High level programming for the Grid

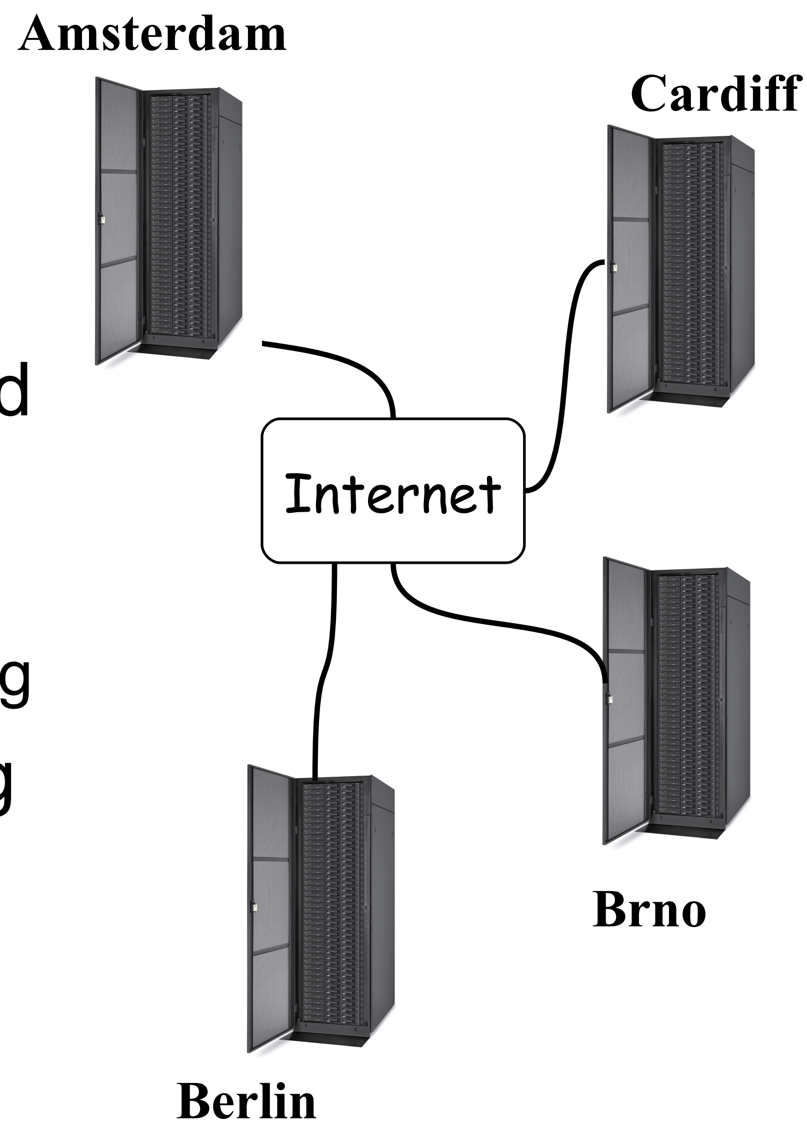
Gosia Wrzesinska
Dept. of Computer Science
Vrije Universiteit Amsterdam

vl·e

vrije Universiteit

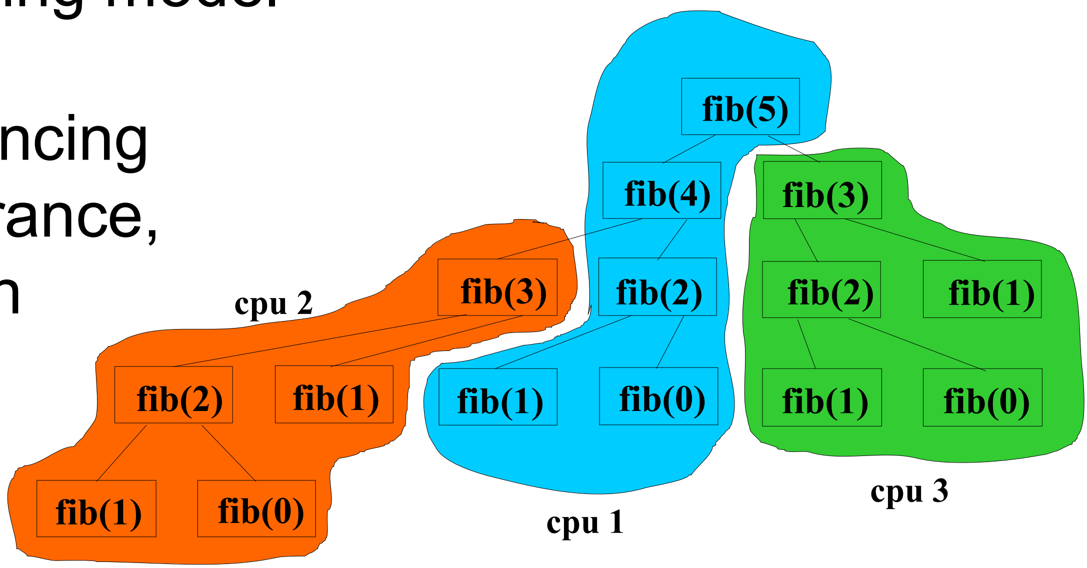
Distributed supercomputing

- Parallel processing on geographically distributed computing systems (grids)
- Programming for grids is hard
  - Heterogeneity
  - Slow network links
  - Nodes joining, leaving, crashing
- We need a grid programming environment to hide this complexity



Satin: Divide-and-Conquer for Grids

- Divide-and-Conquer (fork/join parallelism)
  - Is inherently hierarchical (fits the platform)
  - Has many applications: parallel rendering, SAT solver, VLSI routing, N-body simulation, multiple sequence alignment, grammar based learning
- Satin:
  - High-level programming model
  - Java-based
  - Grid aware load balancing
  - Support for fault tolerance, malleability, migration



Example: Raytracer

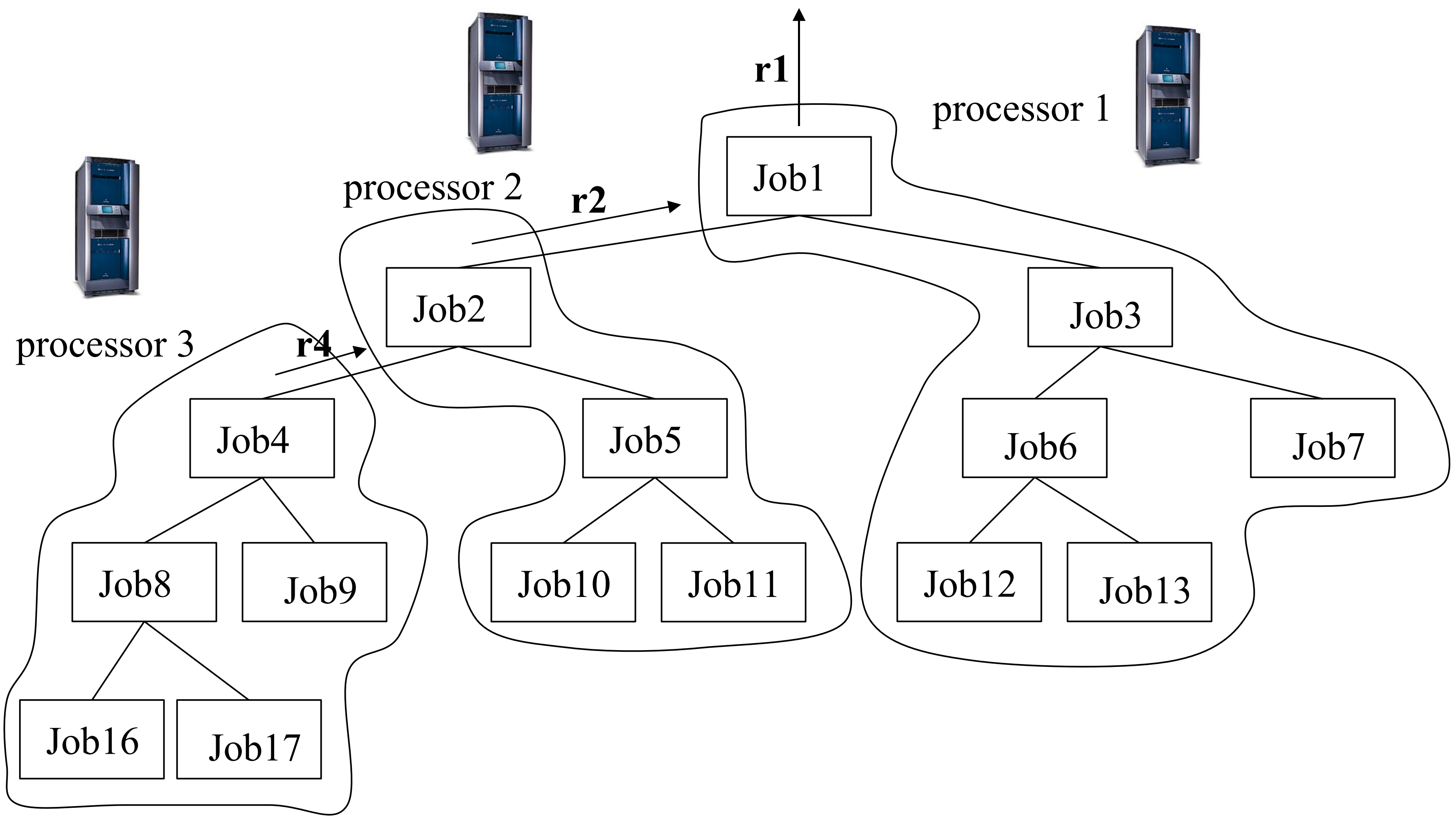
```
public class Raytracer
{
  BitMap render(Scene scene, int x, int y, int w, int h) {
    if (w < THRESHOLD && h < THRESHOLD) {
      /*render sequentially*/
    } else {
      res1 = render(scene,x,y,w/2,h/2);
      res2 = render(scene,x+w/2,y,w/2,h/2);
      res3 = render(scene,x,y+h/2,w/2,h/2);
      res4 = render(scene,x+w/2,y+h/2,w/2,h/2);

      return combineResults(res1, res2, res3, res4);
    }
  }
}
```

Parallelizing the Raytracer

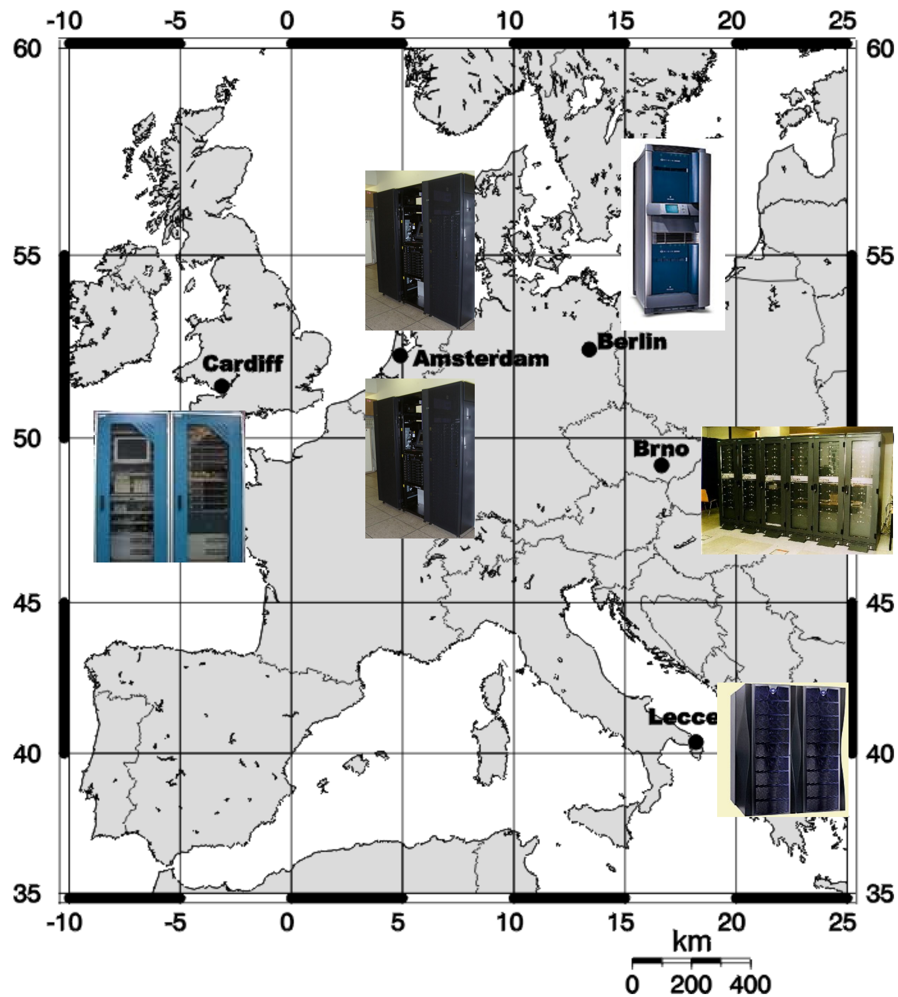
```
interface RaytracerInterface extends satin.Spawnable {
 BitMap render(Scene scene, int x, int y, int w, int h);
}
public class Raytracer extends satin.SatinObject()
implements RaytracerInterface{
  BitMap render(Scene scene, int x, int y, int w, int h) {
    if (w < THRESHOLD && h < THRESHOLD) {
      /*render sequentially*/
    } else {
      res1 = render(scene,x,y,w/2,h/2); /*spawn*/
      res2 = render(scene,x+w/2,y,w/2,h/2); /*spawn*/
      res3 = render(scene,x,y+h/2,w/2,h/2); /*spawn*/
      res4 = render(scene,x+w/2,y+h/2,w/2,h/2); /*spawn*/
      sync();
      return combineResults(res1, res2, res3, res4);
    }
  }
}
```

Running Satin applications



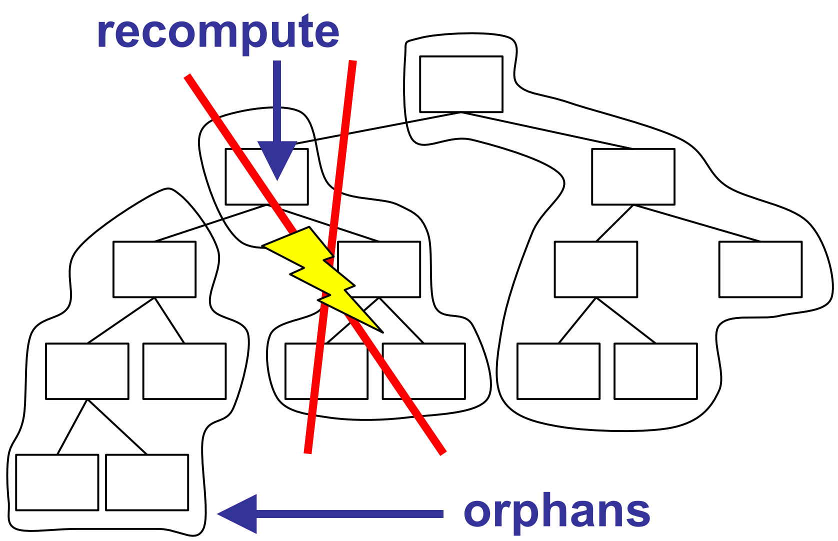
Performance on the Grid

- GridLab testbed: 5 cities in Europe
- 40 cpus in total
- Different architectures, OS
- Large differences in processor speeds
- Latencies:
  - 0.2 – 210 ms daytime
  - 0.2 – 66 ms night
- Bandwidth:
  - 9KB/s – 11MB/s



80% efficiency

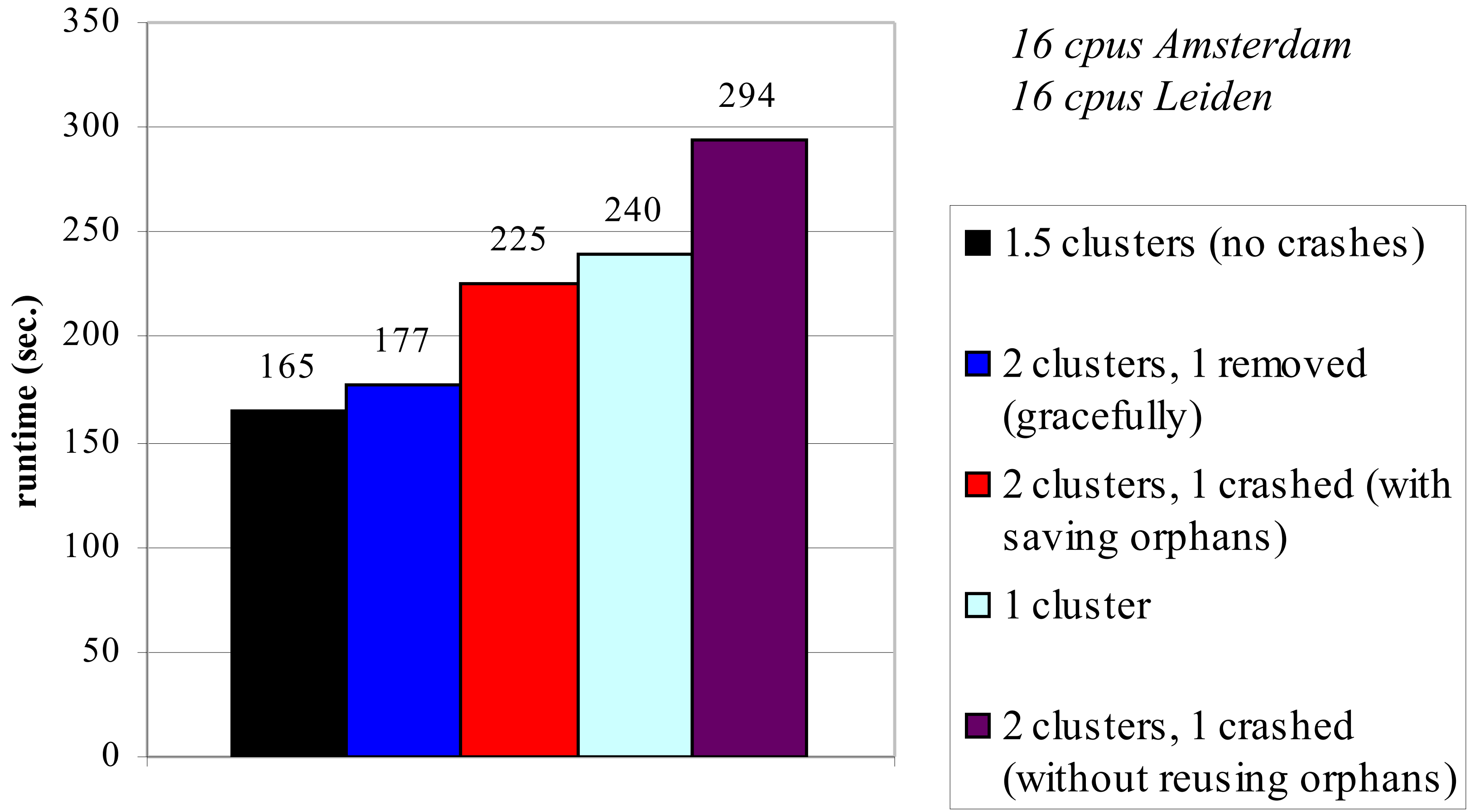
Fault tolerance, malleability, migration



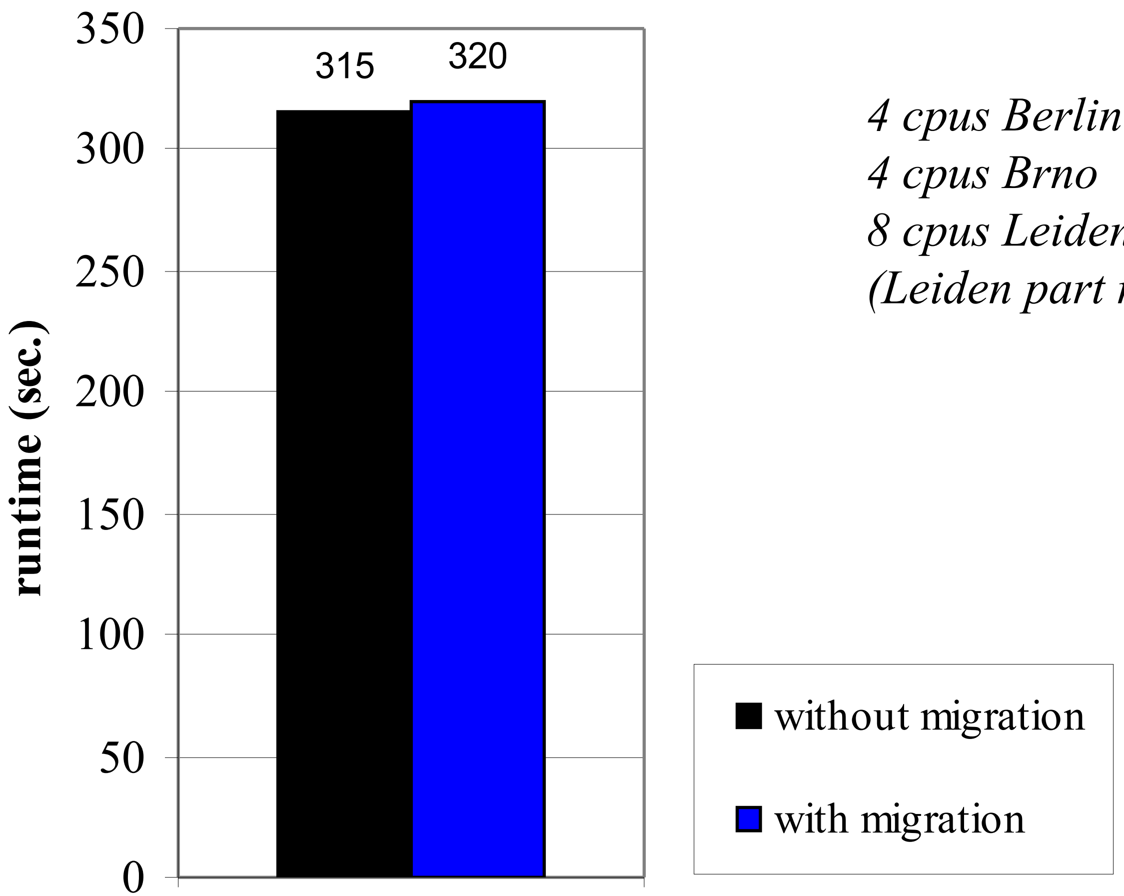
- Join: let it start stealing
- Leave, crash:
  - avoid checkpointing
  - recompute
- Optimizations:
  - reusing orphan jobs
  - reusing results from gracefully leaving processors

- We can:
  - Tolerate crashes with minimal loss of work
  - Add and remove (gracefully) processors with no loss
  - Efficiently migrate (add new nodes + remove old nodes)

The performance of FT and malleability



Efficient migration



4 cpus Berlin
4 cpus Brno
8 cpus Leiden
(Leiden part migrated to Delft)

Shared data for d&c applications

- Data sharing abstraction needed to extend applicability of Satin
  - Branch & bound, game tree search etc.
- Sequential consistency inefficient on the Grid
  - High latencies
  - Nodes leaving and joining
- Applications often allow weaker consistency

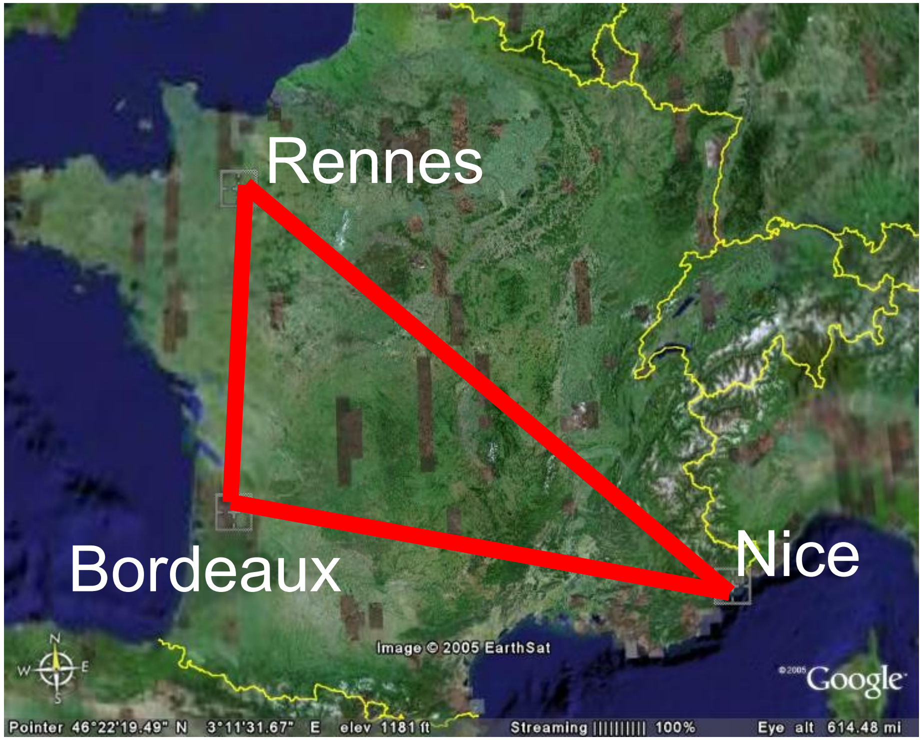
Shared objects with guard consistency

- Define consistency requirements with guard functions
  - Guard checks if the local replica is consistent
- Replicas allowed to become inconsistent as long as guards satisfied
  - If guard unsatisfied, bring replica into consistent state
- Applications: VLSI routing, learning SAT solver, TSP, N-body simulation

Shared objects performance

Grid'5000

- **3 clusters** in France (Grid5000), **120 nodes**
- Wide-area, heterogeneous testbed
- Latency: 4-10 ms
- Bandwidth: 200-1000Mbps
- Ran VLSI routing app



86% efficiency

Summary

- Satin: a grid programming environment
  - Allows rapid development of parallel applications
  - Performs well on wide-area, heterogeneous systems
  - Adapts to changing sets of resources
  - Tolerates node crashes
  - Provides divide-and-conquer + shared objects programming model
  - Applications: parallel rendering, SAT solver, VLSI routing, N-body simulation, multiple sequence alignment, grammar based learning etc.

Acknowledgements

Henri Bal
Jason Maassen
Rob van Nieuwpoort
Ceriel Jacobs
Kees Verstoep
Kees van Reeuwijk
Maik Nijhuis
Thilo Kielmann

vl·e
GridLab
Grid'5000
vrije Universiteit

Publications and software distribution available at:

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

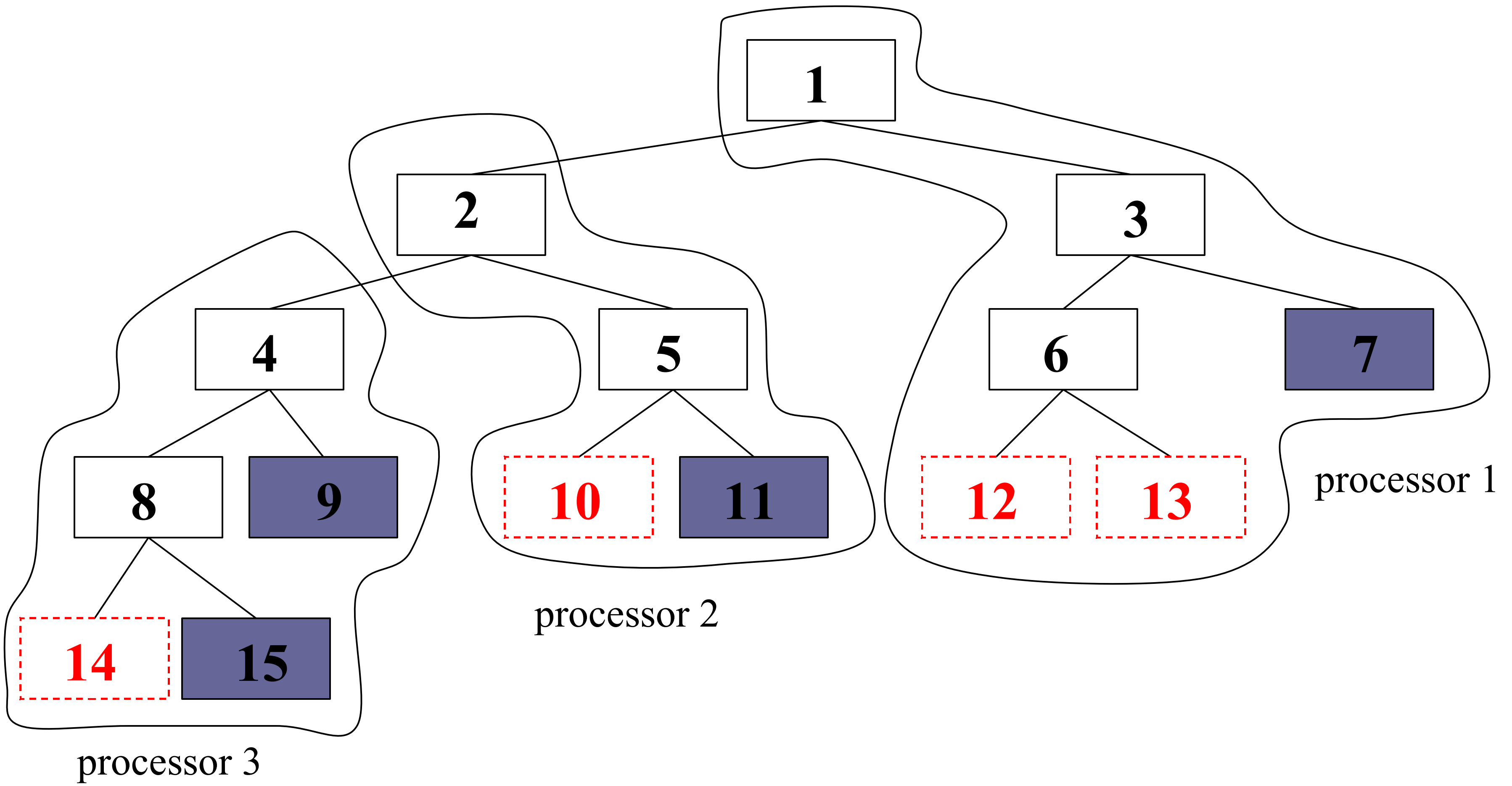
Additional Slides

Guards: example

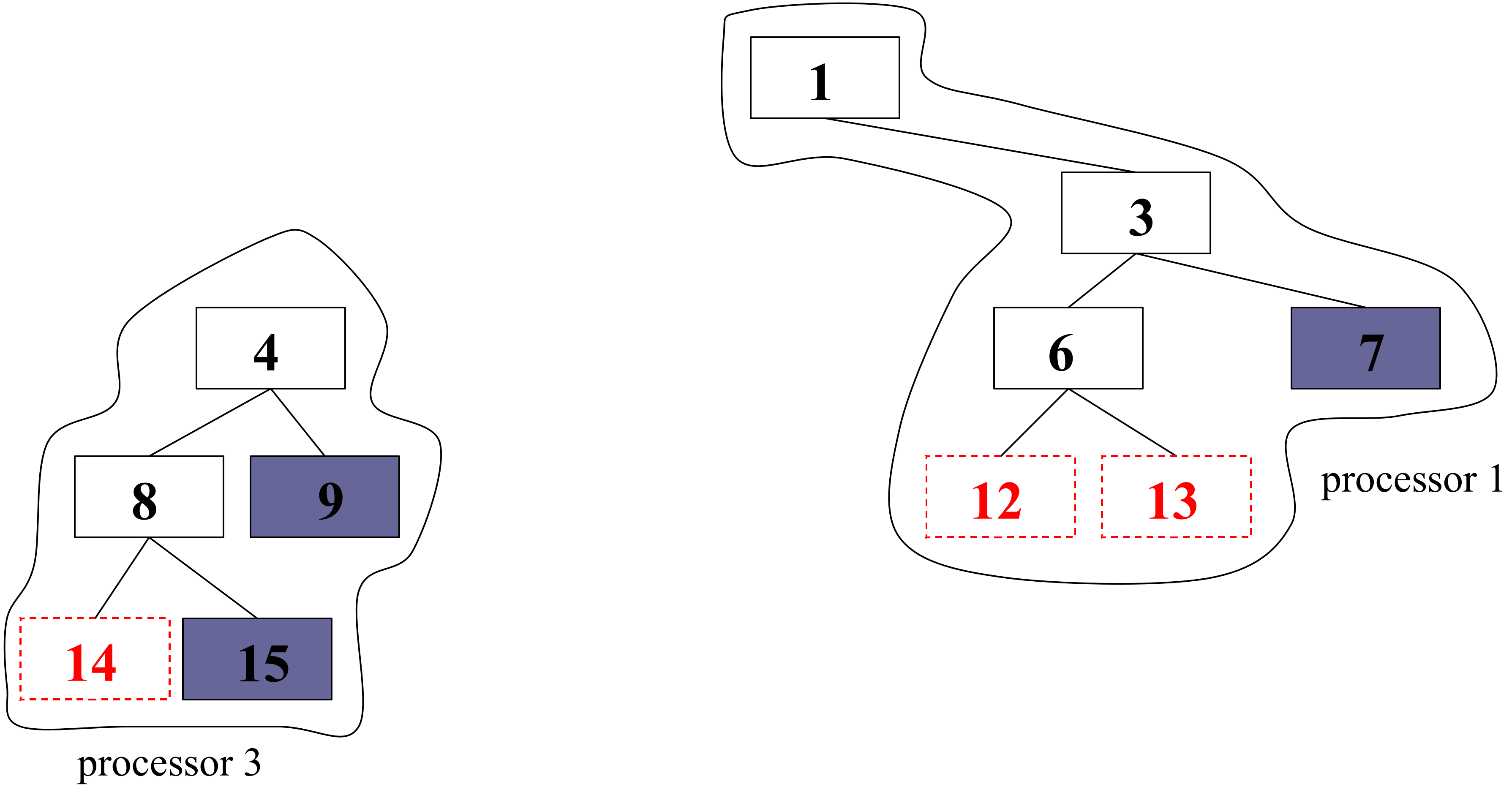
```
/*divide-and-conquer job*/
List computeForces(byte[] nodeId, int iteration, Bodies bodies)
{
  /*compute forces for subtree rooted at nodeId*/
}

/*guard function*/
boolean guard_computeForces(byte[] nodeId, int iteration, Bodies bodies)
{
  return (bodies.iteration+1 != iteration);
}
```

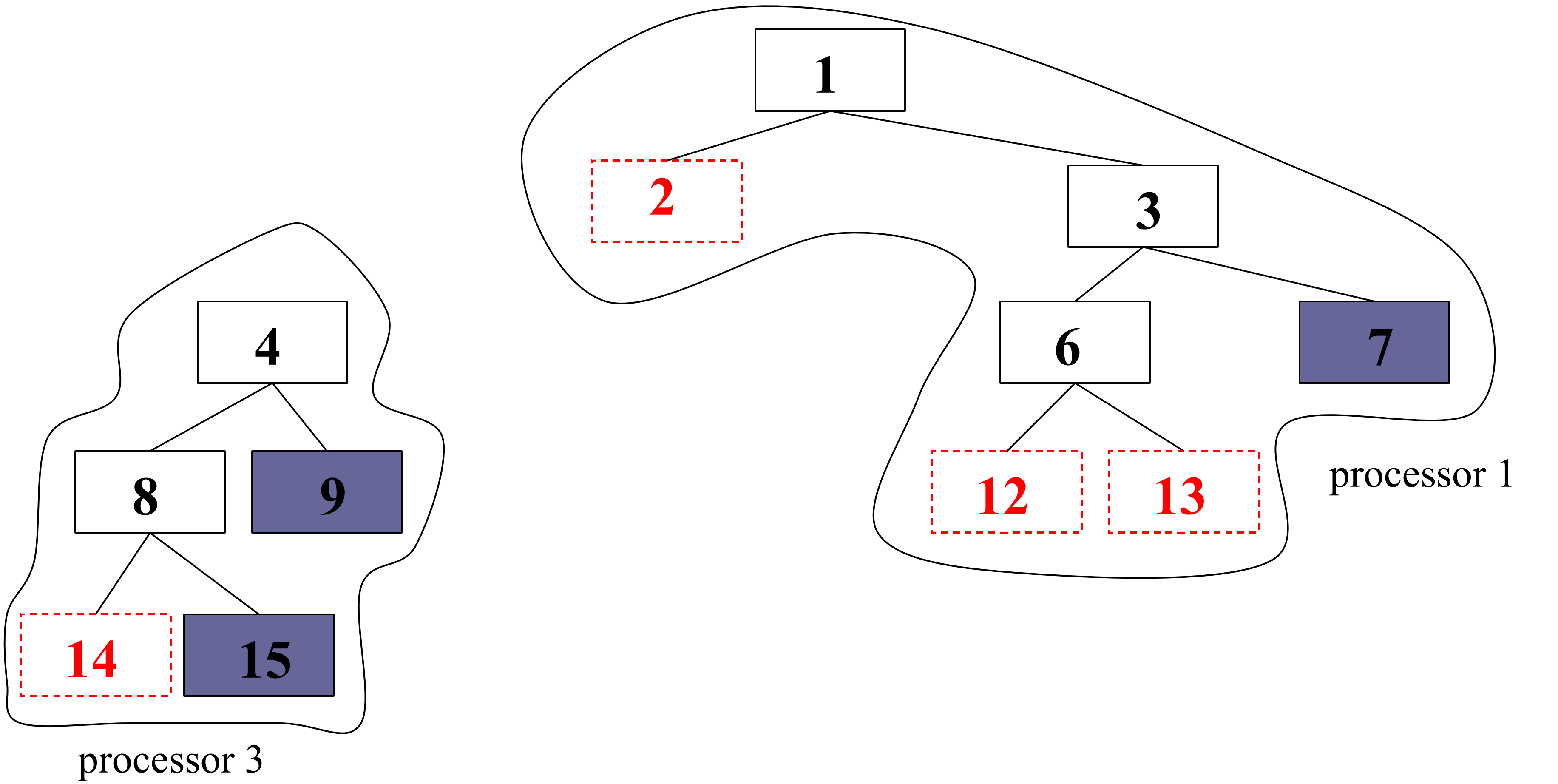
Handling orphan jobs - example



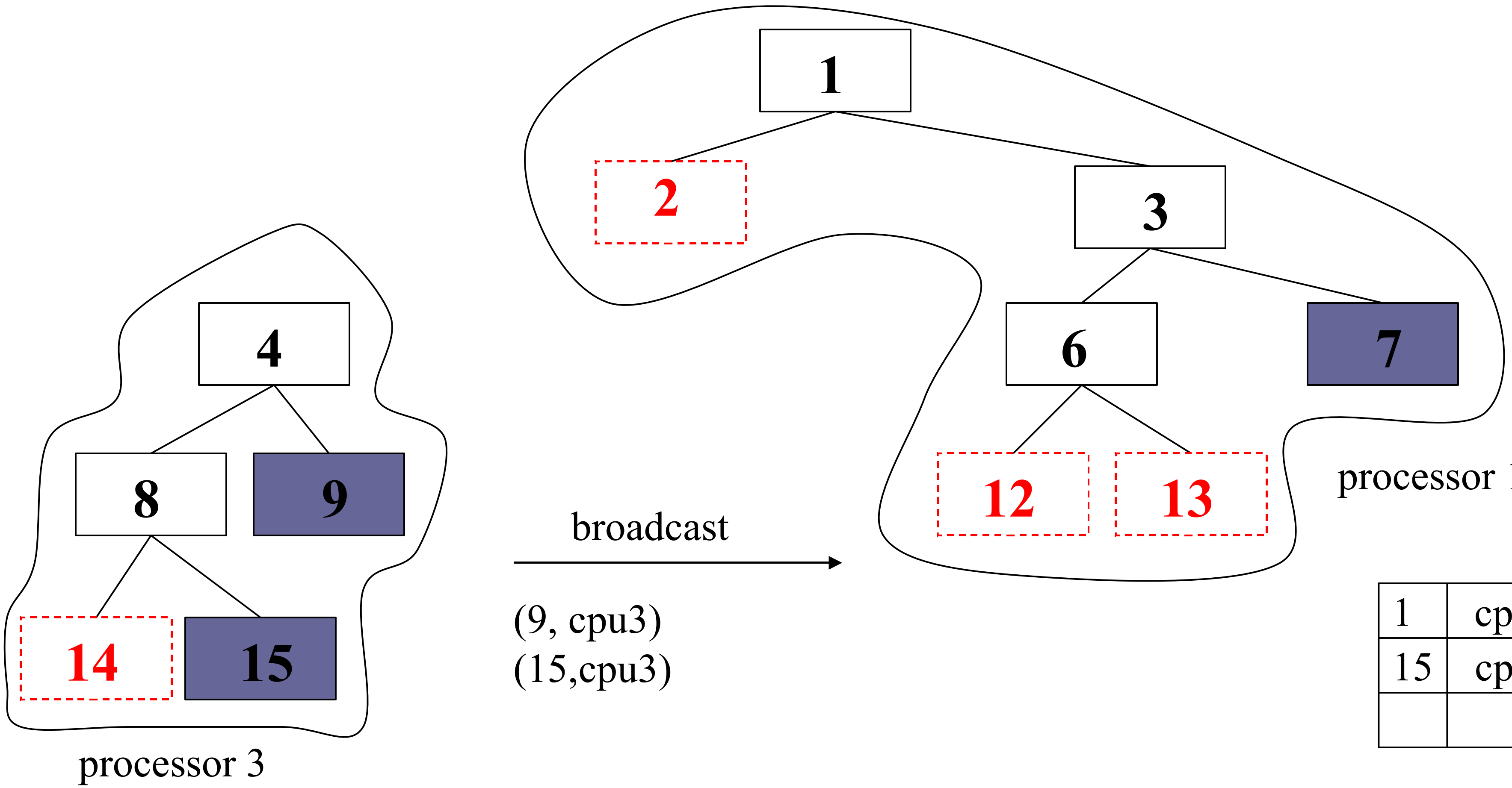
Handling orphan jobs - example



Handling orphan jobs - example

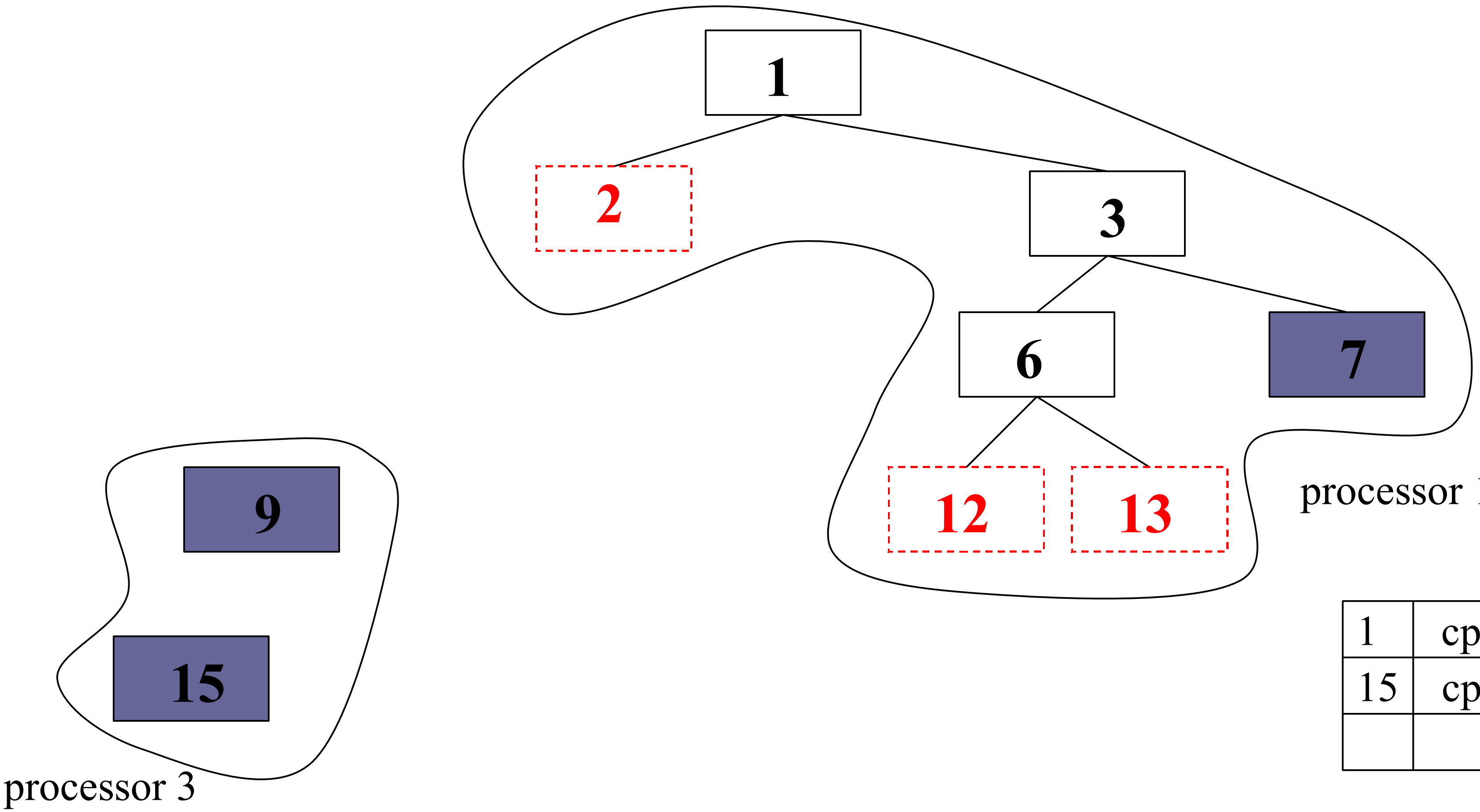


Handling orphan jobs - example



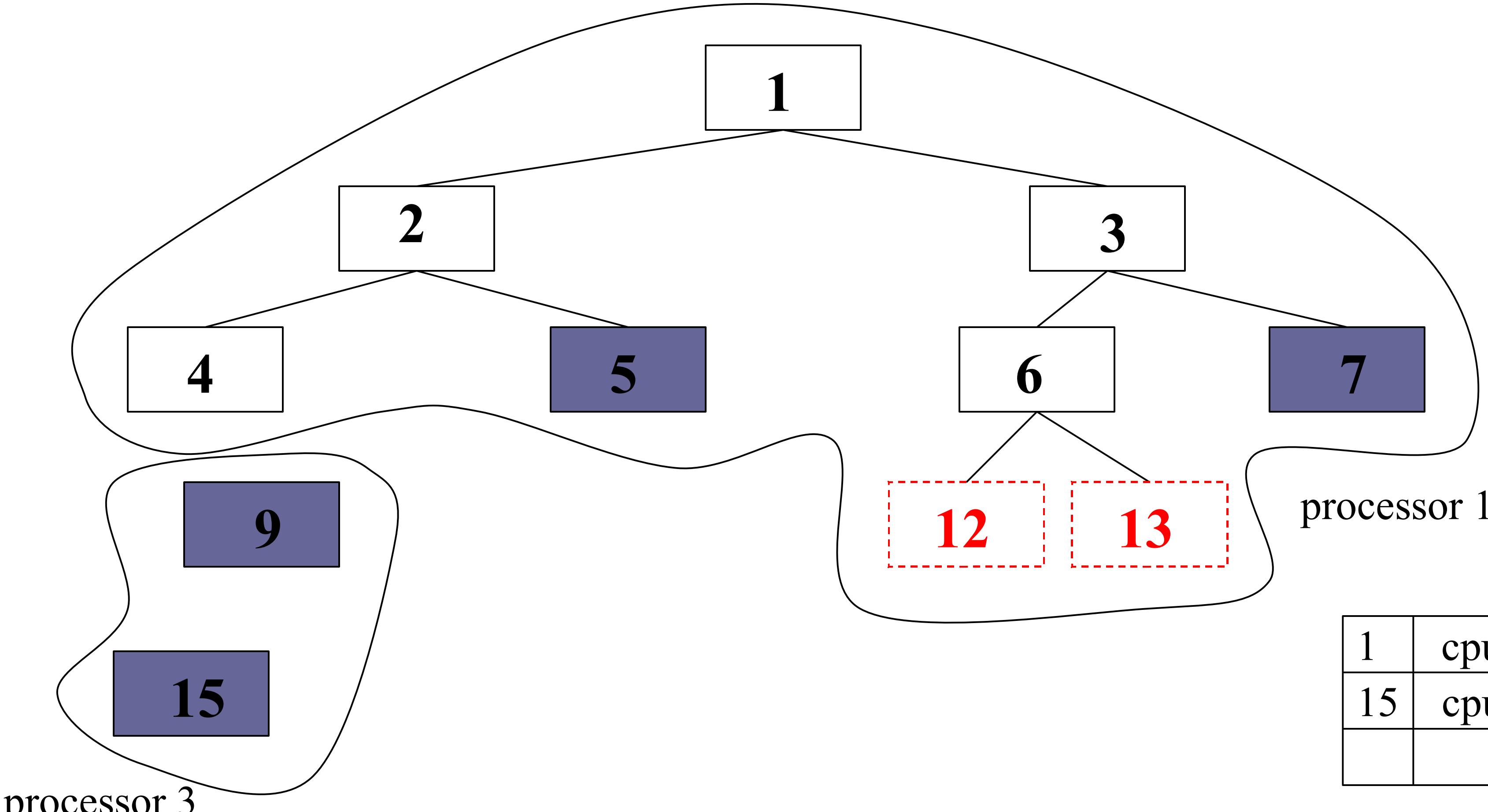
1	cpu3
15	cpu3

Handling orphan jobs - example



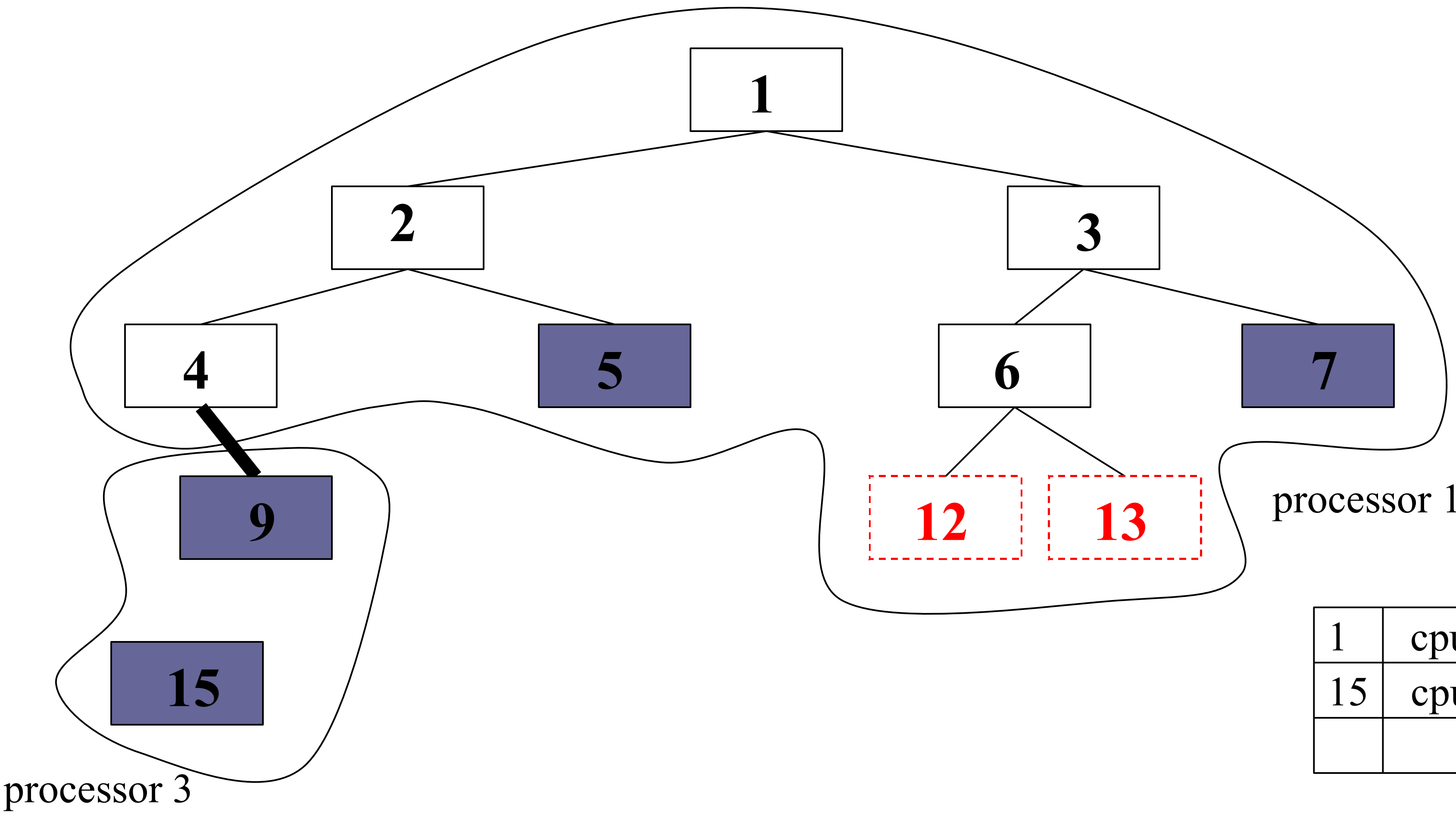
1	cpu3
15	cpu3

Handling orphan jobs - example



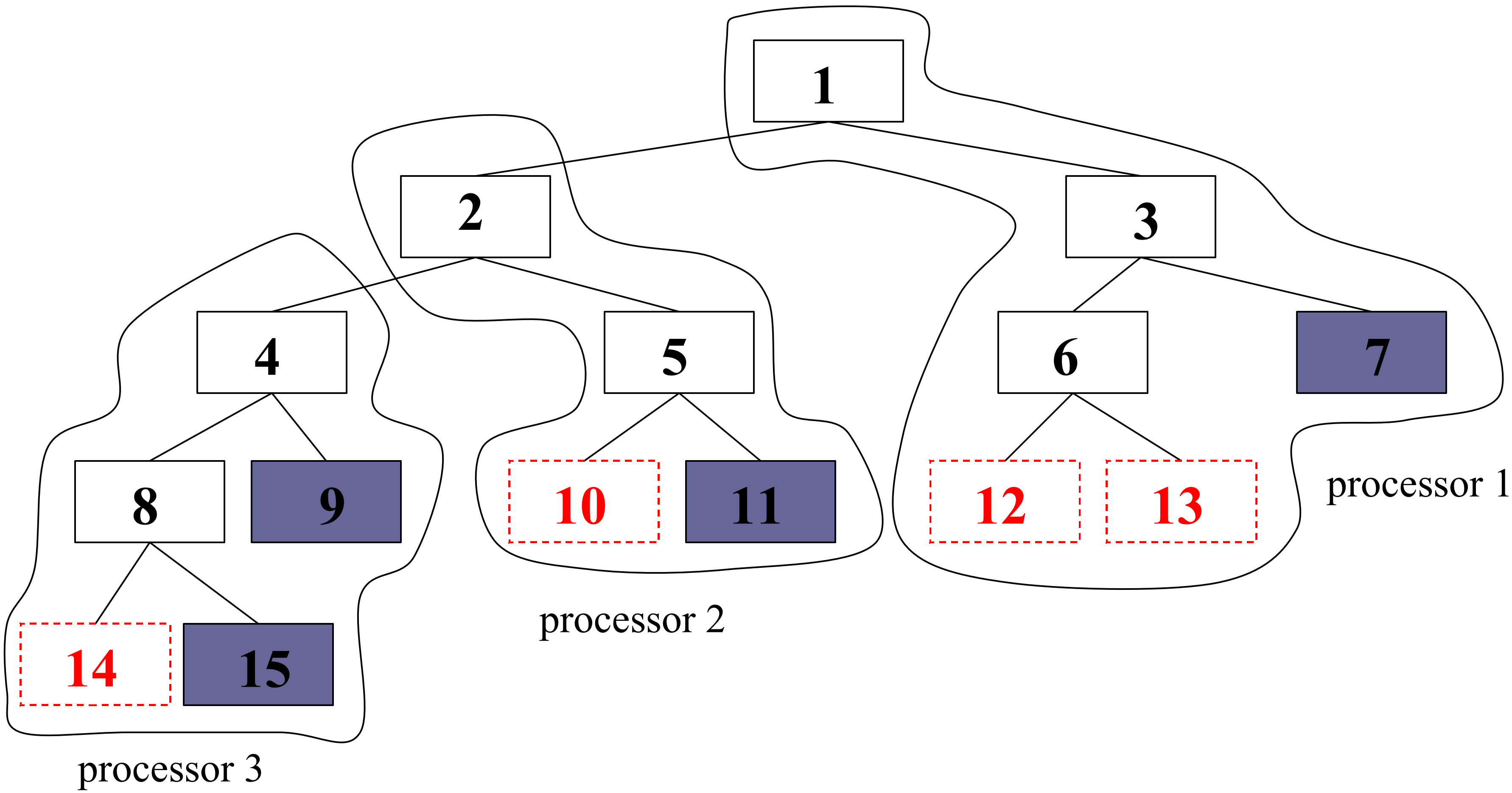
1	cpu3
15	cpu3

Handling orphan jobs - example

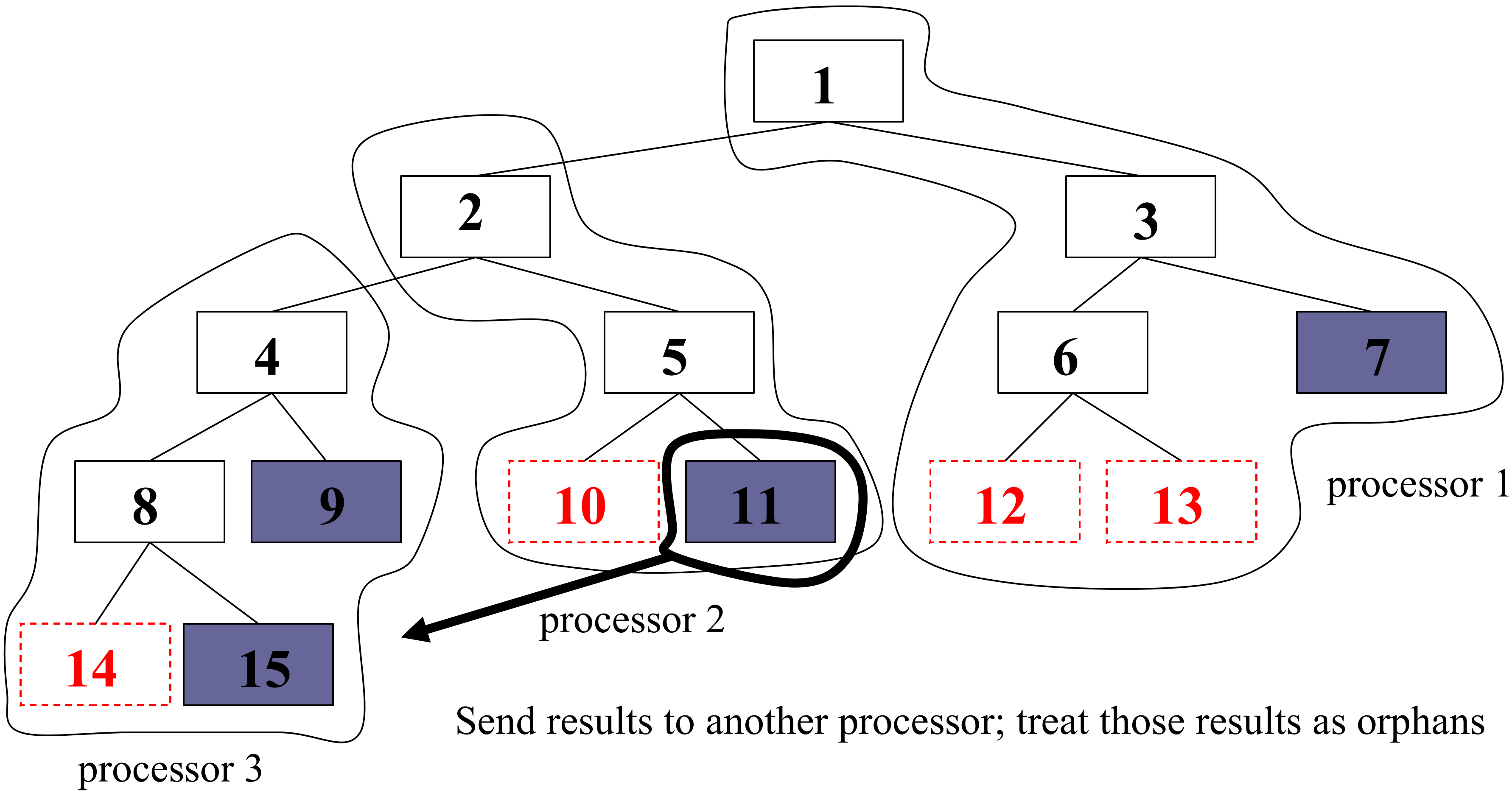


1	cpu3
15	cpu3

Processors leaving gracefully

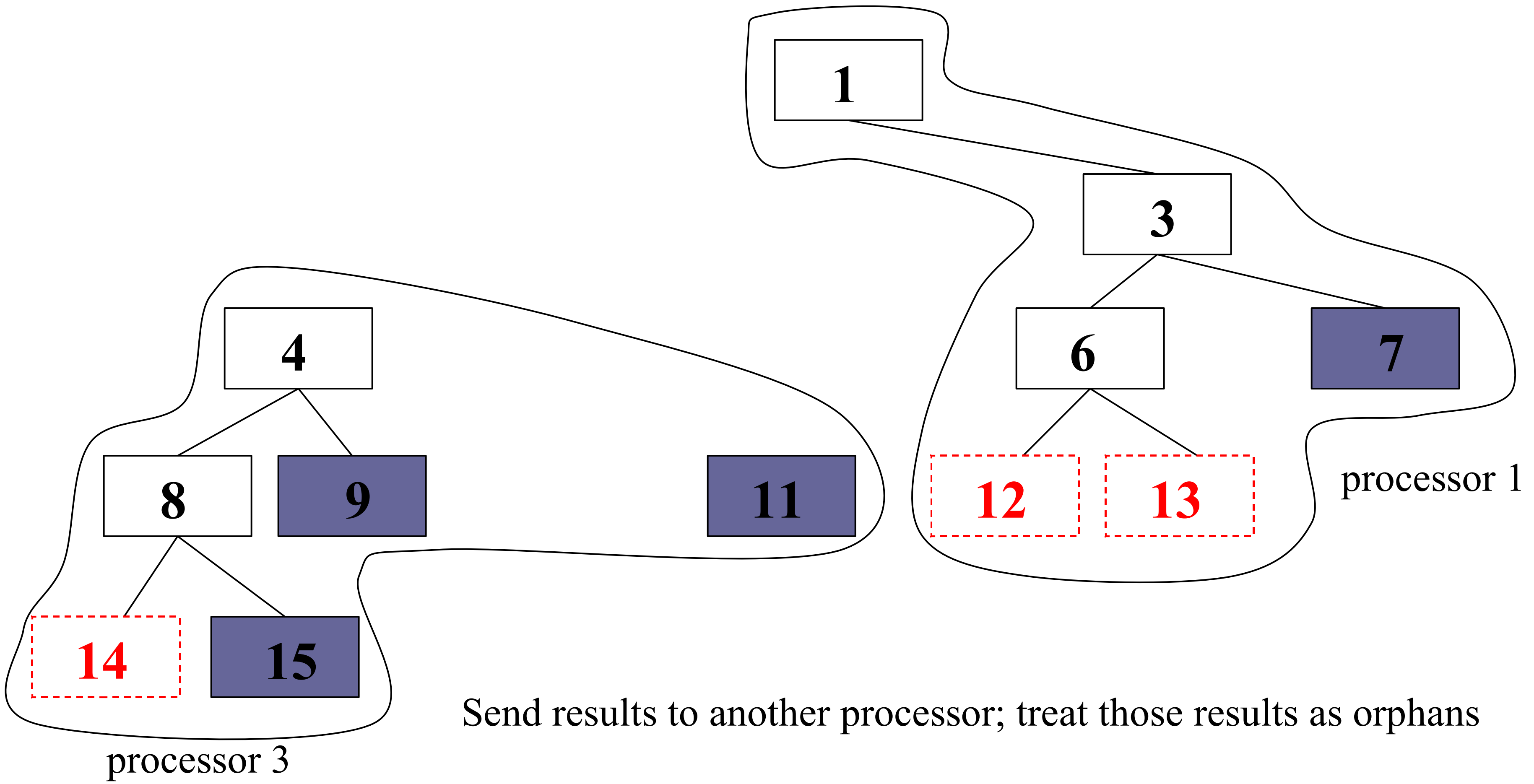


Processors leaving gracefully



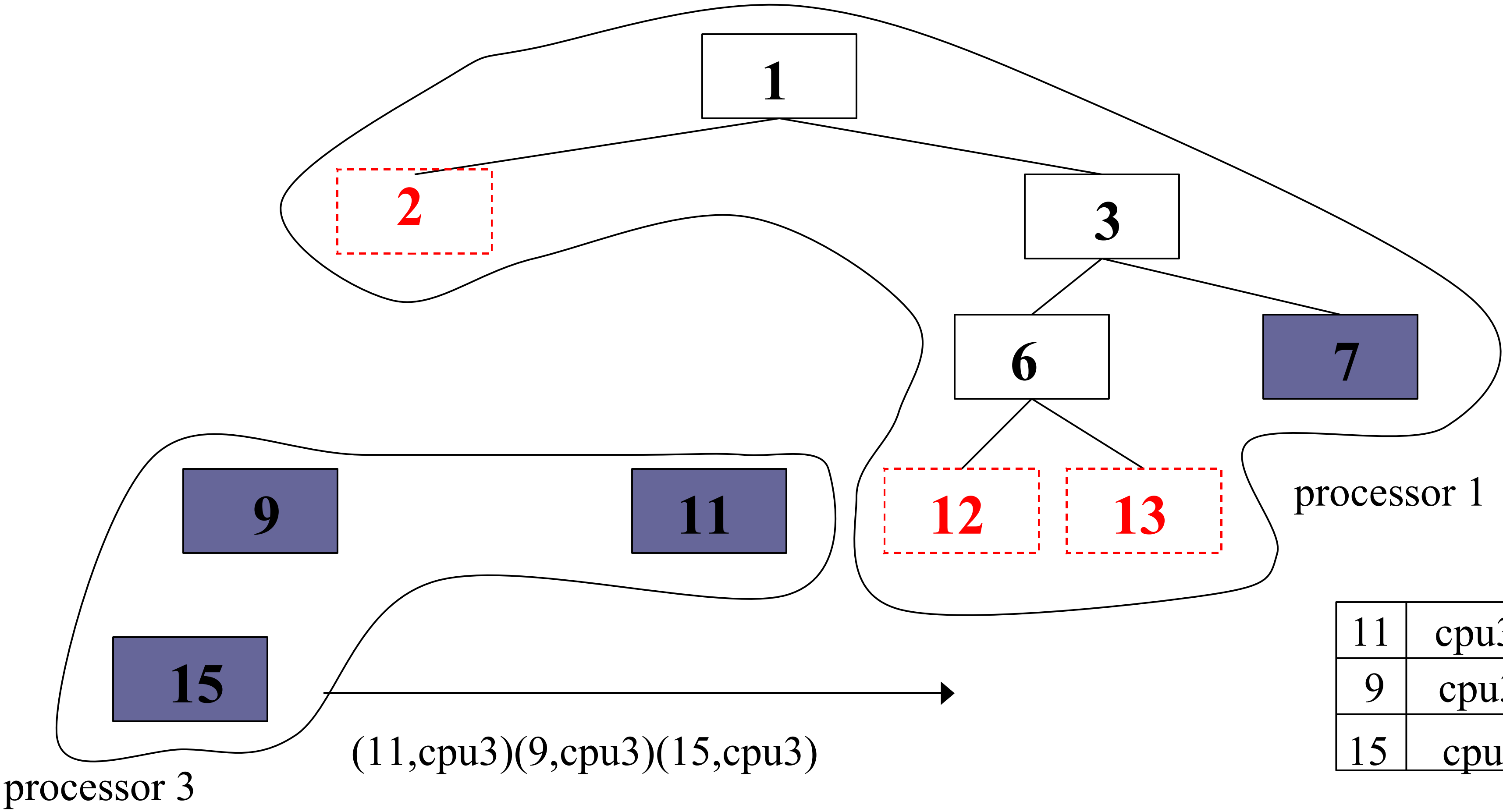
Send results to another processor; treat those results as orphans

Processors leaving gracefully



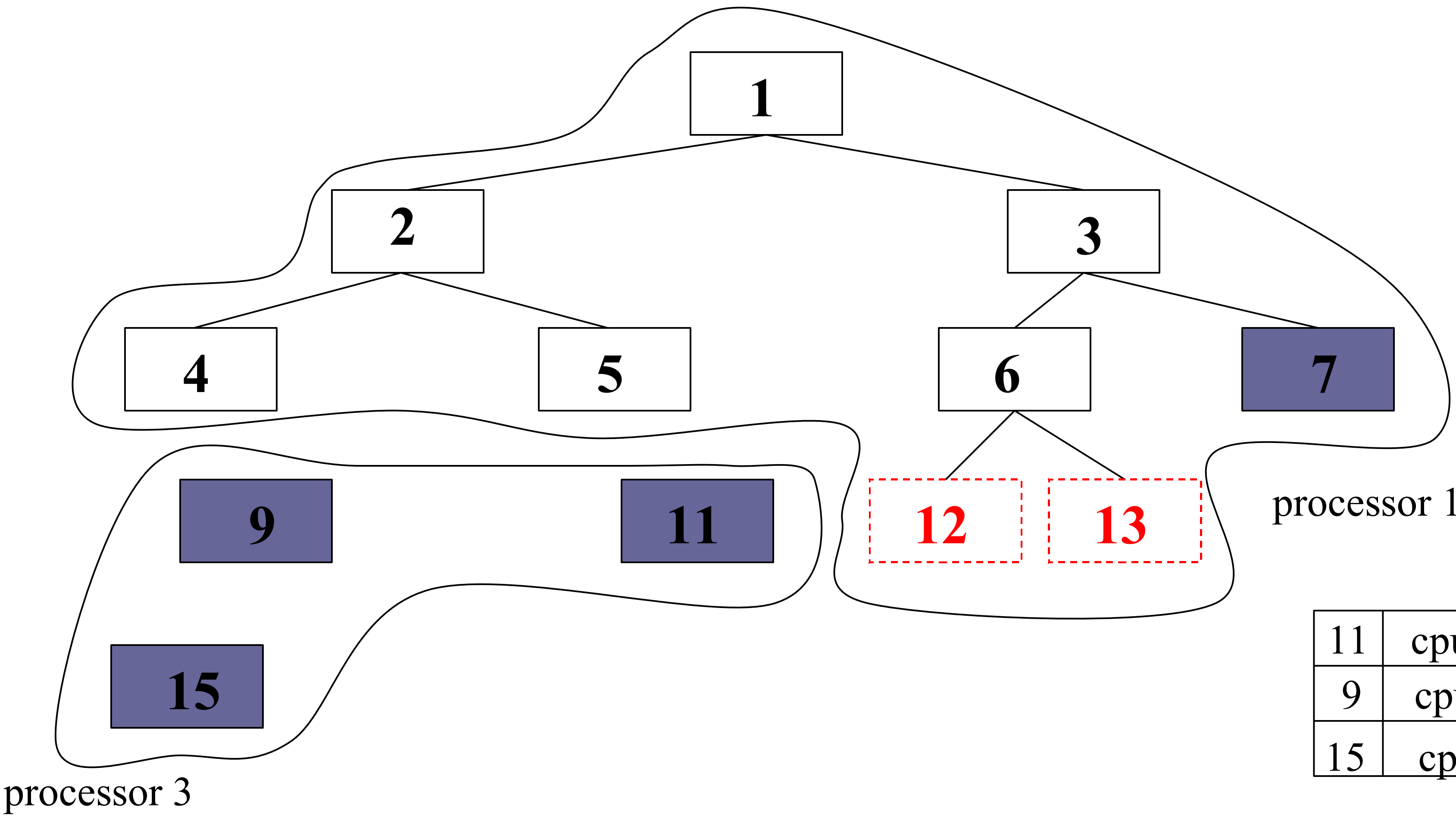
Send results to another processor; treat those results as orphans

Processors leaving gracefully



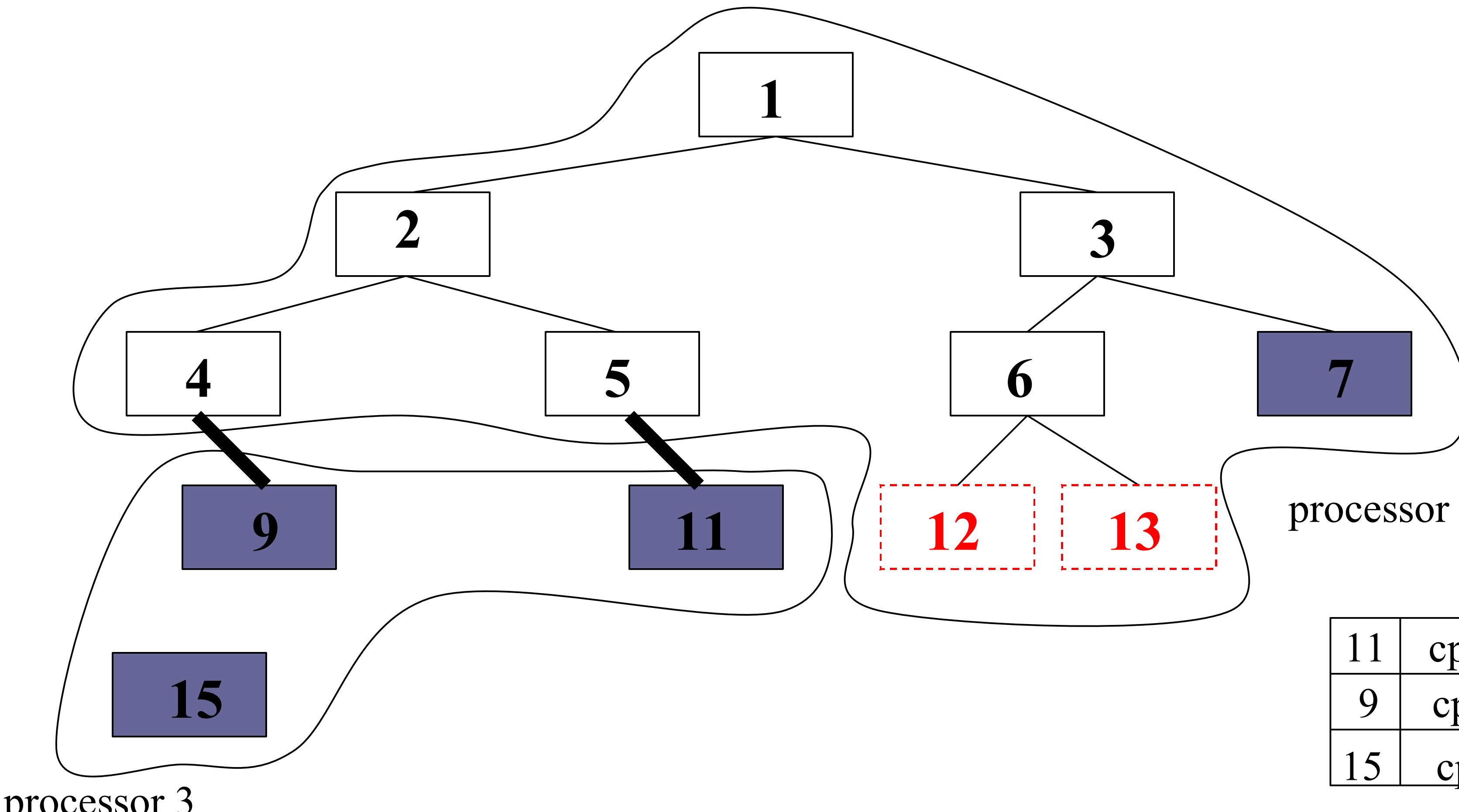
11	cpu3
9	cpu3
15	cpu3

Processors leaving gracefully



11	cpu3
9	cpu3
15	cpu3

Processors leaving gracefully



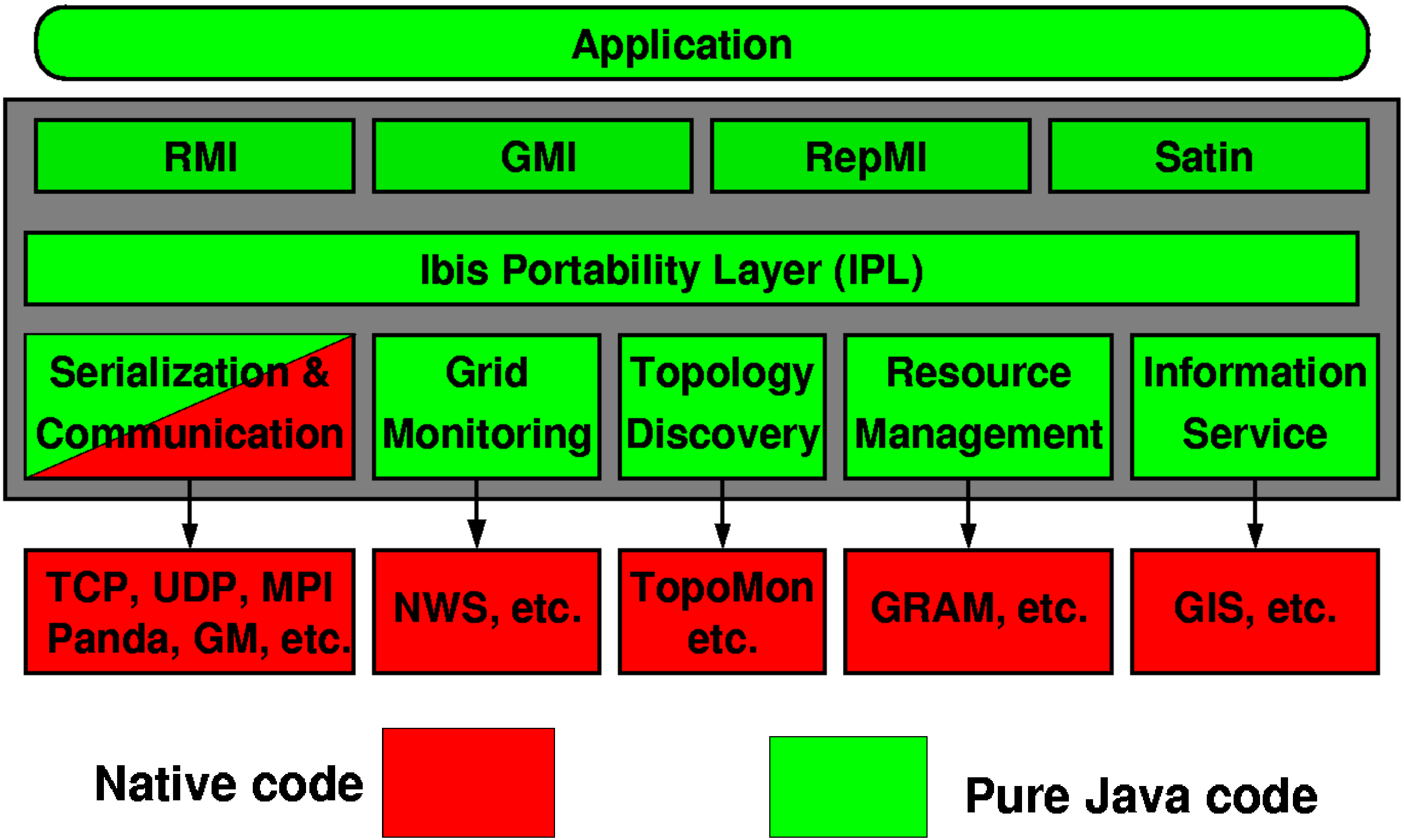
11	cpu3
9	cpu3
15	cpu3

The Ibis system

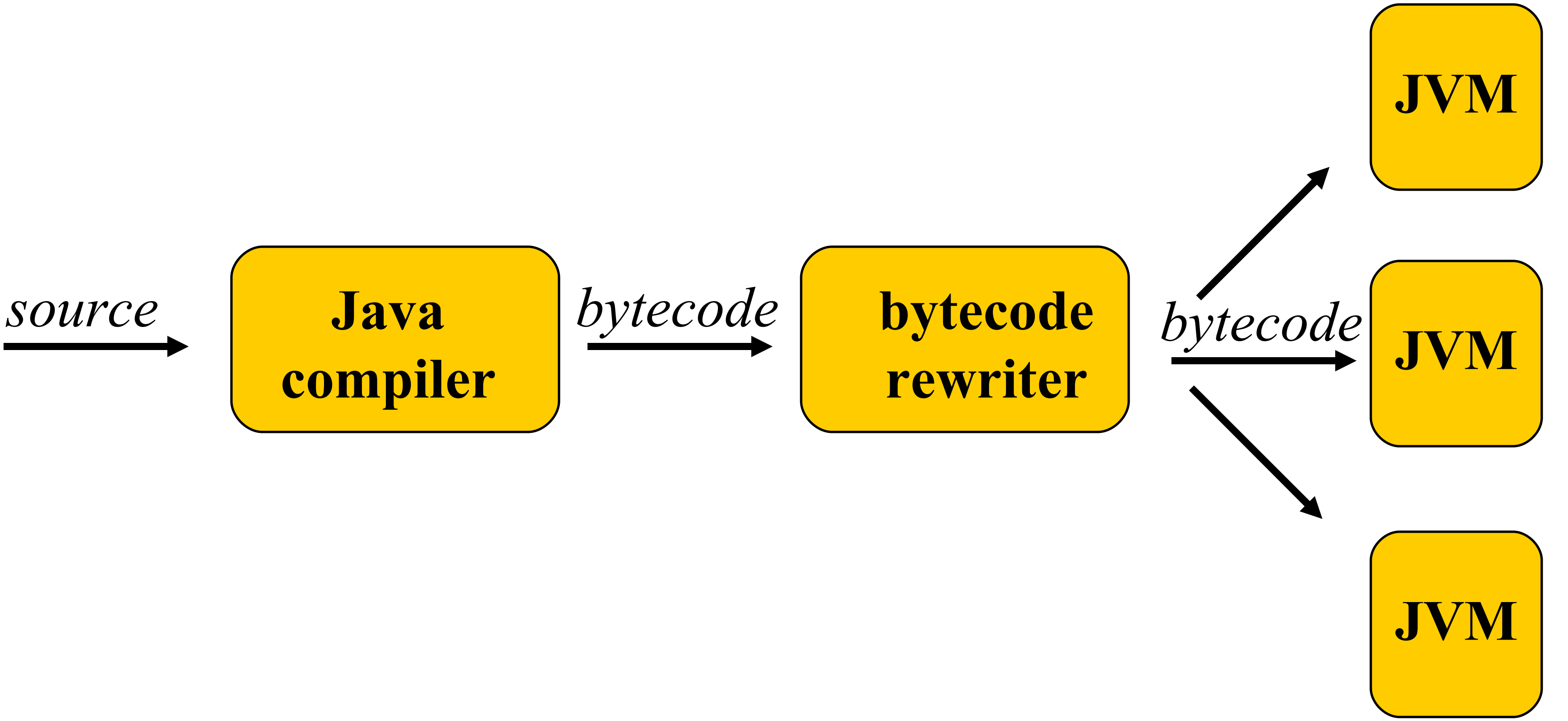
- Java-centric => portability
  - „write once, run anywhere”
- Efficient communication
  - Efficient pure Java implementation
  - Optimized solutions for special cases with native code
- High level programming models:
  - Divide & Conquer (Satin)
  - Remote Method Invocation (RMI)
  - Replicated Method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

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

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

Compiling Satin programs



Executing Satin programs

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

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

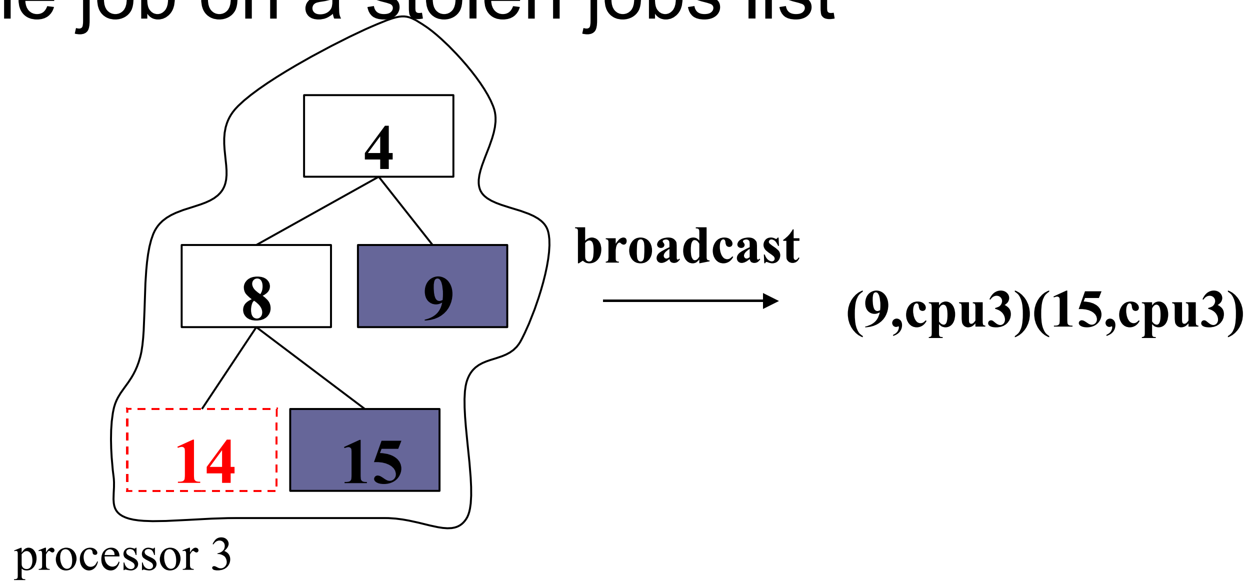
- Cluster-aware Random Stealing (CRS) [van Nieuwpoort et al., PPoPP 2001]
  - When idle:
    - Send asynchronous steal request to random node in different cluster
    - In the meantime steal locally (synchronously)
    - Only one wide-area steal request at a time

Configuration

Location	Type	OS	CPU	CPUs
Amsterdam, The Netherlands	Cluste r	Linux	Pentium-3	8 x 1
Amsterdam, The Netherlands	SMP	Solaris	Sparc	1 x 2
Brno, Czech Republic	Cluste r	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

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



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.

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

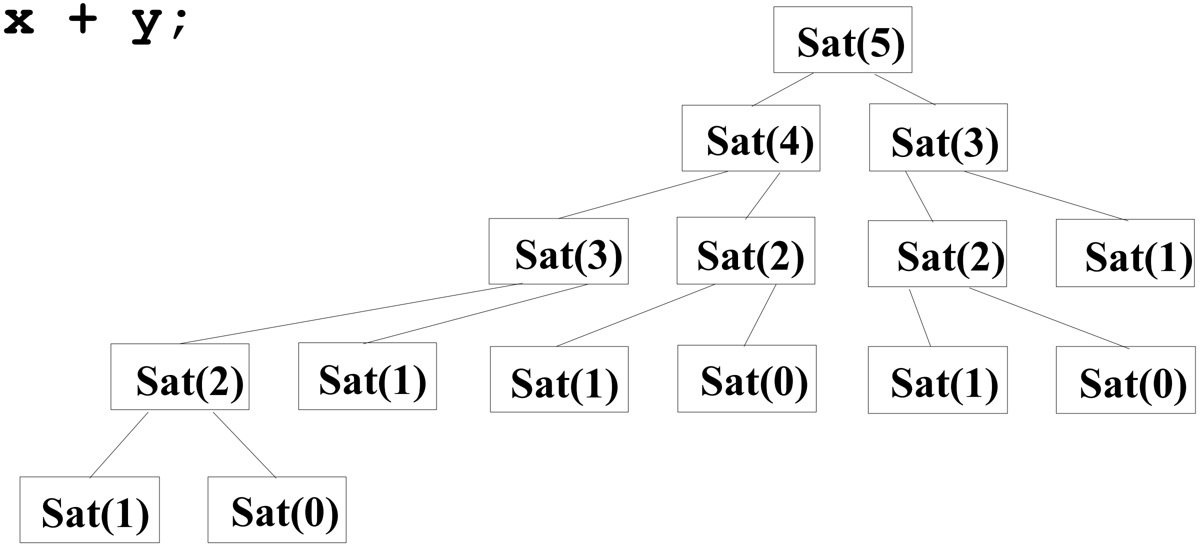
Shared Objects - example

```
public interface BarnesHutInterface extends WriteMethods {
    void computeForces(
```

Satin “Hello world”: Satonacci

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

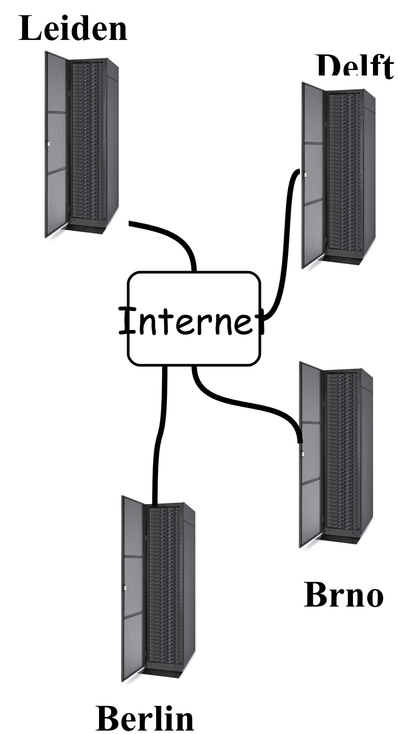
Single-threaded Java



Parallelizing Satonacci

```
public interface SatInter extends
ibis.satin.Spawnable {
      public int Sat (int n);
}

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



Satonacci – c.d.