < 2) return n;
      int x = fib(n-1);
      int y = fib(n-2);
      return x + y;
   }
}
```

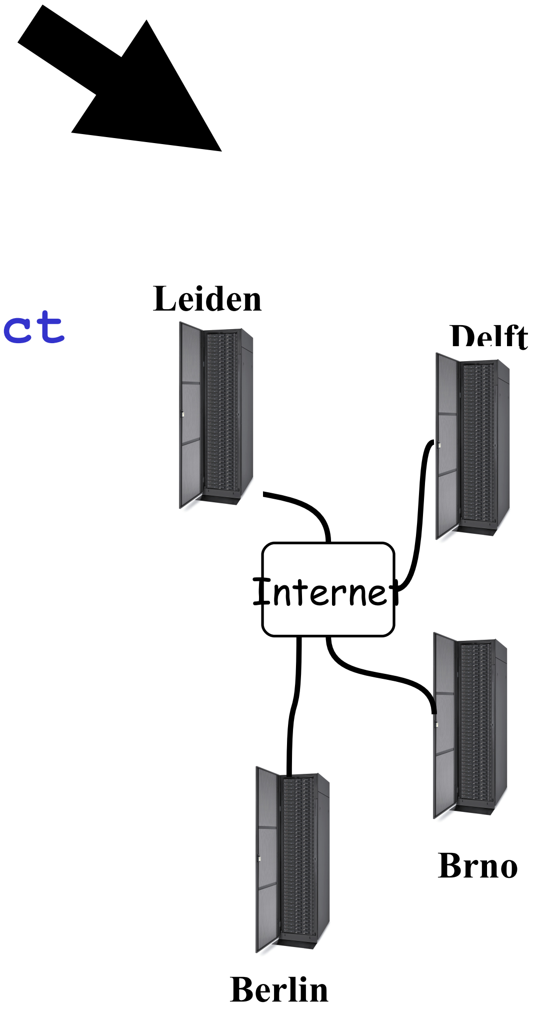
Single-threaded Java



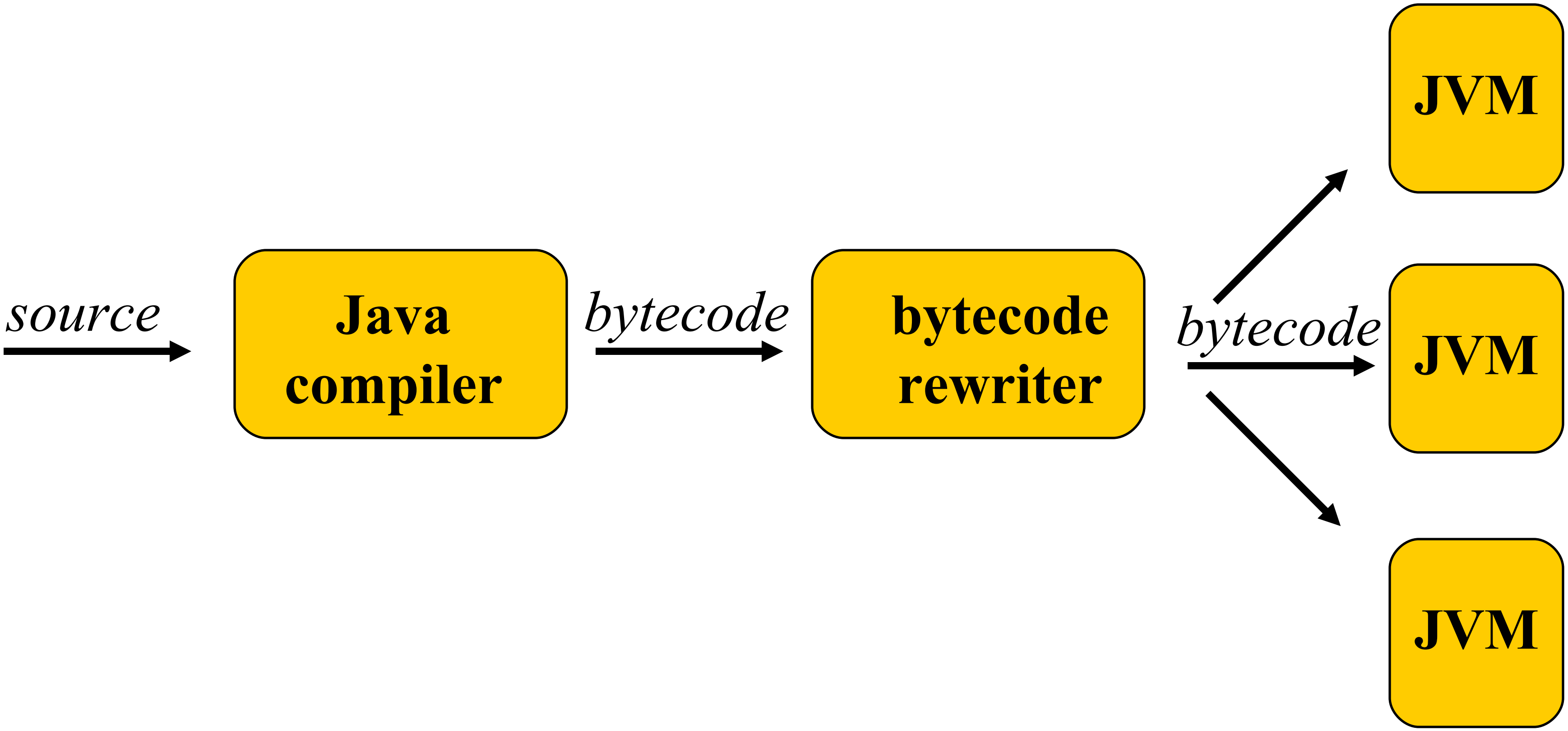
Example: Fibonacci

```
public interface FibInter extends
ibis.satin.Spawnable {
      public int fib (int n);
}

class Fib extends ibis.satin.SatinObject
implements FibInter {
   public int fib (int n) {
      if (n < 2) return n;
      int x = fib(n-1); /*spawned*/
      int y = fib(n-2); /*spawned*/
      sync();
      return x + y;
   }
}
```



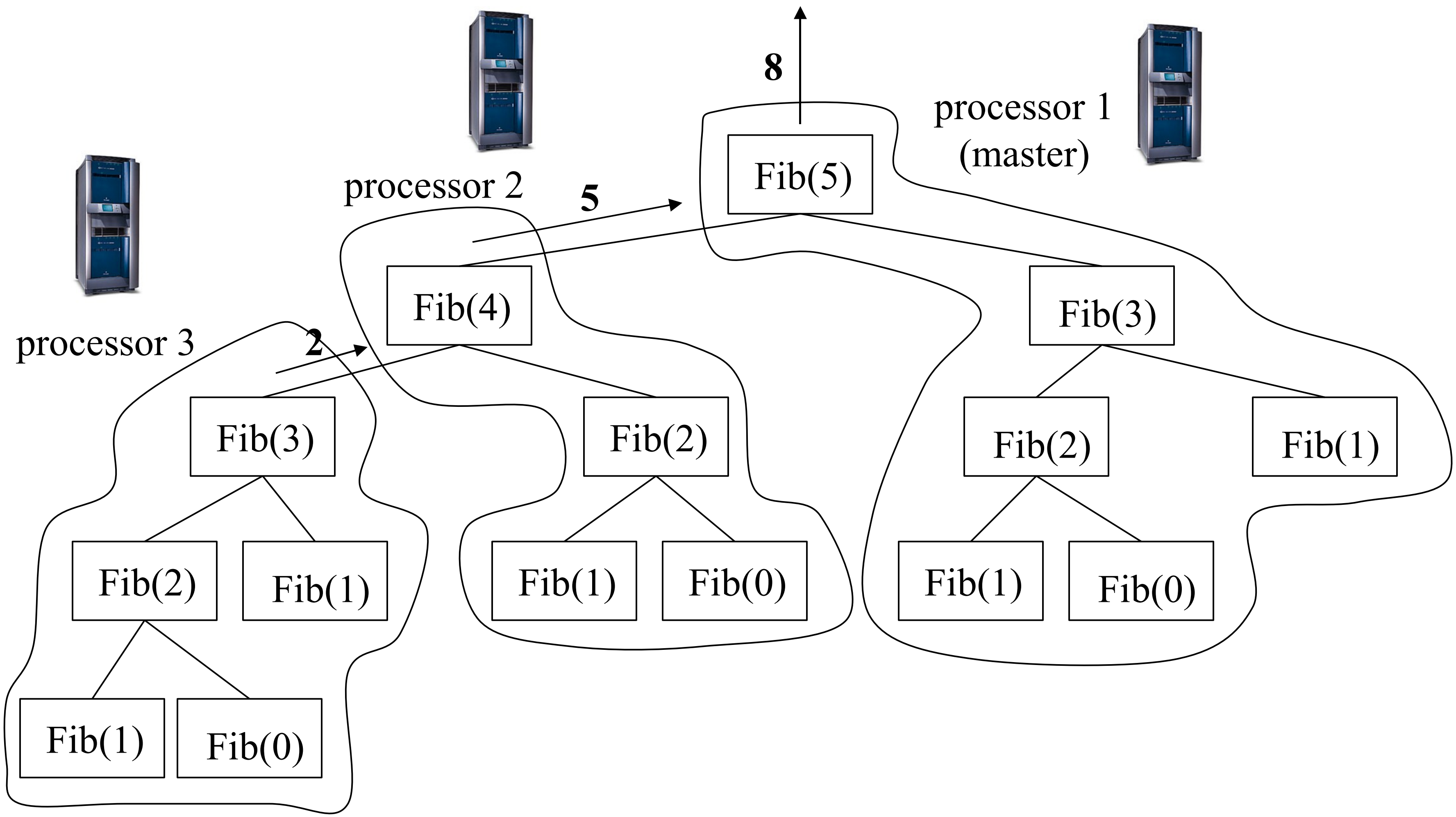
Compiling Satin programs



Executing Satin programs

- Spawn: put work in work queue
- Sync:
  - Run work from queue
  - If empty: steal (load balancing)

Example application: Fibonacci



Satin: load balancing for Grids

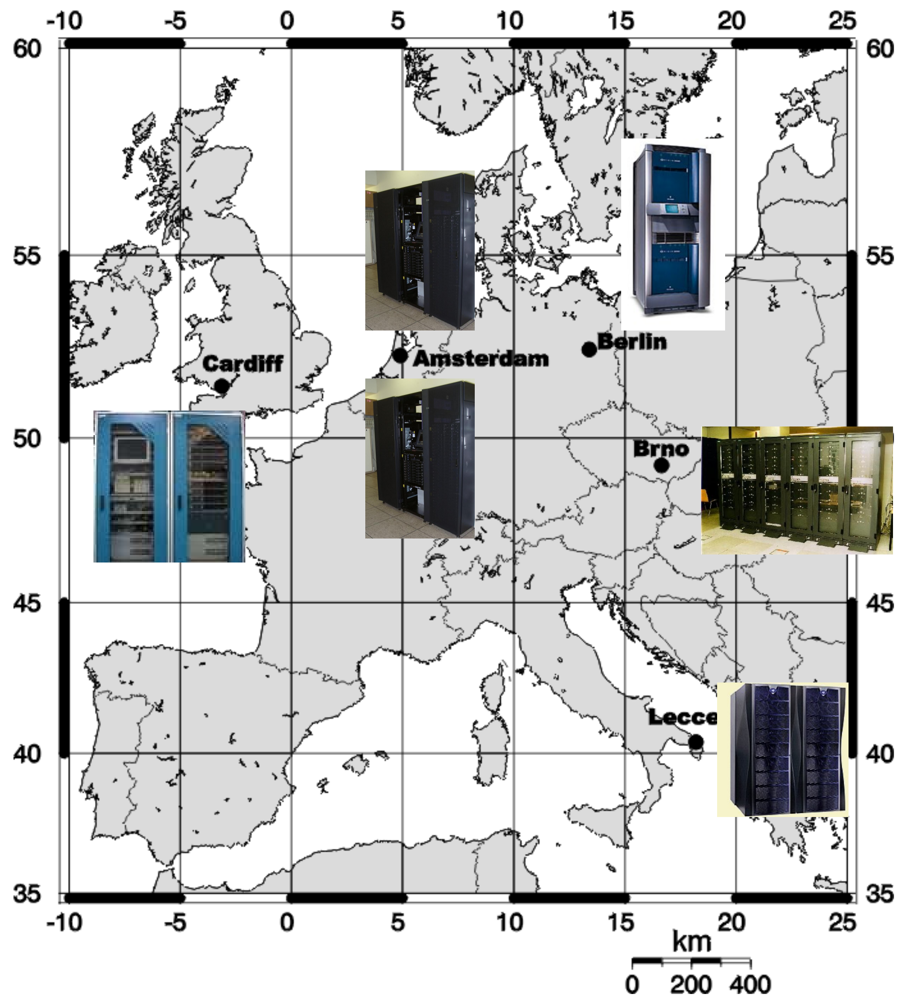
- Random Stealing (RS)
  - Pick a victim at random
  - Provably optimal on a single cluster (Cilk)
  - Problems on multiple clusters:
    - (C-1)/C % stealing over WAN
    - Synchronous protocol

Grid-aware load balancing

- Cluster-aware Random Stealing (CRS) [van Nieuwpoort et al., PPoPP 2001]
  - When idle:
    - Send asynchronous steal request to random node in different cluster
    - In the meantime steal locally (synchronously)
    - Only one wide-area steal request at a time

Satin raytracer on a grid

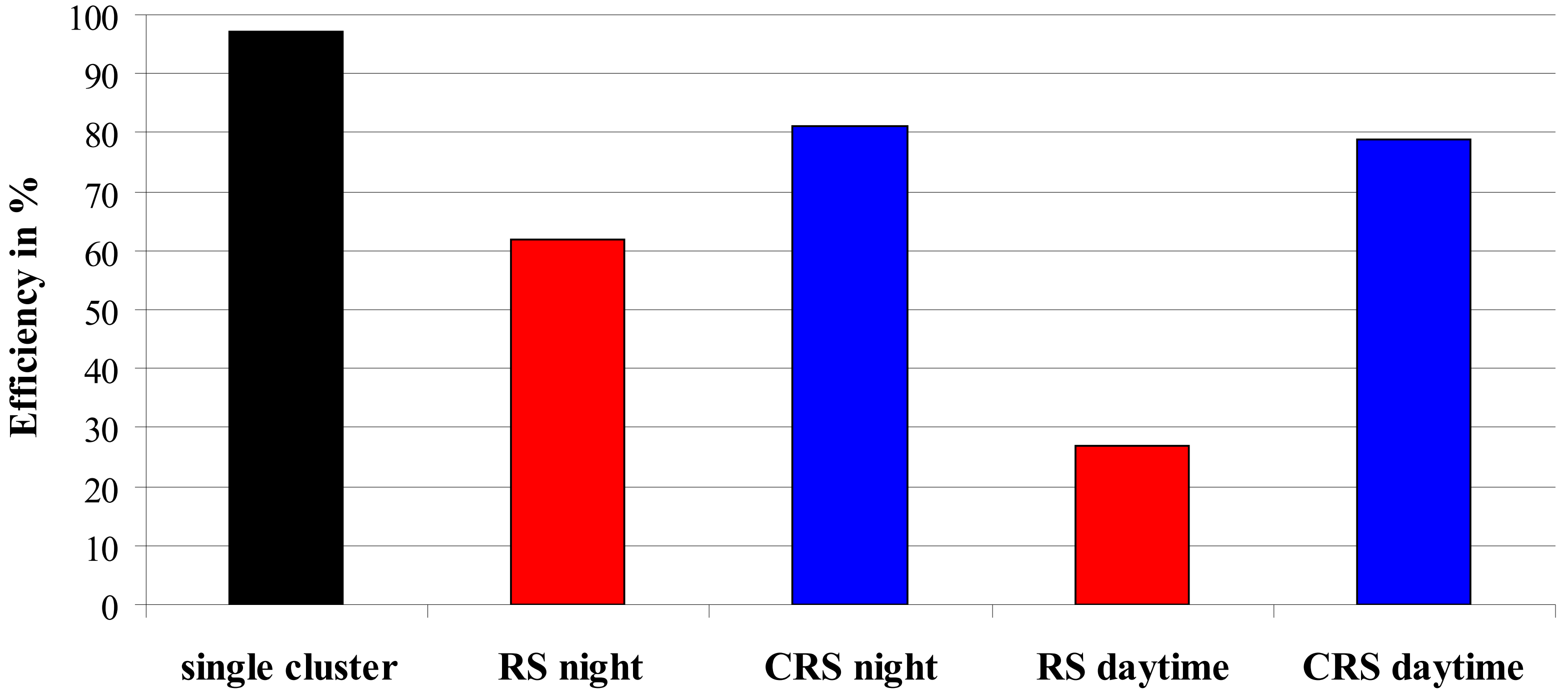
- GridLab testbed: 5 cities in Europe
- 40 cpus
- Distance up to 2000km
- Factor of 10 difference in CPU speeds
- Latencies:
  - 0.2 – 210 ms daytime
  - 0.2 – 66 ms night
- Bandwidth:
  - 9KB/s – 11MB/s
- Three orders of magnitude difference in communication speeds



Configuration

Location	Type	OS	CPU	CPUs
Amsterdam, The Netherlands	Cluster	Linux	Pentium-3	8 x 1
Amsterdam, The Netherlands	SMP	Solaris	Sparc	1 x 2
Brno, Czech Republic	Cluster	Linux	Xeon	4 x 2
Cardiff, Wales, UK	SMP	Linux	Pentium-3	1 x 2
ZIB Berlin, Germany	SMP	Irix	MIPS	1 x 16
Lecce, Italy	SMP	Tru64	Alpha	1 x 4

CRS performance on GridLab testbed



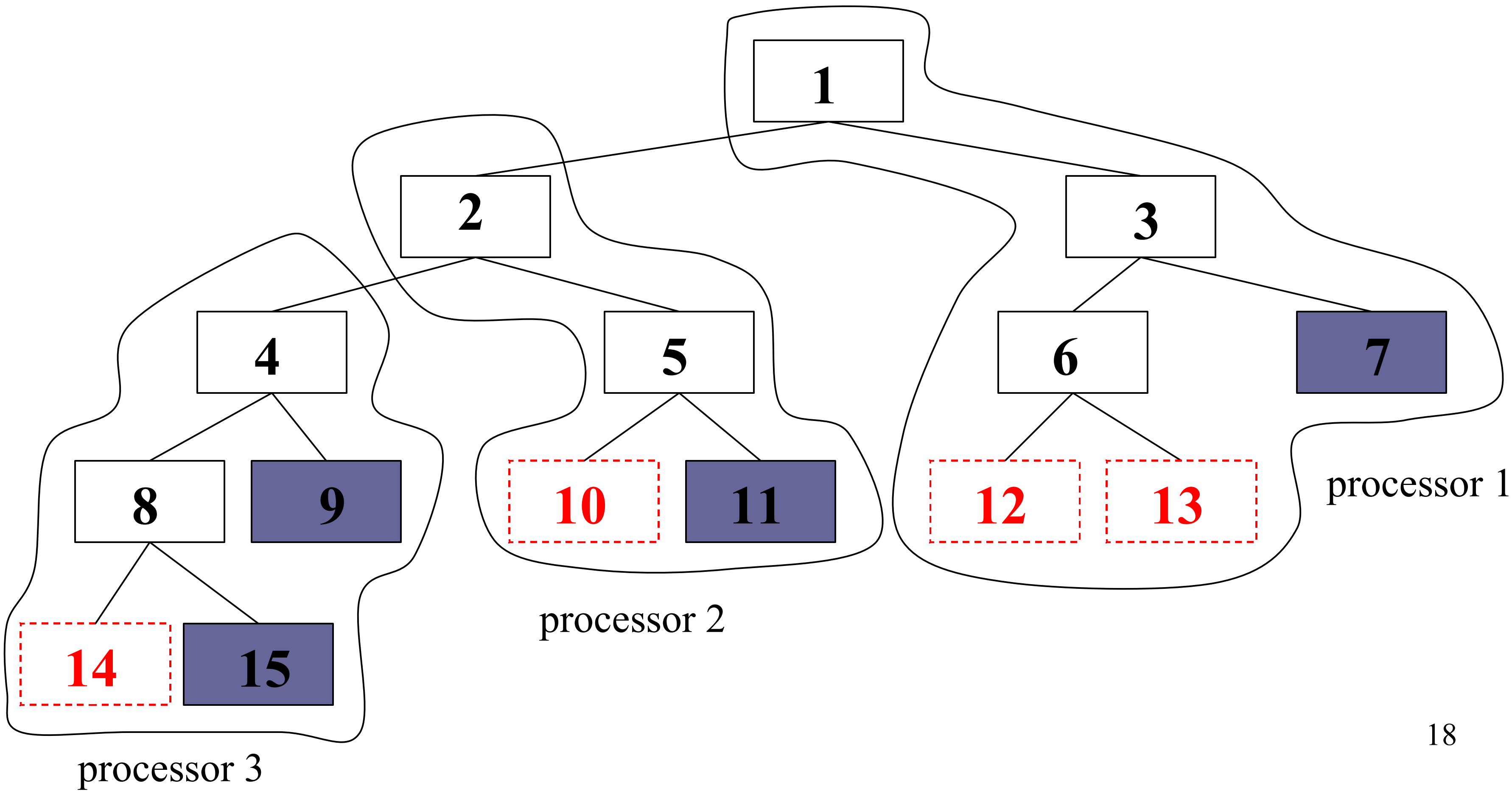
Satin summary

- Satin allows rapid development of parallel applications which are able to run with high efficiency in geographically distributed and highly heterogeneous environments
- Applications:
  - Barnes-Hut, Raytracer, SAT solver, TSP, Knapsack
  - All master-worker algorithms
- Still missing: FT, malleability, migration

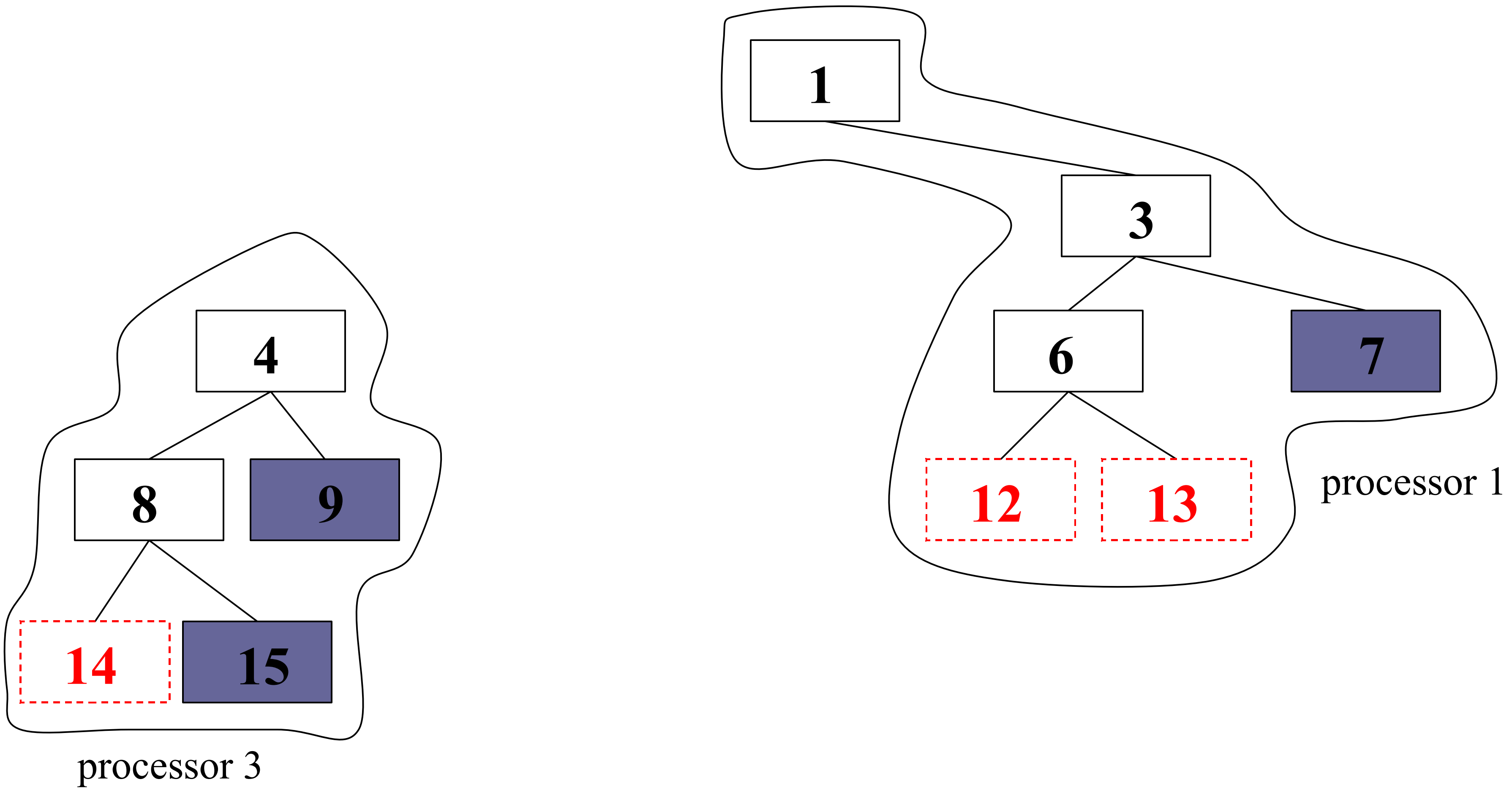
Fault-tolerance, malleability, migration

- Can be implemented by handling processors joining or leaving the ongoing computation
- Processors may leave either unexpectedly (crash) or gracefully
- Handling joining processors is trivial:
  - Let them start stealing jobs
- Handling leaving processors is harder:
  - Recompute missing jobs
  - Problems: orphan jobs, partial results from gracefully leaving processors

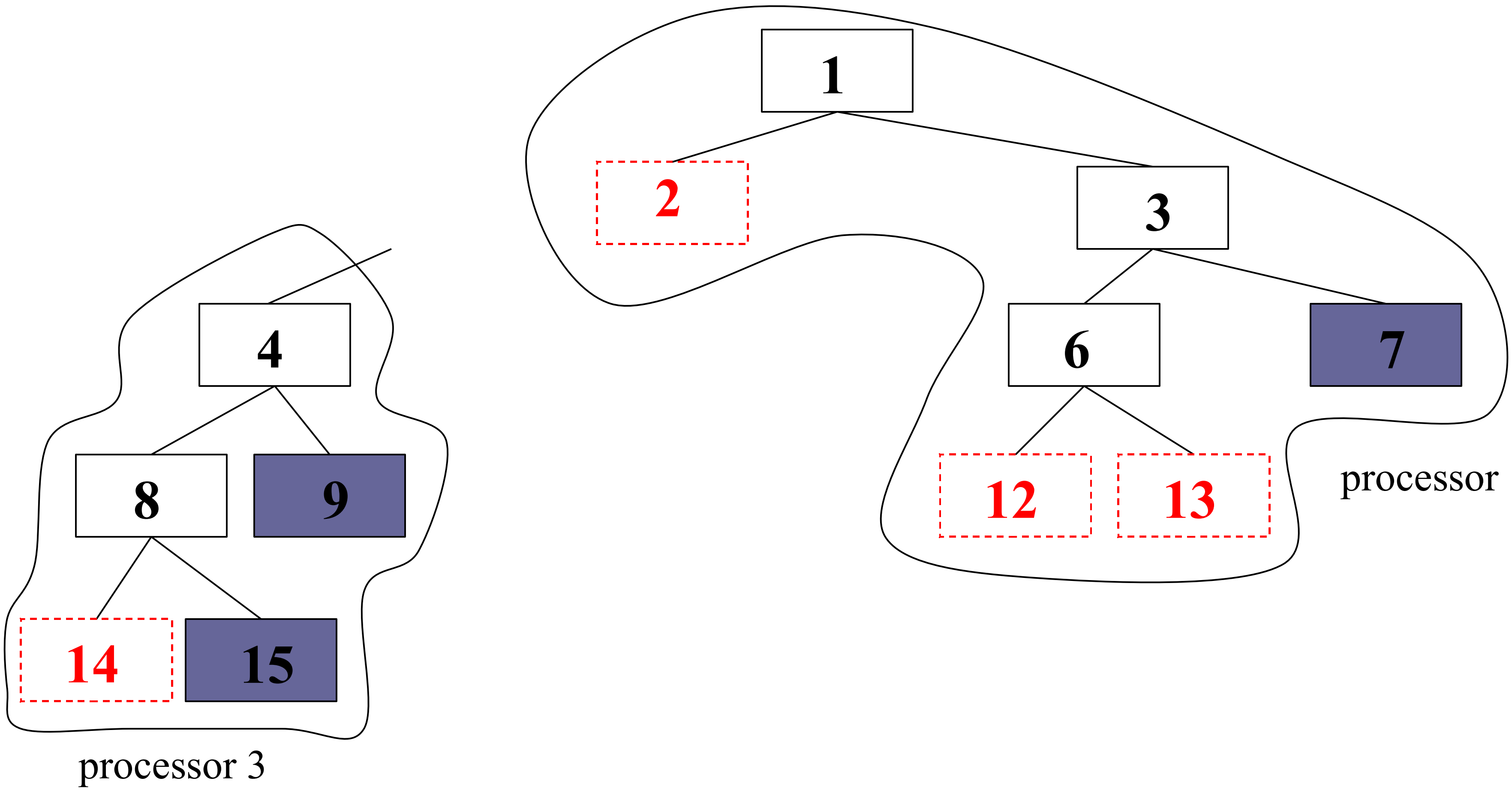
Crashing processors



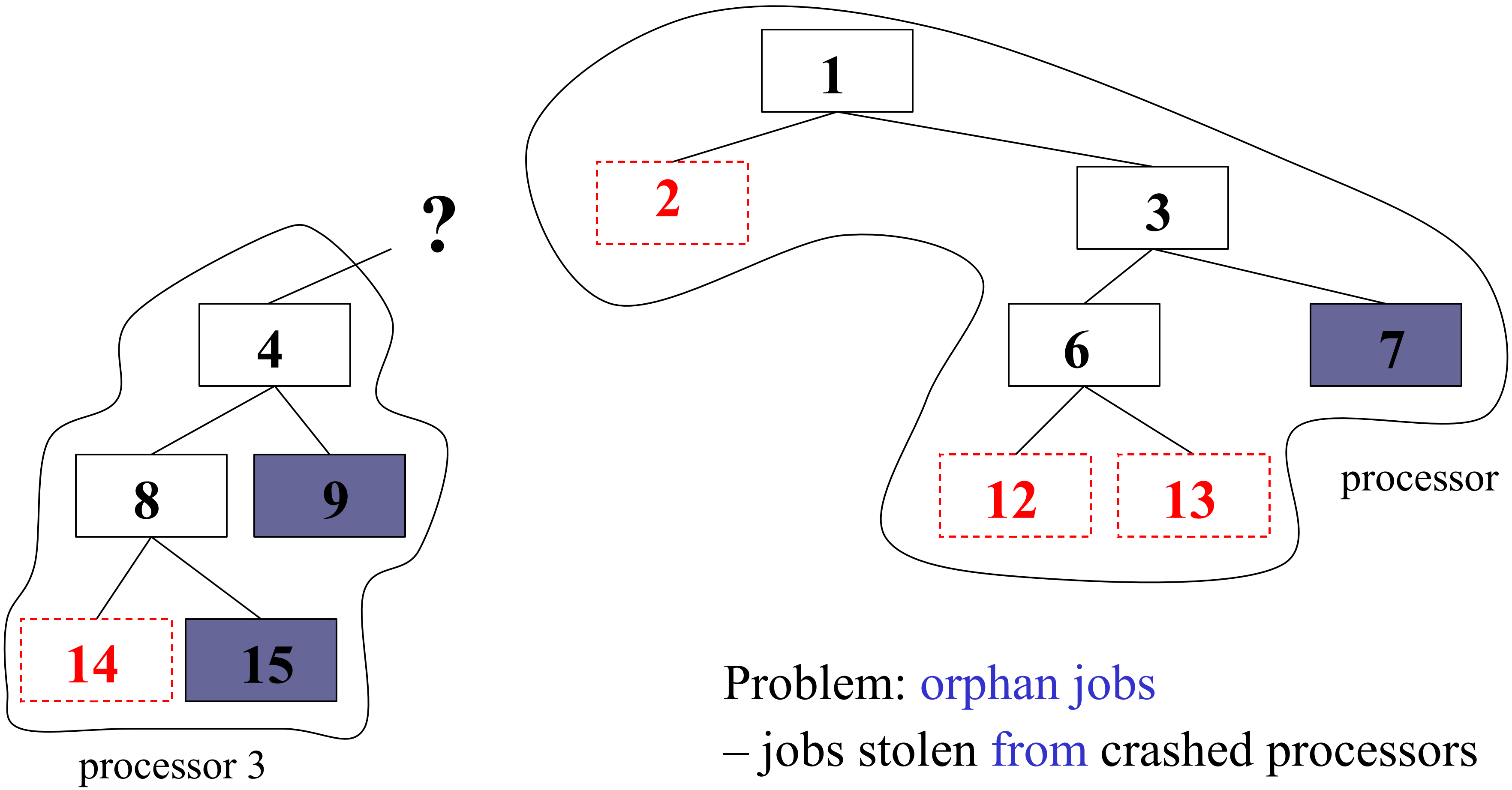
Crashing processors



Crashing processors



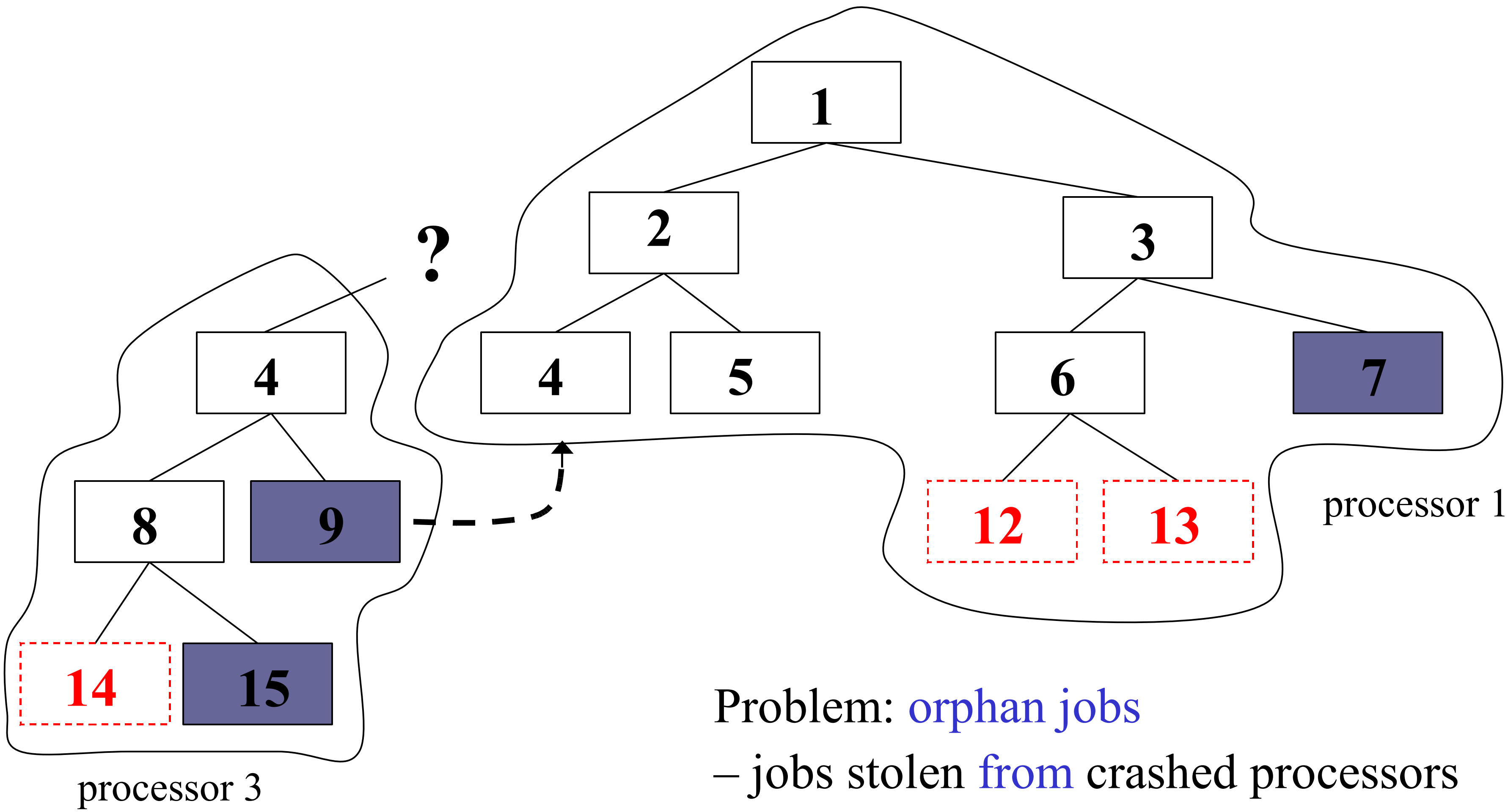
Crashing processors



Problem: orphan jobs

– jobs stolen from crashed processors

Crashing processors

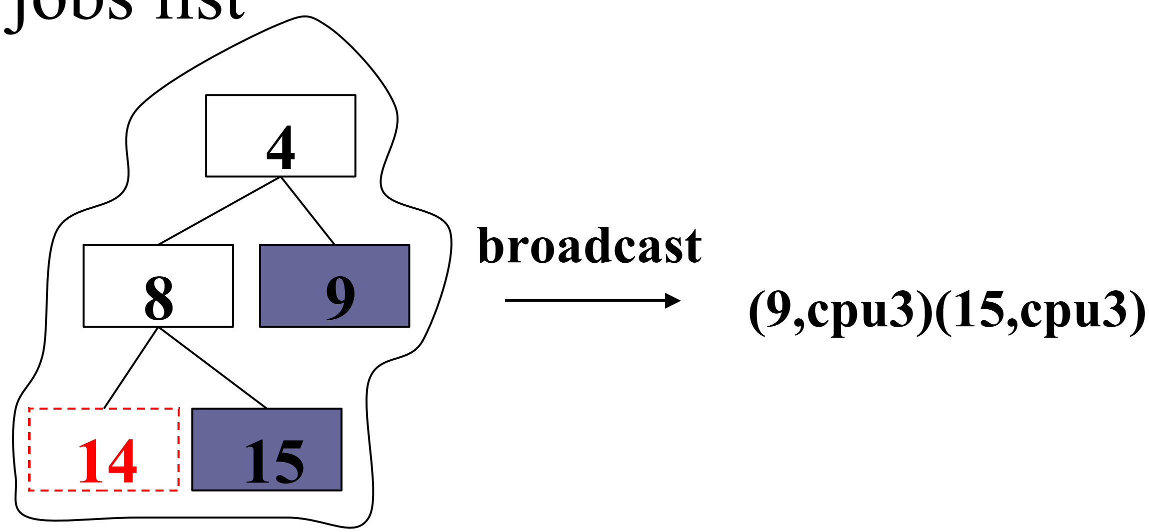


Problem: orphan jobs

– jobs stolen from crashed processors

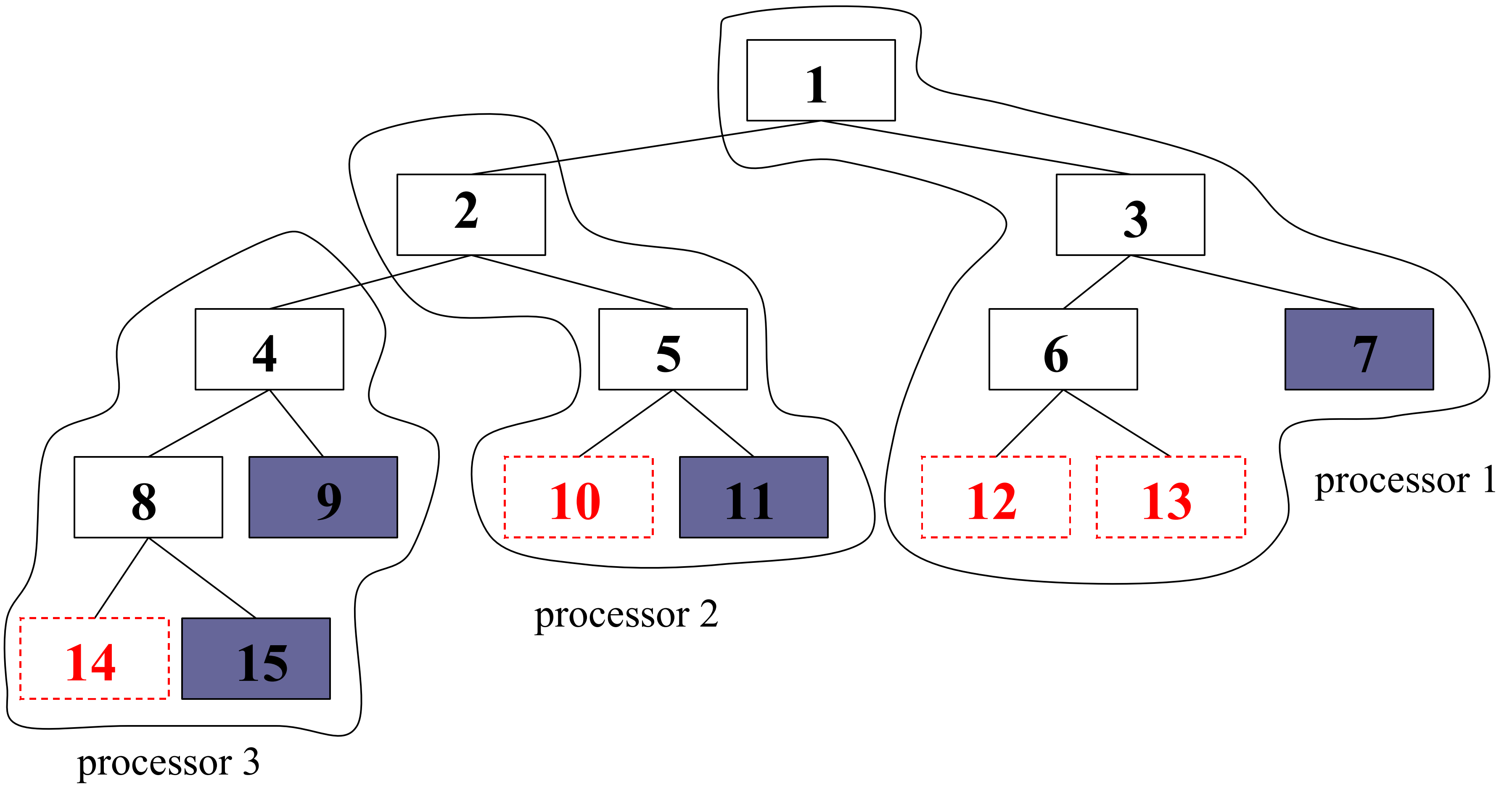
Handling orphan jobs

- For each finished orphan, broadcast (jobID,processorID) tuple; abort the rest
- All processors store tuples in orphan tables
- Processors perform lookups in orphan tables for each recomputed job
- If successful: send a result request to the owner (async), put the job on a stolen jobs list

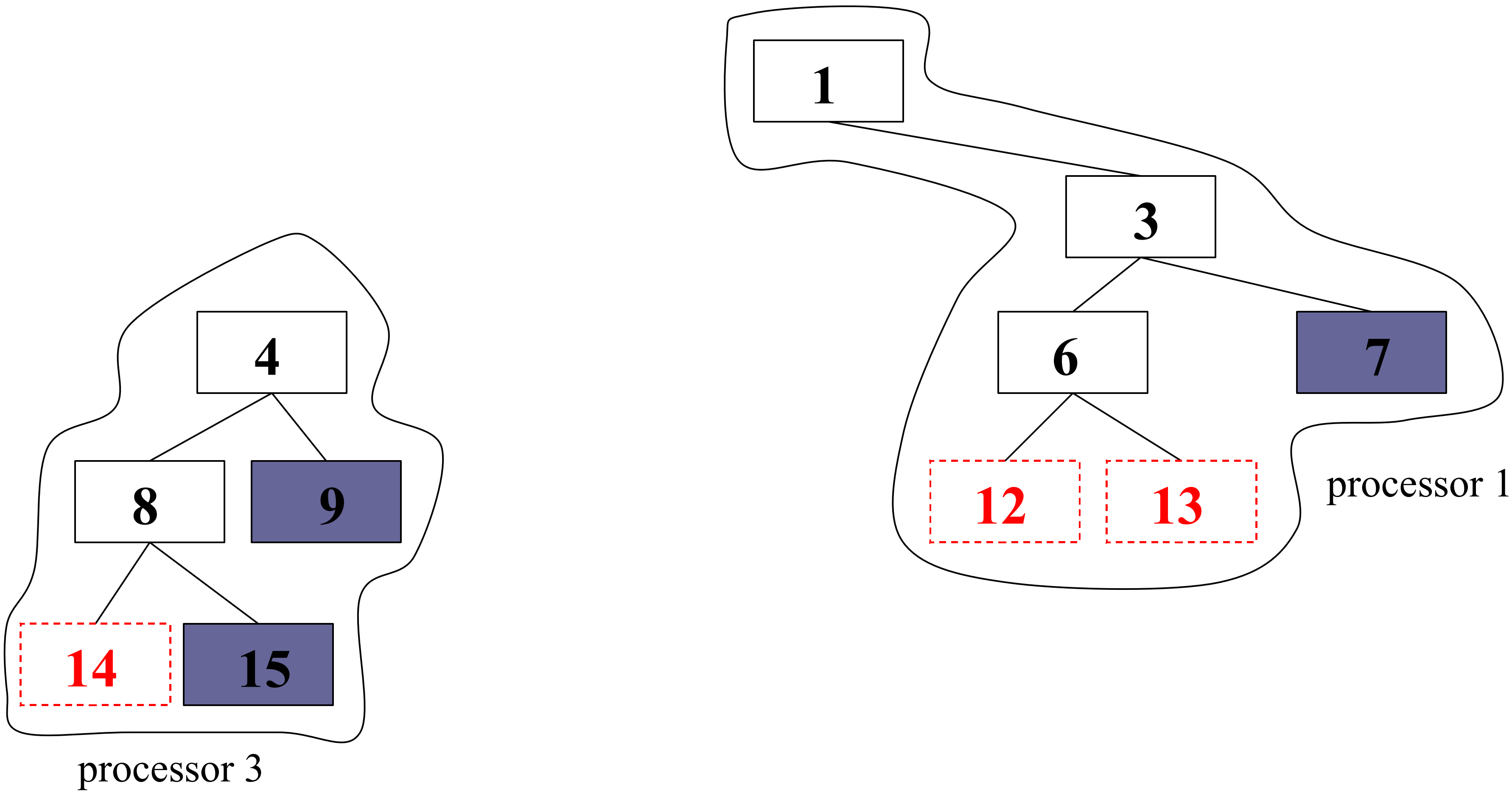


processor 3

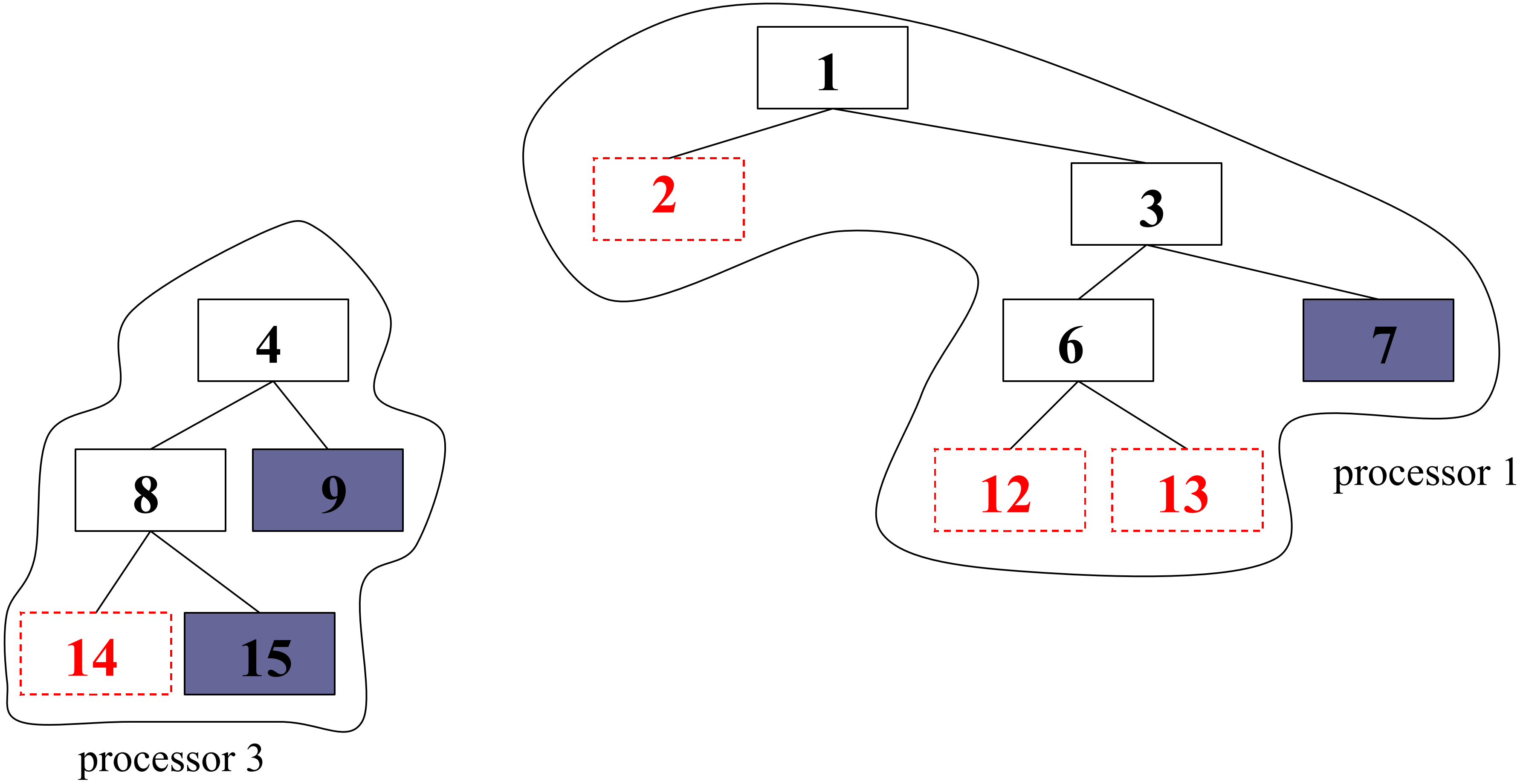
Handling orphan jobs - example



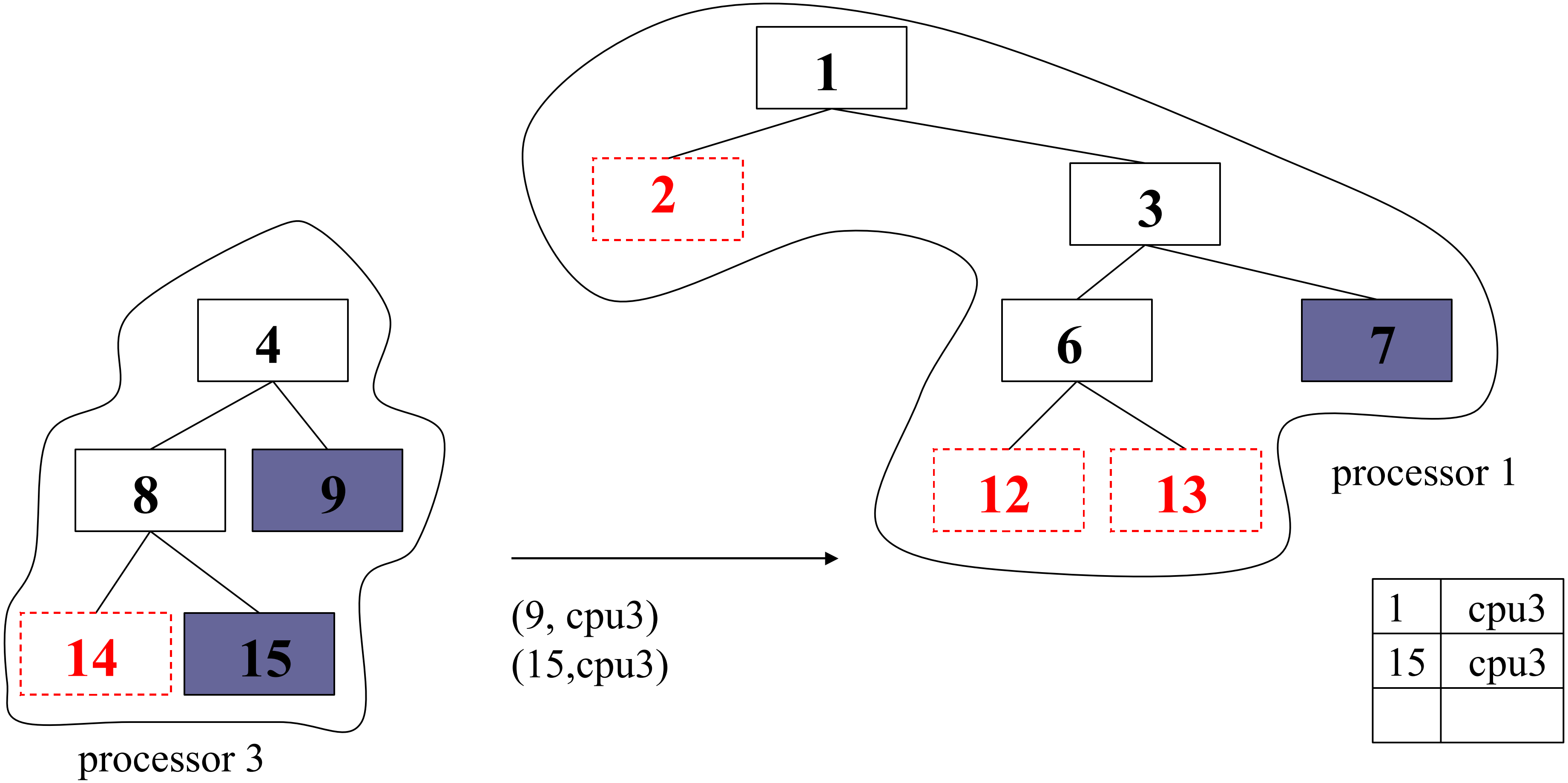
Handling orphan jobs - example



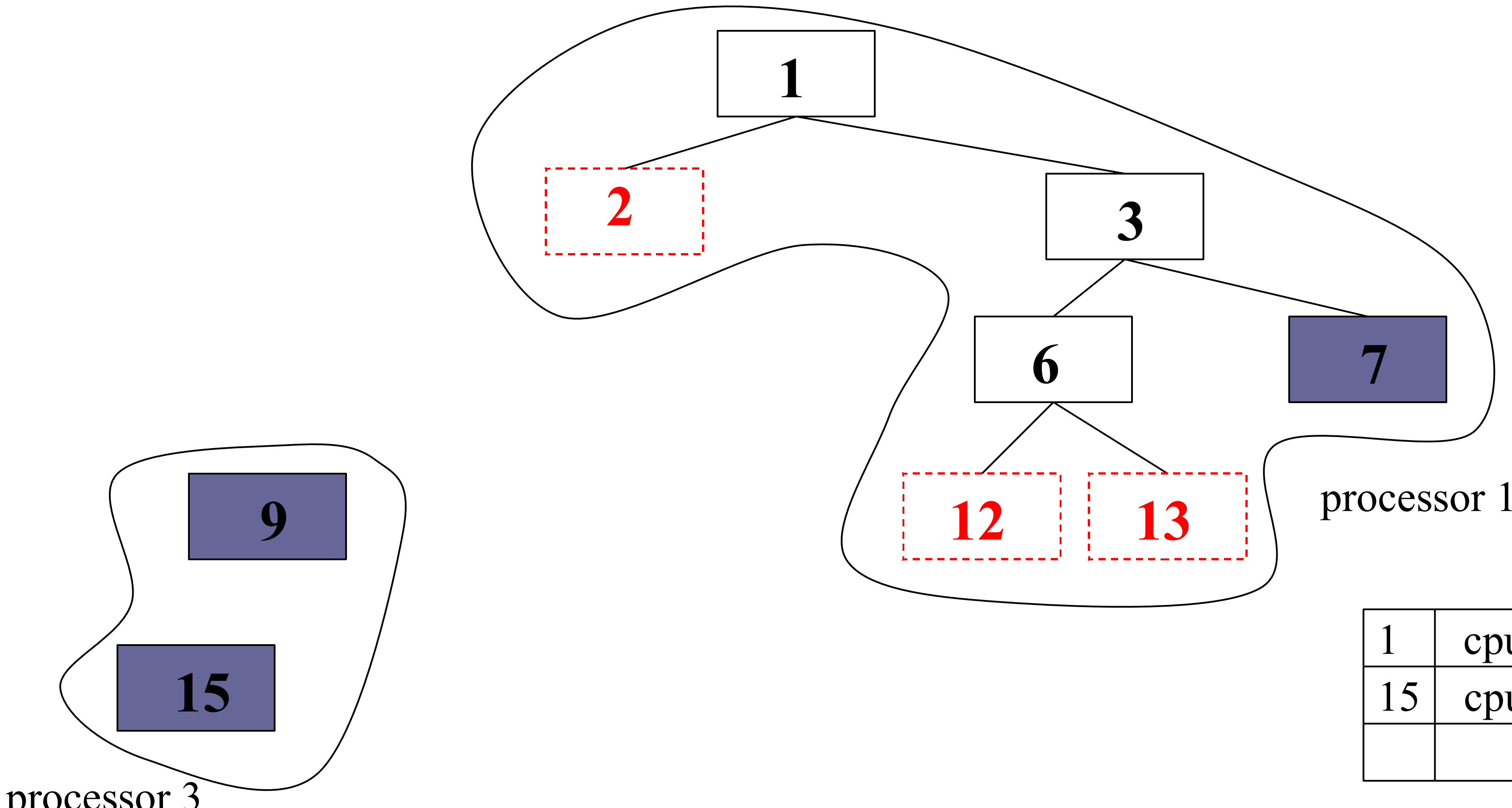
Handling orphan jobs - example



Handling orphan jobs - example

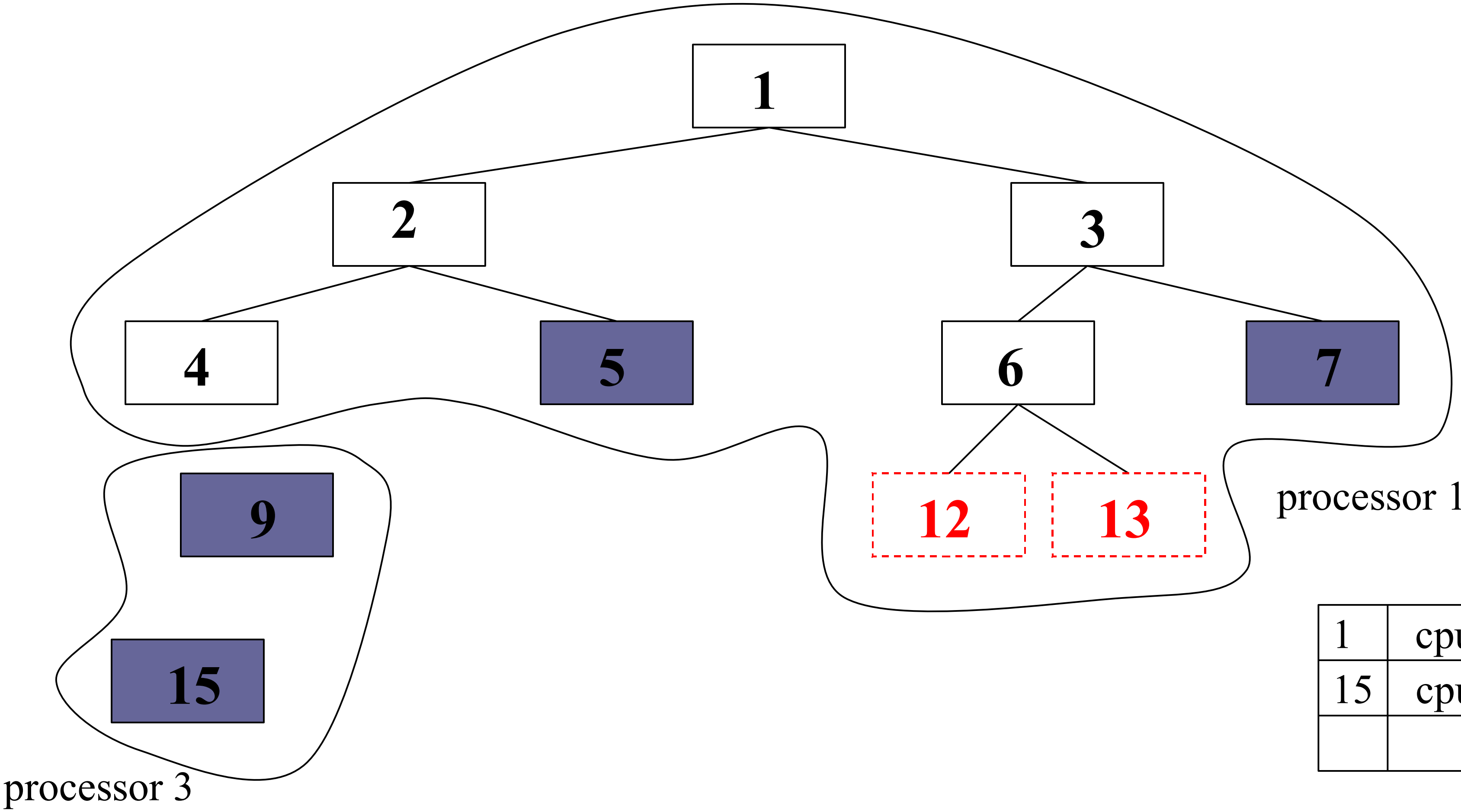


Handling orphan jobs - example



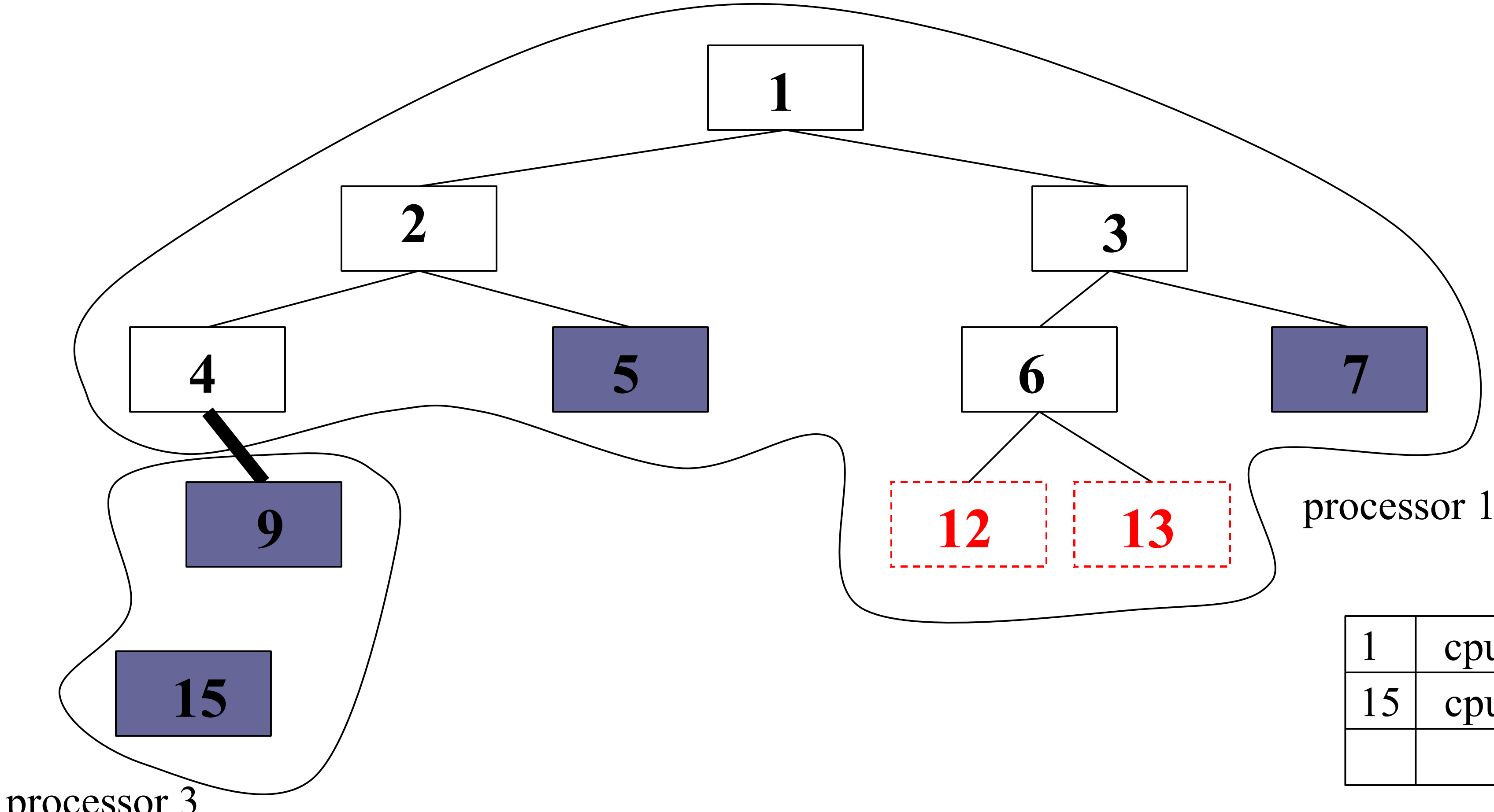
1	cpu3
15	cpu3

Handling orphan jobs - example



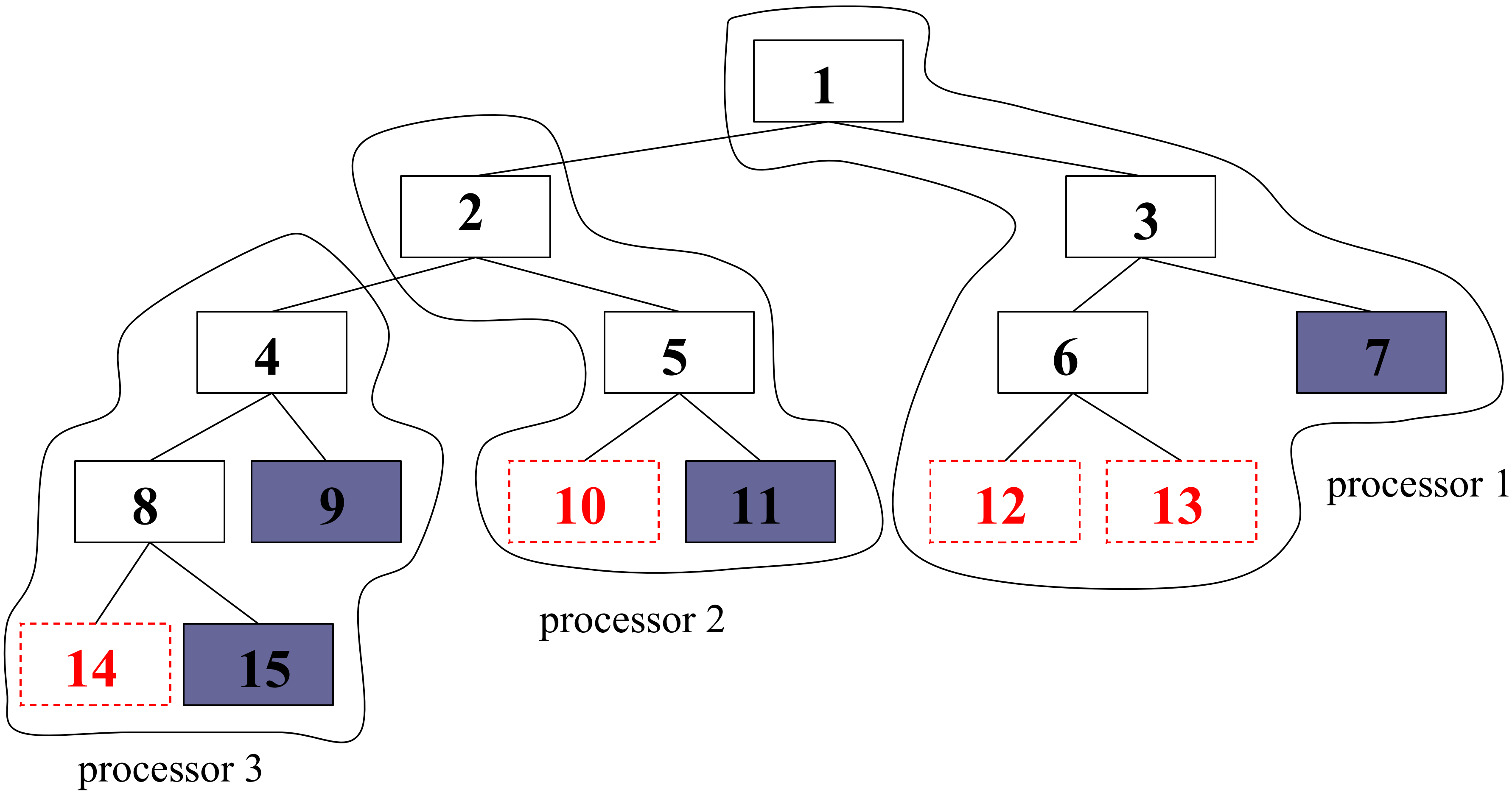
1	cpu3
15	cpu3

Handling orphan jobs - example

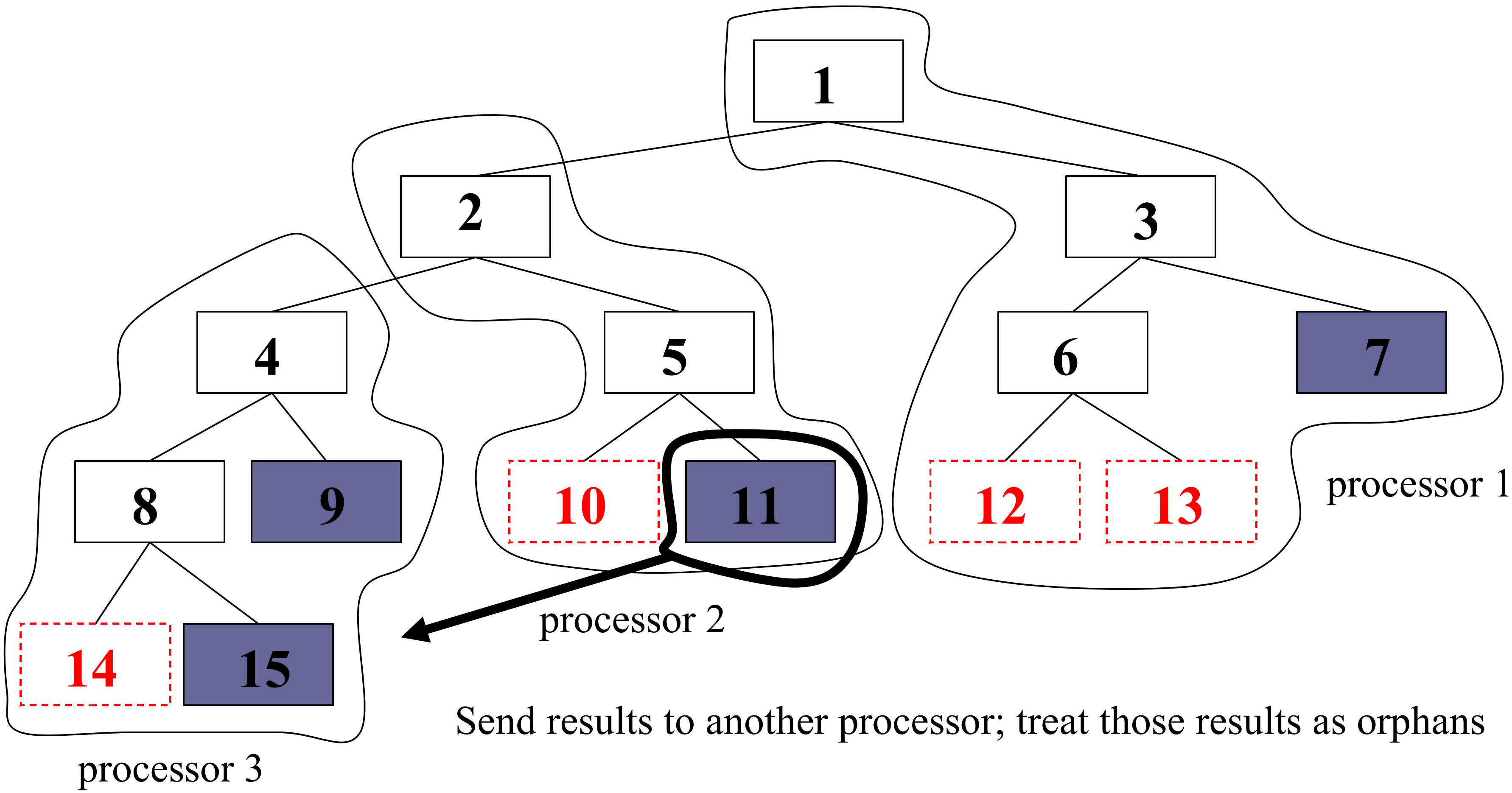


1	cpu3
15	cpu3

Processors leaving gracefully

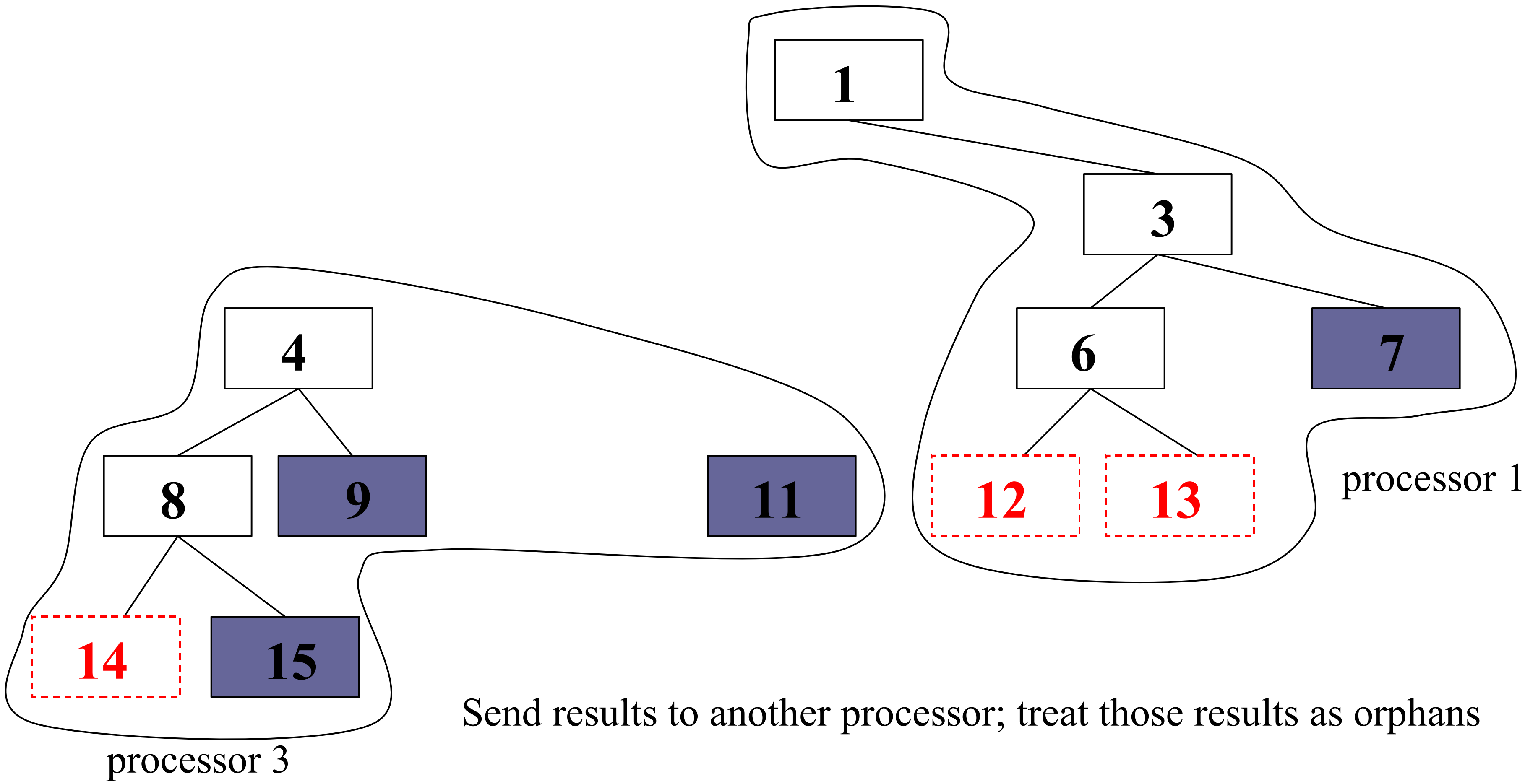


Processors leaving gracefully



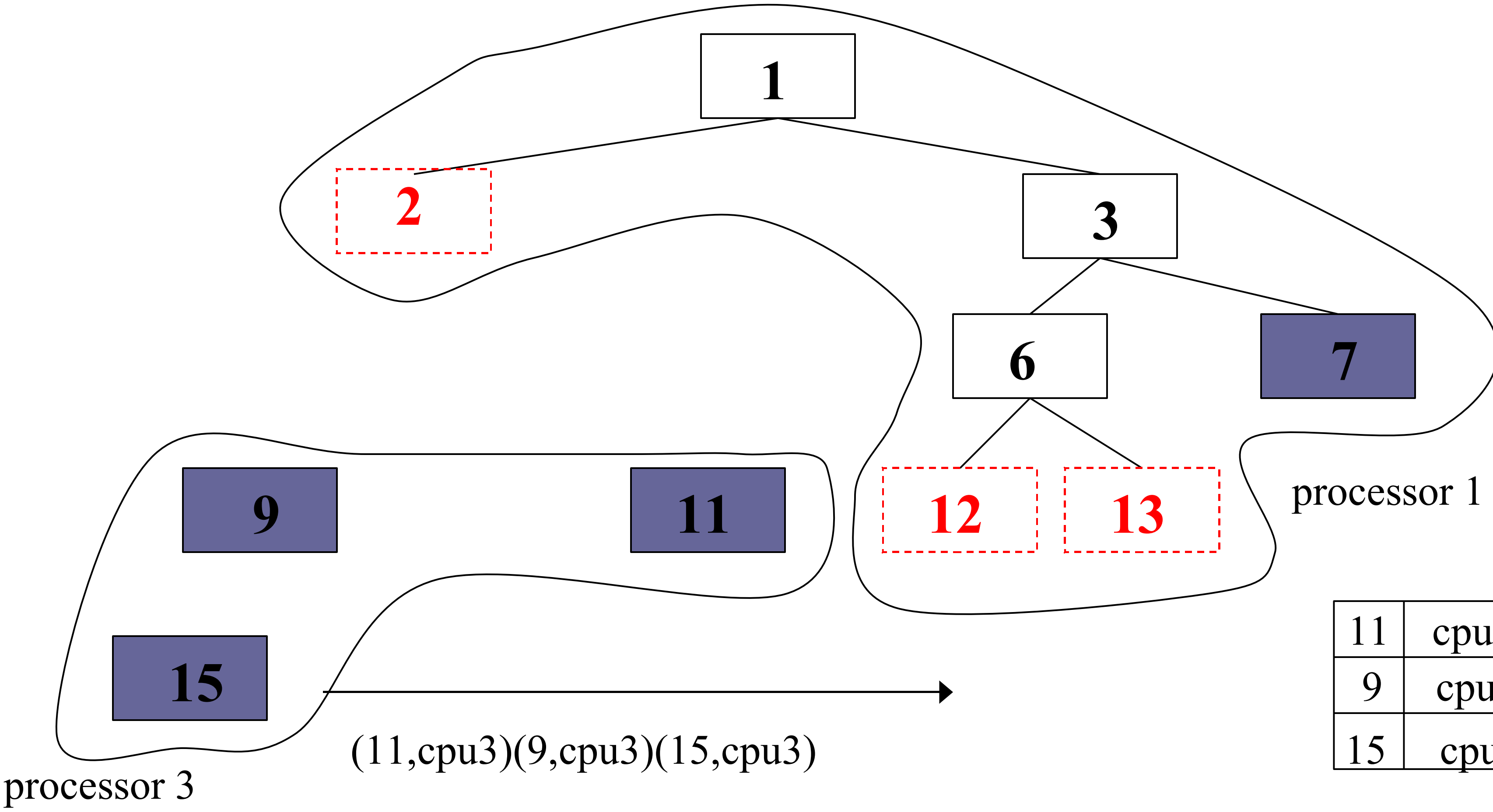
Send results to another processor; treat those results as orphans

Processors leaving gracefully



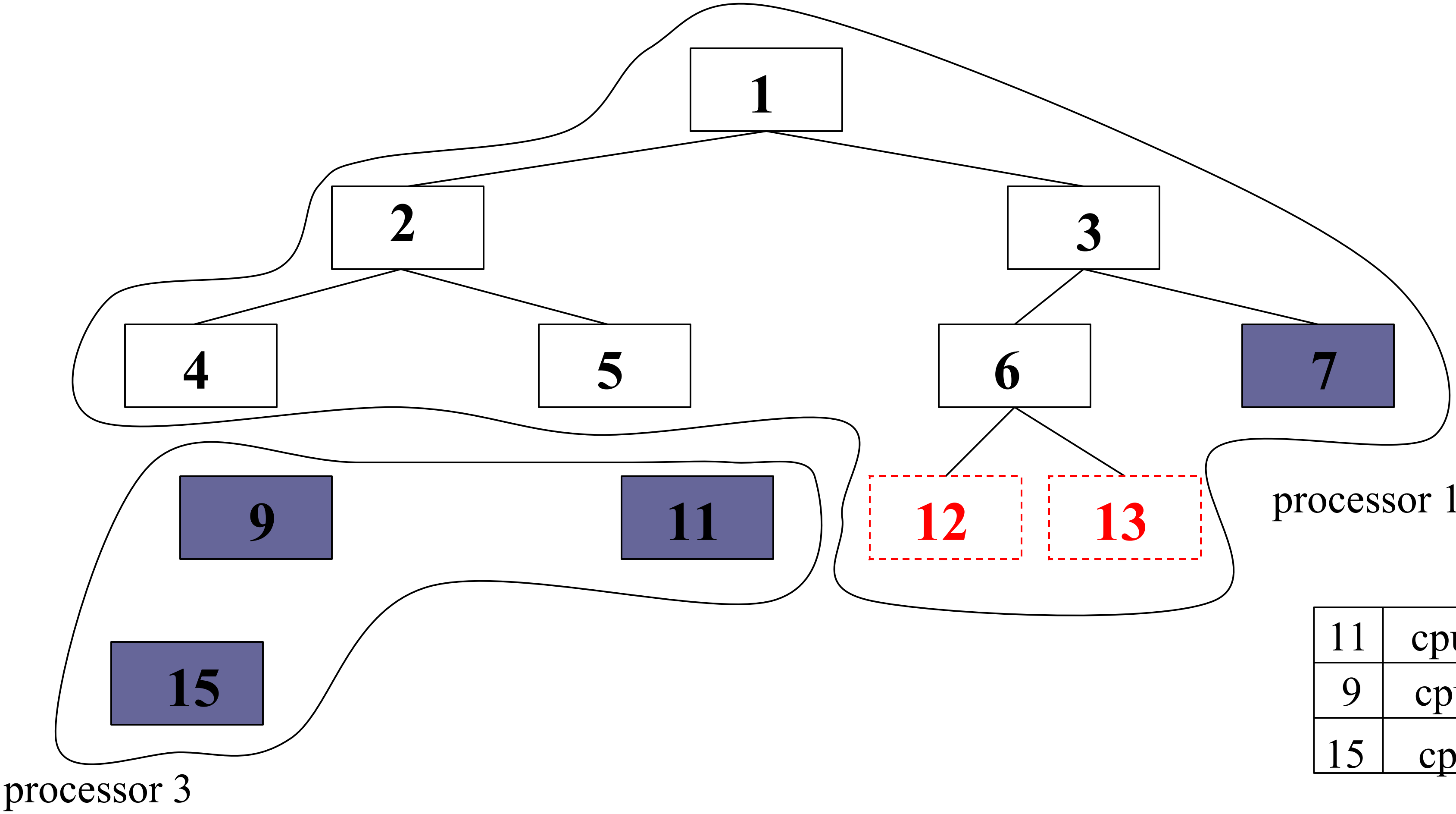
Send results to another processor; treat those results as orphans

Processors leaving gracefully



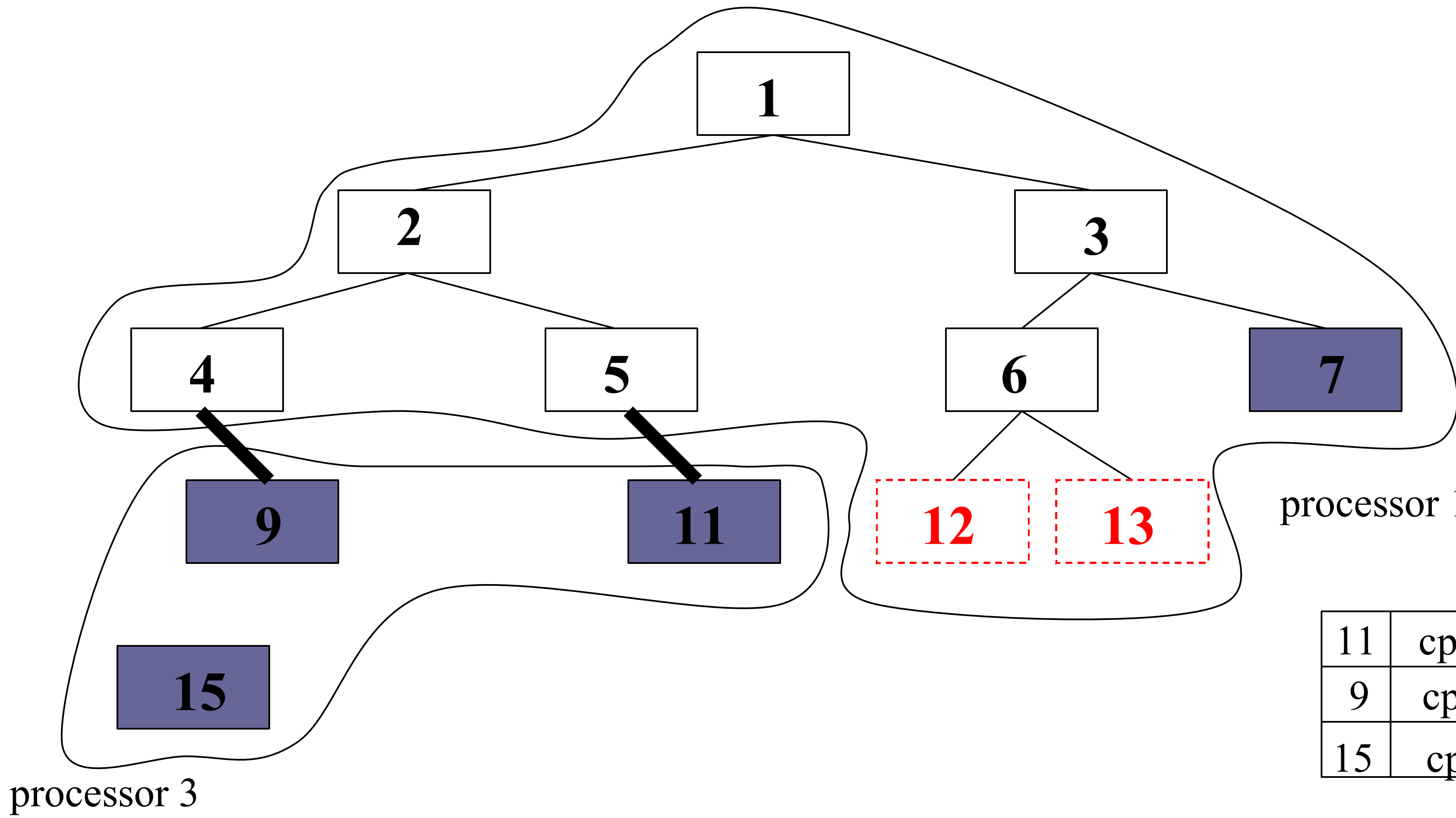
11	cpu3
9	cpu3
15	cpu3

Processors leaving gracefully



11	cpu3
9	cpu3
15	cpu3

Processors leaving gracefully



11	cpu3
9	cpu3
15	cpu3

A crash of the master

- Master: the processor that started the computation by spawning the root job
- If master crashes:
  - Elect a new master
  - Execute normal crash recovery
  - New master restarts the applications
  - In the new run, all results from the previous run are reused

Some remarks about scalability

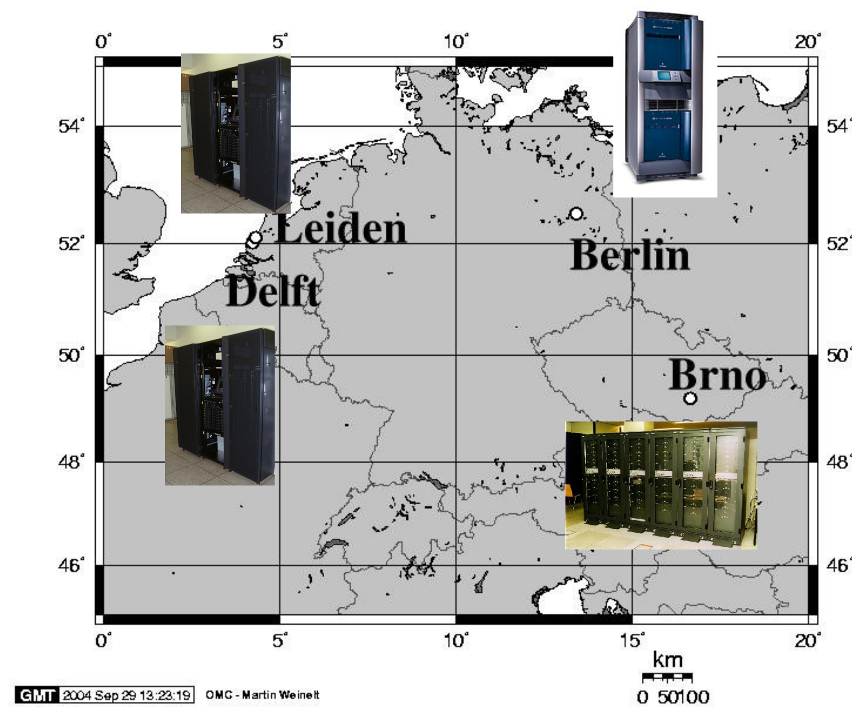
- Little data is broadcast (< 1% jobs, pointers)
- Message combining
- Lightweight broadcast: no need for reliability, synchronization, etc.

Performance evaluation

- Leiden, Delft (DAS-2) + Berlin, Brno (GridLab)
- Bandwidth:

  62 – 654 Mbit/s
- Latency:

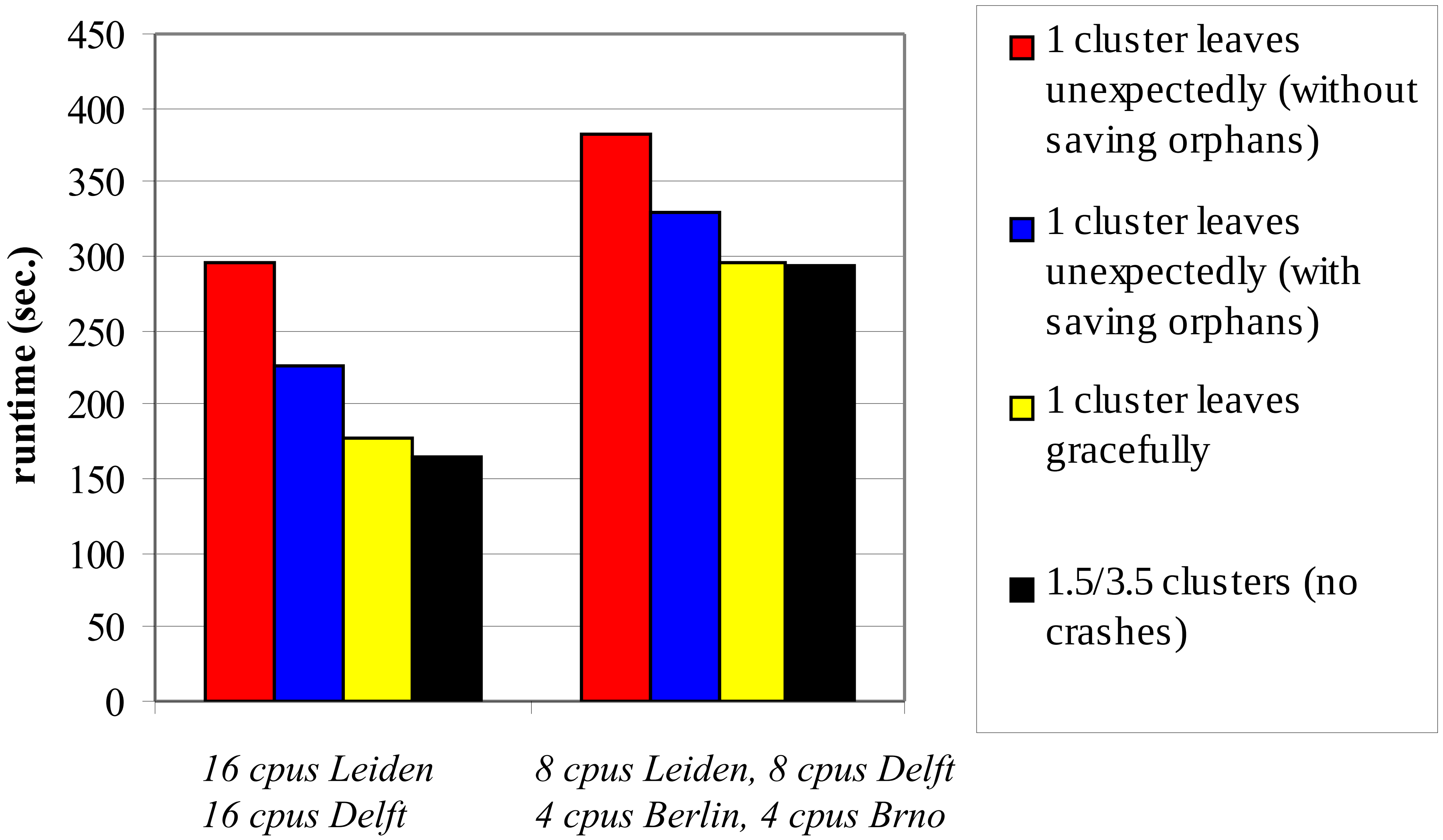
  2 – 21 ms



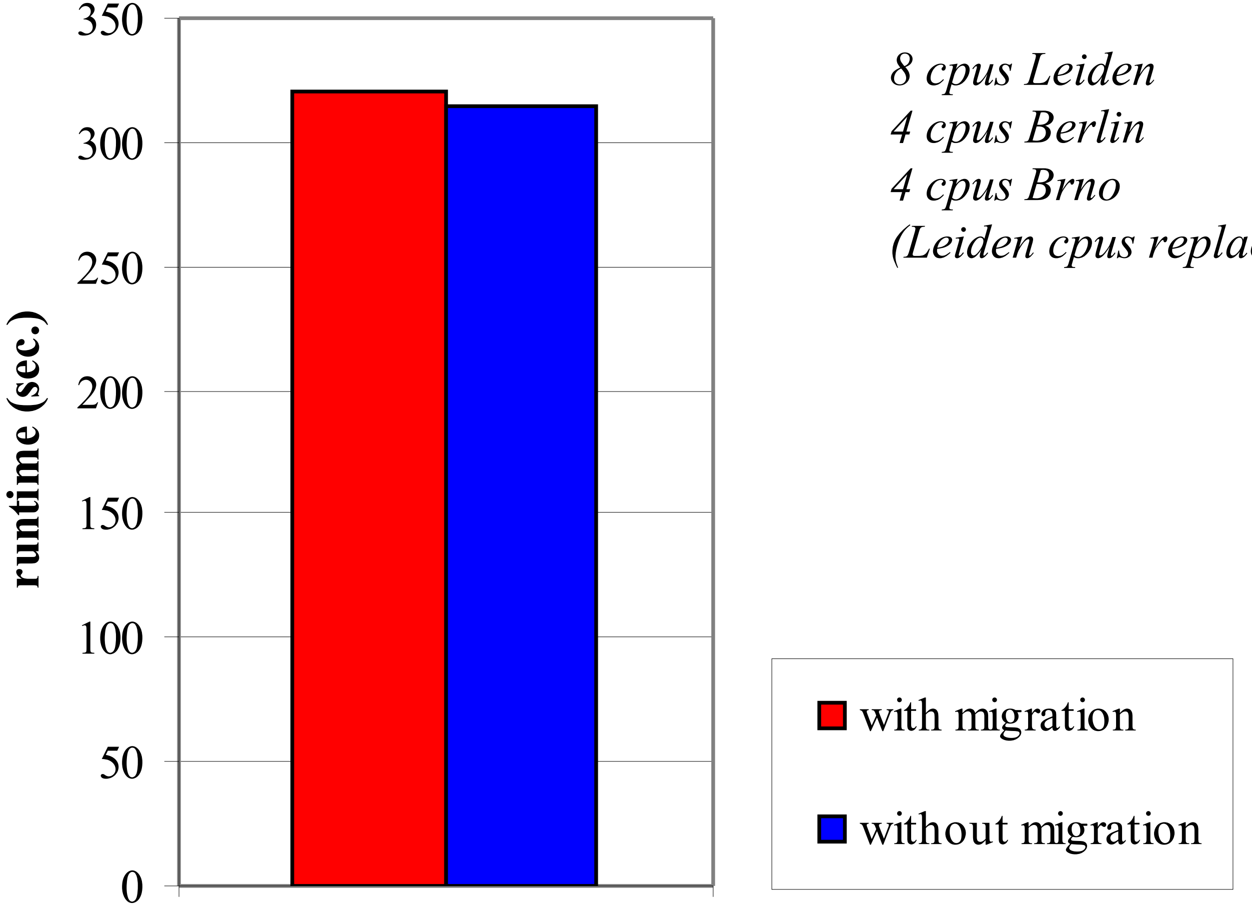
GridLab

Advanced School for Computing and Imaging

Impact of saving partial results



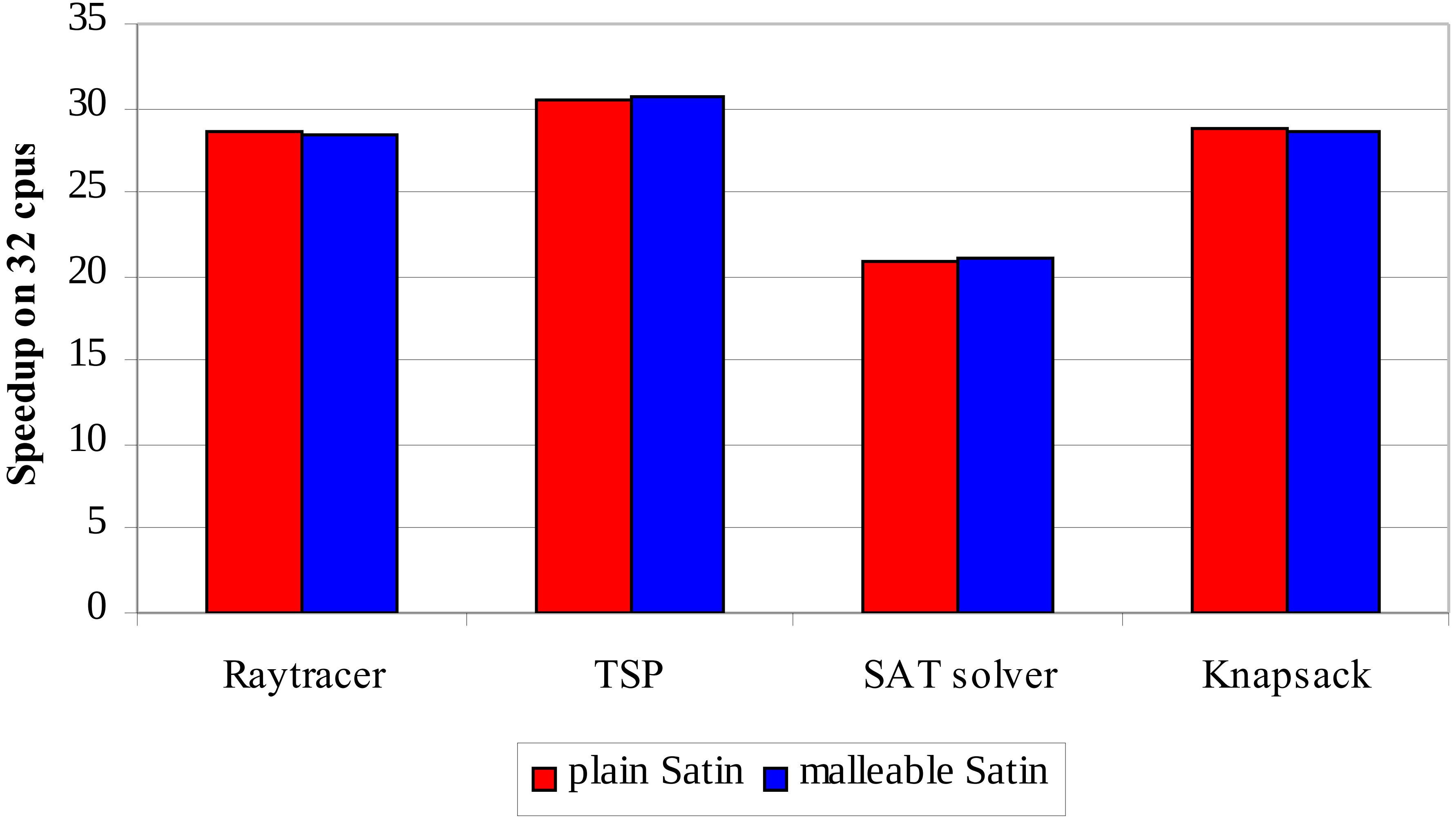
Migration overhead



8 cpus Leiden
4 cpus Berlin
4 cpus Brno
(Leiden cpus replaced by Delft)

Crash-free execution overhead

Used: 32 cpus in Delft



Summary

- Satin implements fault-tolerance, malleability and migration for divide-and-conquer applications
- Save partial results by repairing the execution tree
- Applications can adapt to changing numbers of cpus and migrate without loss of work (overhead < 10%)
- Outperform traditional approach by 25%
- No overhead during crash-free execution

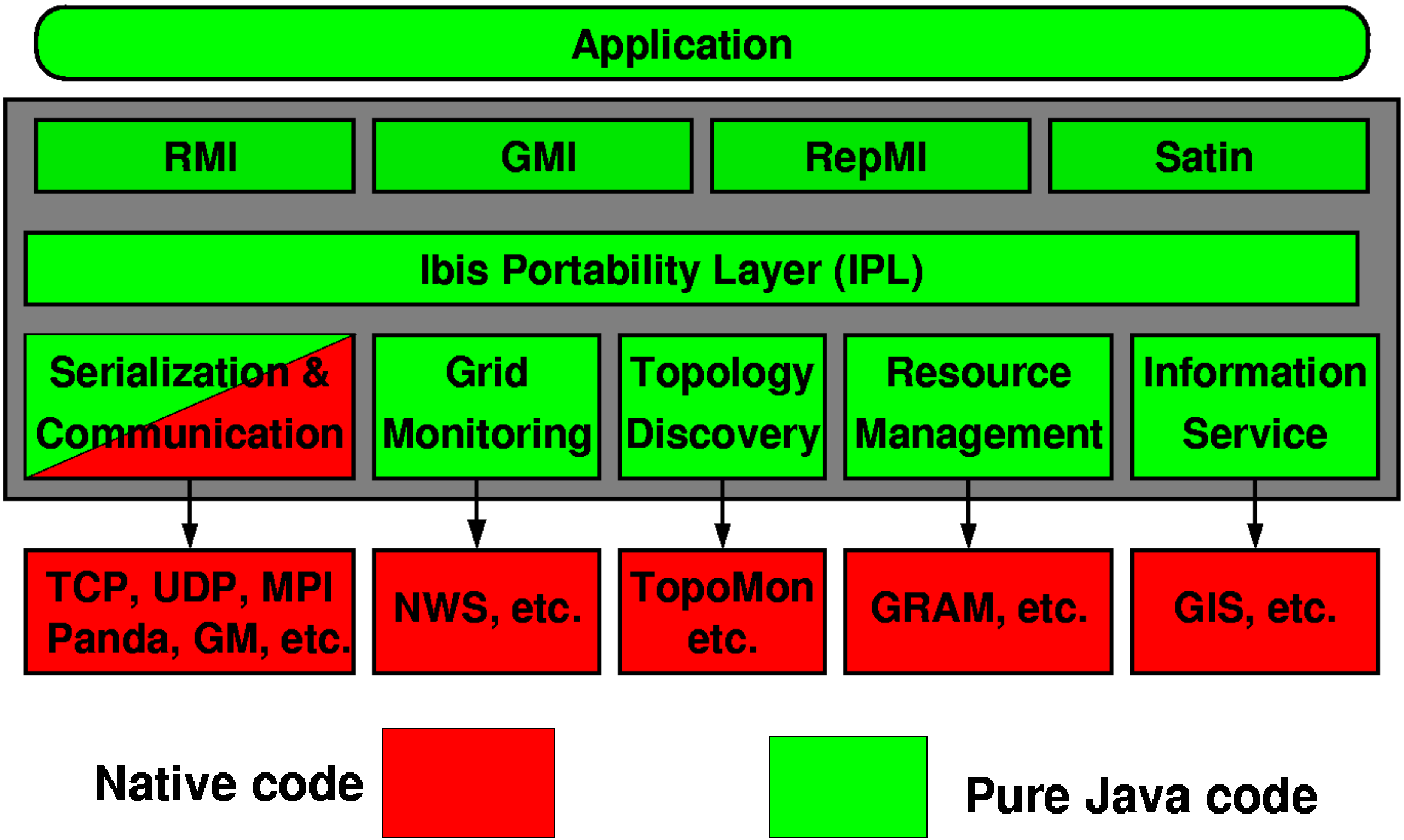
Further information

Publications and a software distribution available at:

http://www.cs.vu.nl/ibis/

Additional slides

Ibis design



Application
RMI
GMI
RepMI
Satin
Ibis Portability Layer (IPL)
Serialization & Communication
Grid Monitoring
Topology Discovery
Resource Management
Information Service
TCP, UDP, MPI Panda, GM, etc.
NWS, etc.
TopoMon etc.
GRAM, etc.
GIS, etc.
Native code
Pure Java code

Partial results on leaving cpus

If processors leave gracefully:

- Send all finished jobs to another processor
- Treat those jobs as orphans = broadcast (jobID, processorID) tuples
- Execute the normal crash recovery procedure

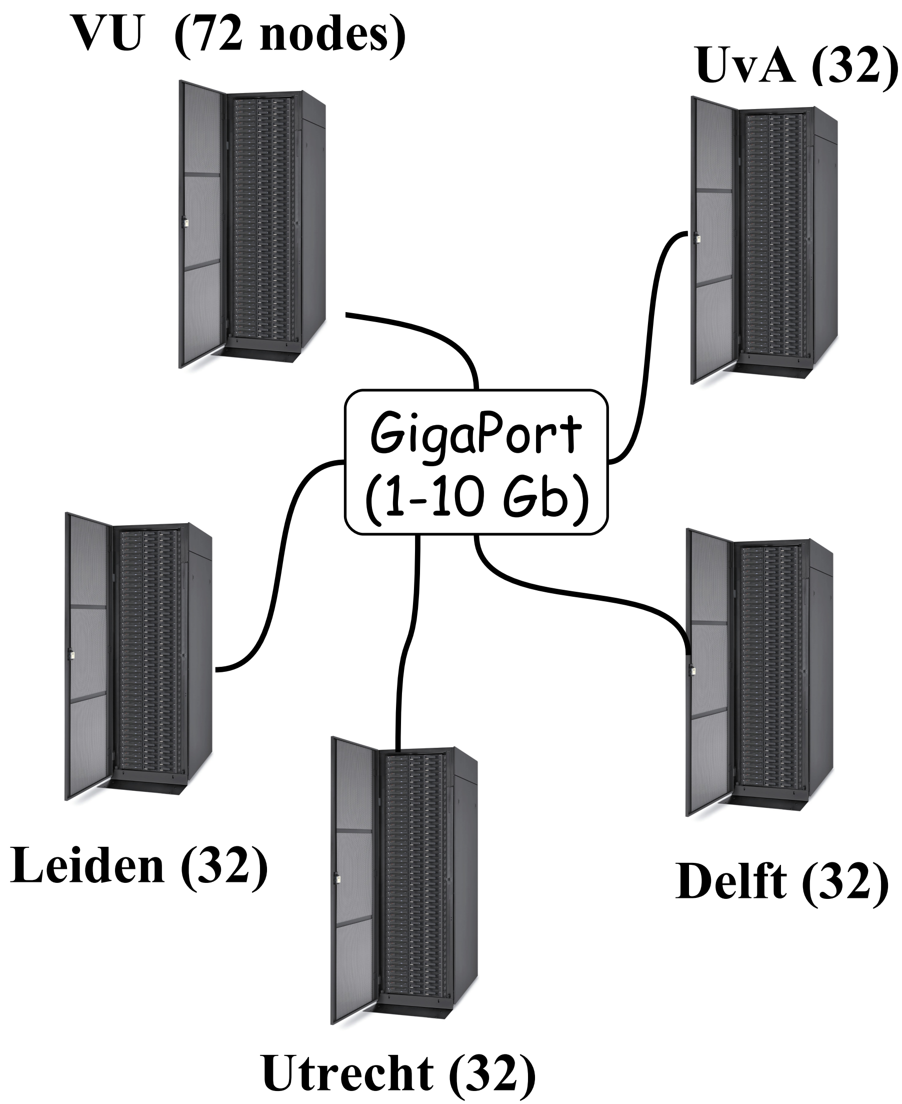
Job identifiers

- rootId = 1
- childId = parentId * branching_factor + child_no
- Problem: need to know maximal branching factor of the tree
- Solution: strings of bytes, one byte per tree level

Distributed ASCI Supercomputer (DAS) – 2

Node configuration

Dual 1 GHz Pentium-III
>= 1 GB memory
100 Mbit Ethernet +
(Myrinet)
Linux



Example

```
interface FibInter
   extends ibis.satin.Spawnable {
      public int fib(long n);
}

class Fib
  extends ibis.satin.SatinObject
  implements FibInter {
   public int fib (int n) {
      if (n < 2) return n;
      int x = fib (n - 1);
      int y = fib (n - 2);
      sync();
      return x + y;
   }
}
```

Java + divide&conquer

GridLab testbed

Grid results

Program	sites	CPUs	Efficiency
Raytracer	5	40	81 %
SAT-solver	5	28	88 %
Compression	3	22	67 %

- Efficiency based on normalization to single CPU type (1GHz P3)