Satin: Simple and Efficient Java-based Grid Programming

Rob V. van Nieuwpoort
Jason Maassen
Thilo Kielmann
Henri E. Bal

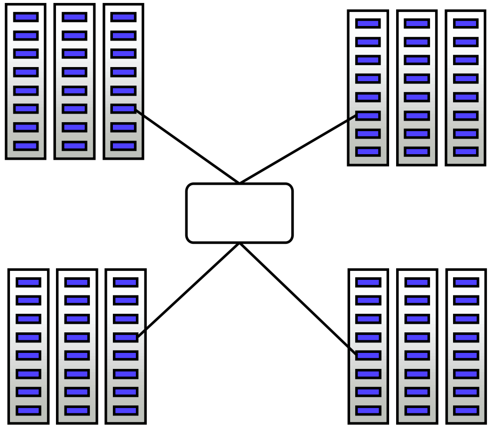
vrije Universiteit Amsterdam

AGridM 2003, September 28, New Orleans

Distributed supercomputing

- Parallel processing on geographically distributed computing systems (grids)
- Our goals:
  - Don’t use individual supercomputers / clusters, but combine multiple systems
  - Provide high-level programming support

Optimizing for the grid



- Grids usually are hierarchical
  - Collections of clusters, supercomputers
  - Fast local links, slow wide-area links
- Can optimize algorithms to exploit this hierarchy
  - Message combining + latency hiding on wide-area links
  - Collective operations for wide-area systems (MagPIe)
  - Successful for many applications
- This talk: Load balancing divide-and-conquer applications

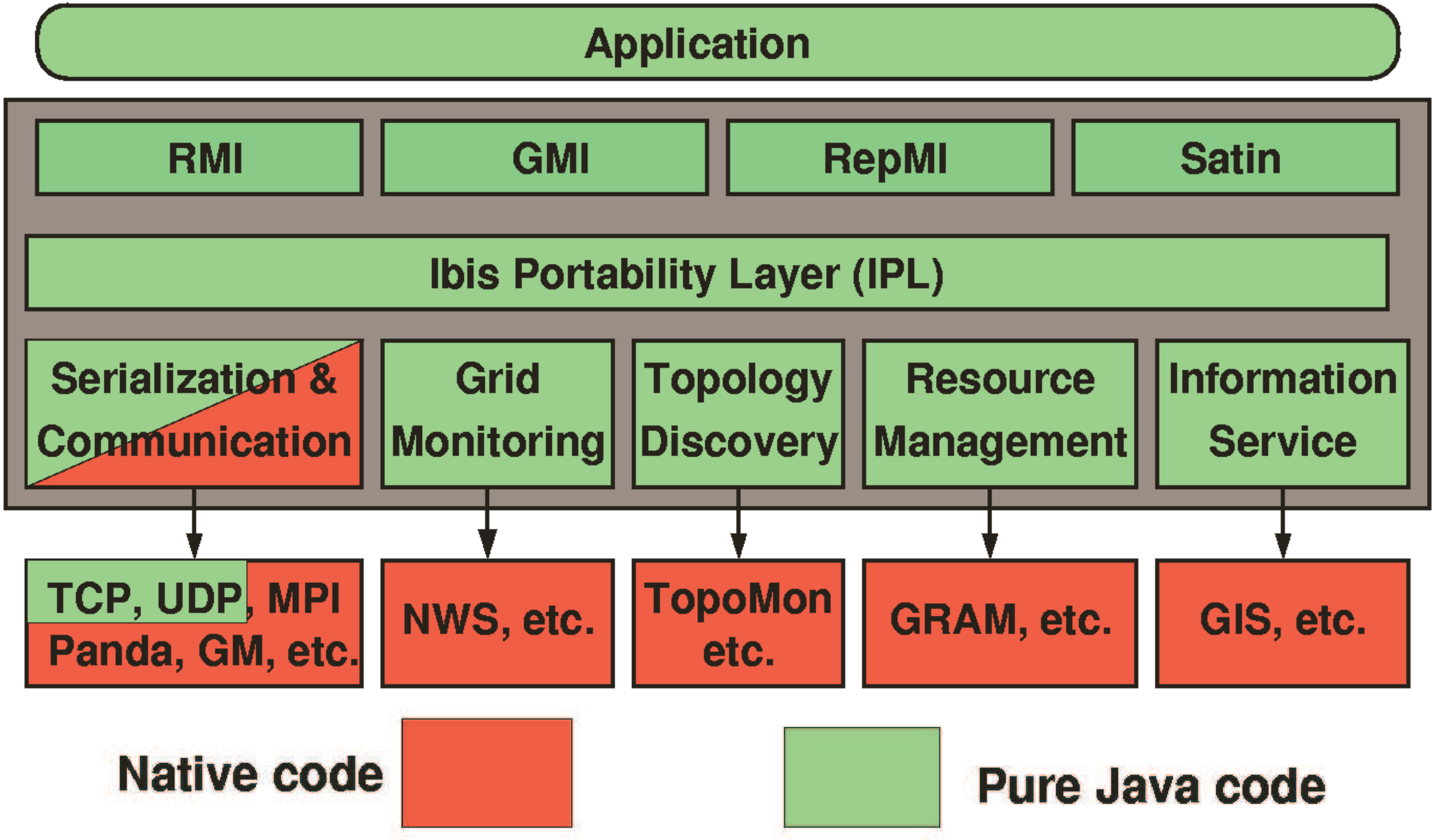
Outline

- Ibis: Java-based grid middleware
- Ibis performance
- Divide-and-conquer with Satin
- Load Balancing
- Case study on a real heterogeneous grid

The Ibis system

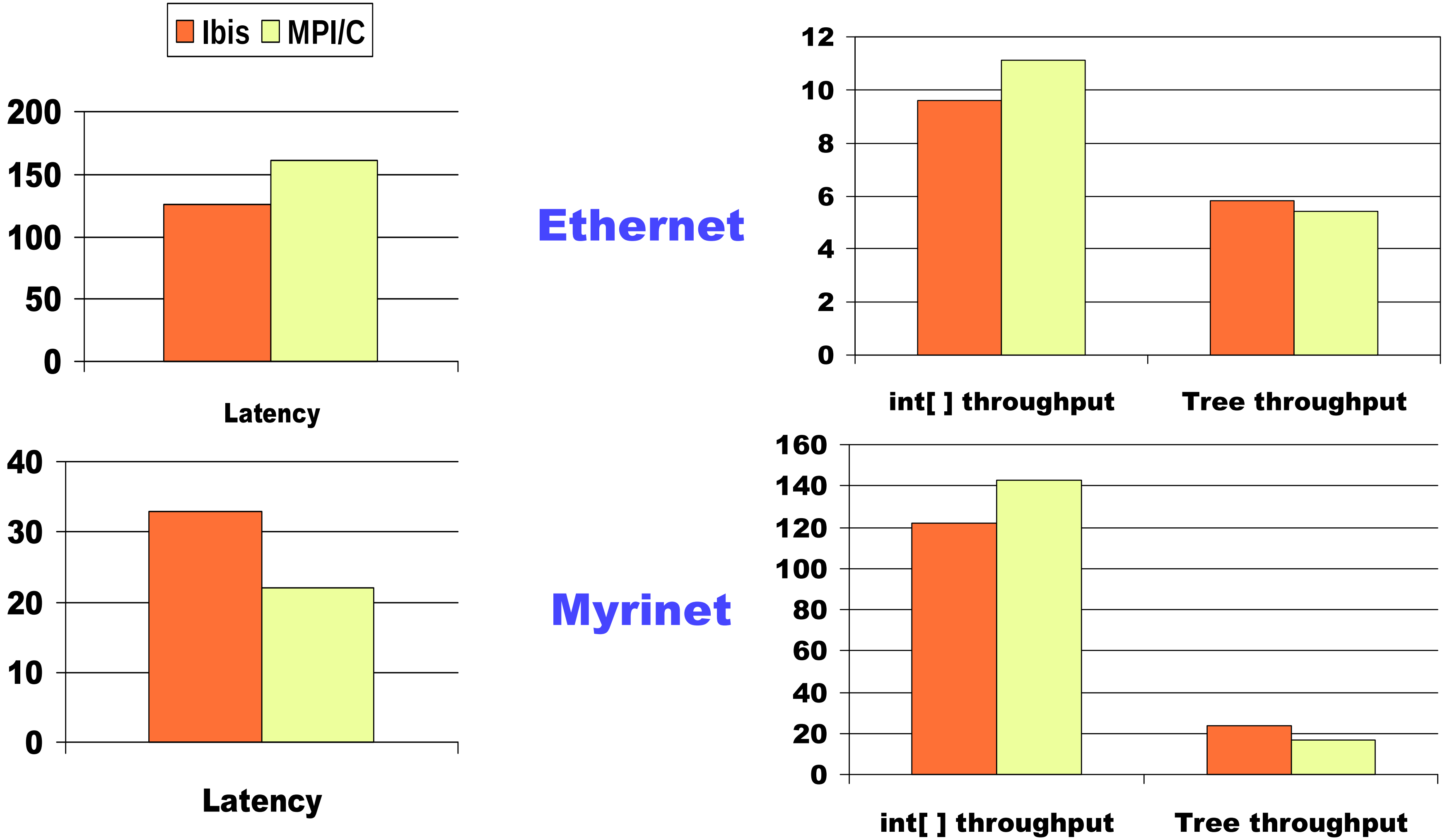
- High-level & efficient programming support for distributed supercomputing on heterogeneous grids
- Strategy:
  - Reasonably efficient "*Write once, run everywhere*" solution in 100% Java
  - High performance (native) solutions for special cases
- But using a single interface for this!

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

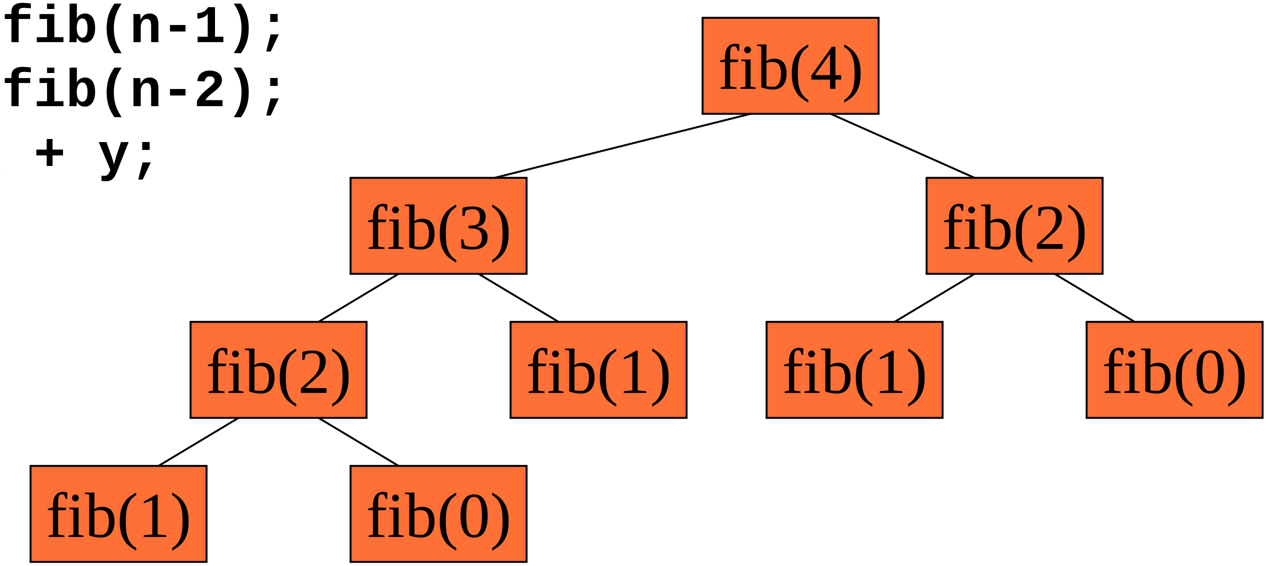
Ibis performance



Latency (µs) & throughput (MB/s), on 1 GHz Pentium-IIIs

Divide-and-conquer example

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

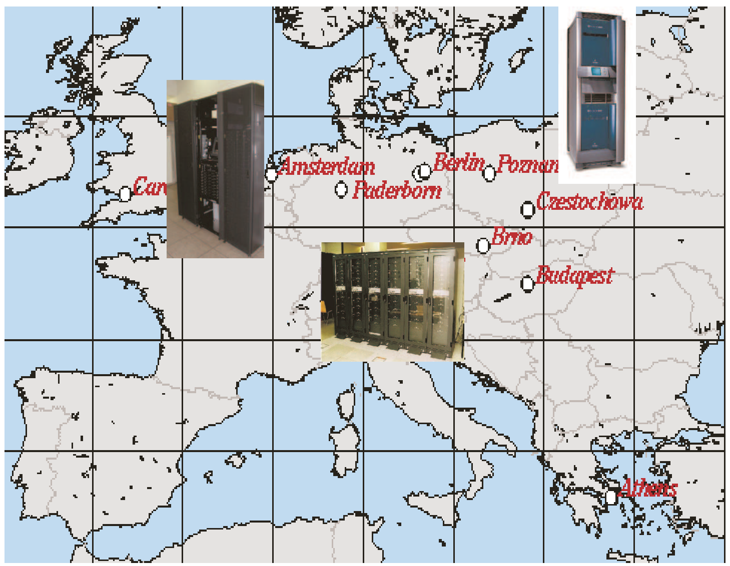


Single-threaded Java

Satin example

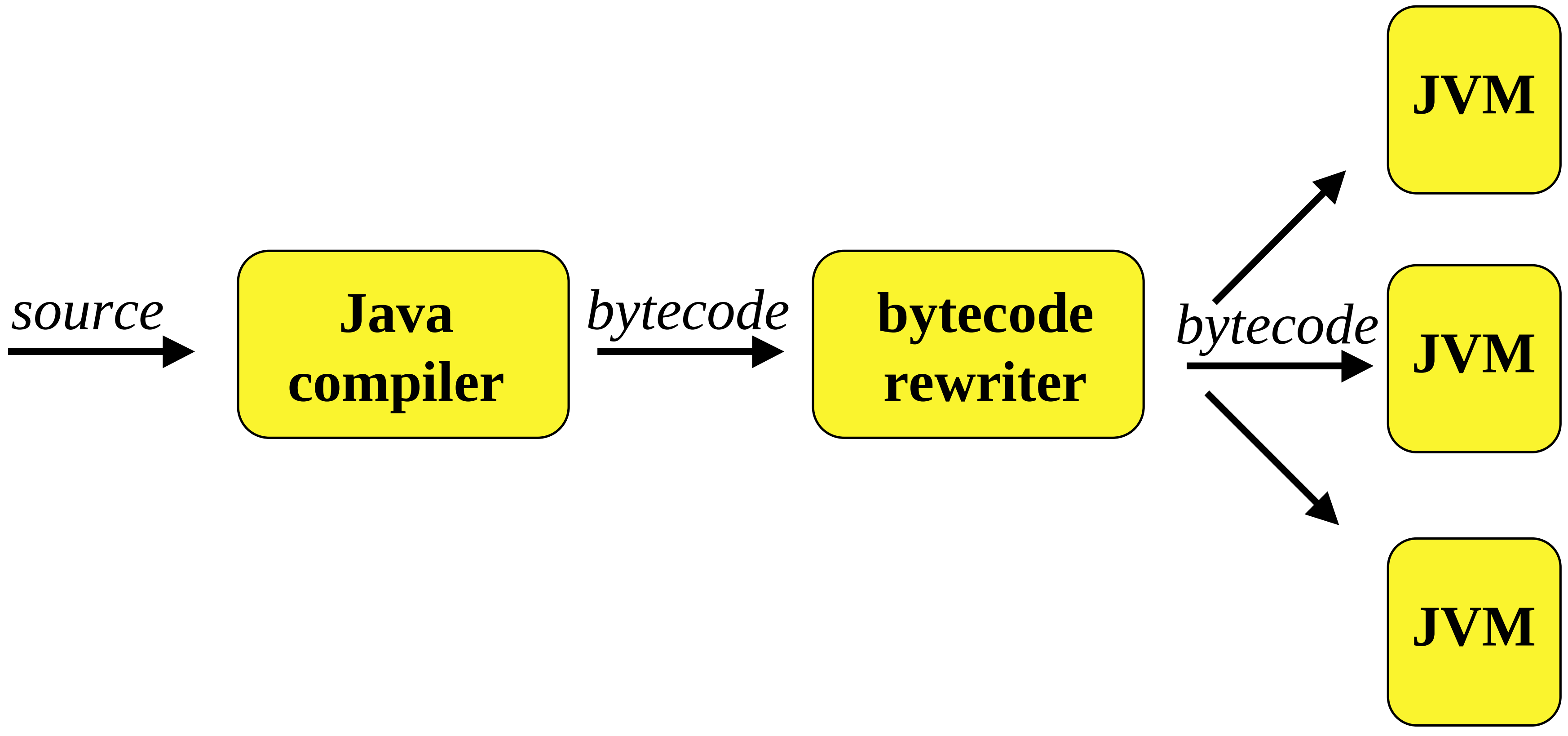
```
interface FibInter
   extends ibis.satin.Spawnable {
      public int fib(int n);
}

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



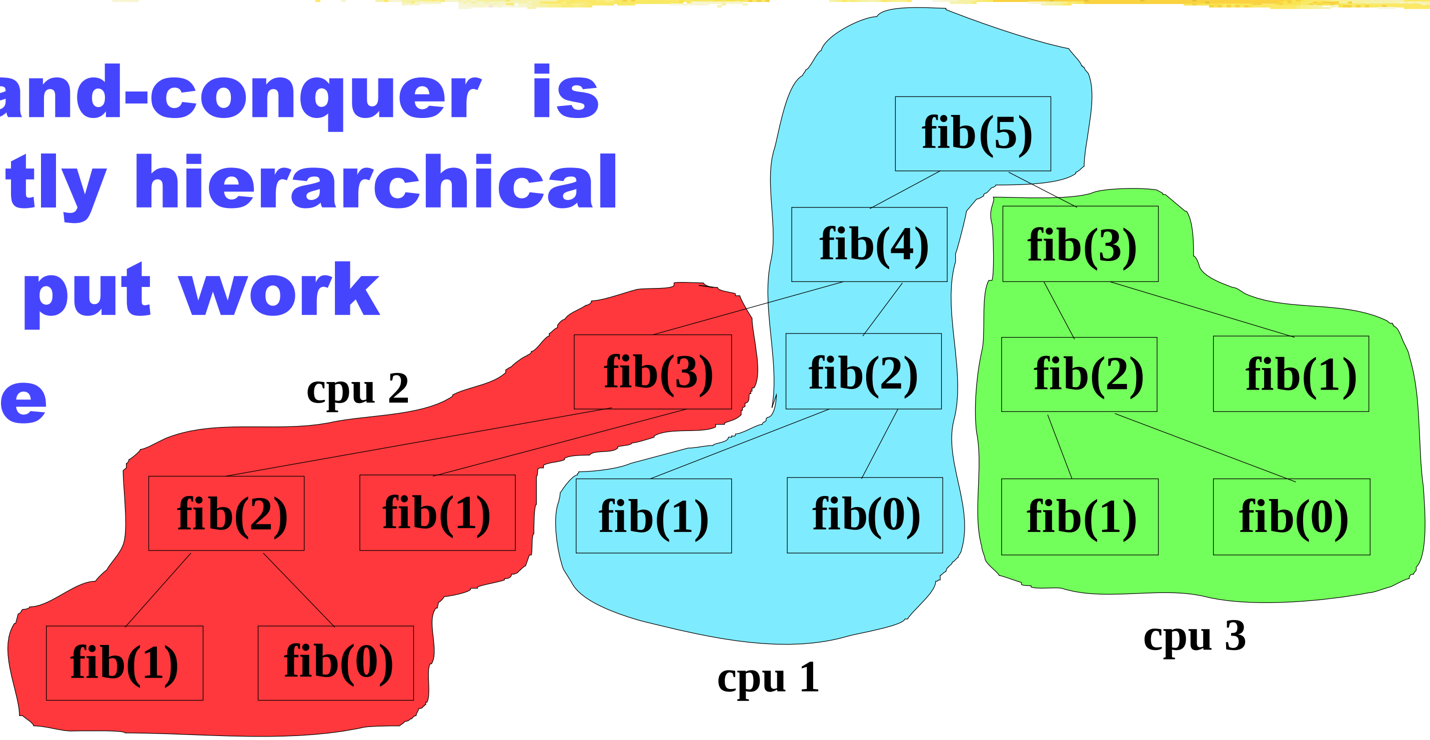
Parallel Satin

Compiling Satin programs



Parallel divide-and-conquer

- Divide-and-conquer is inherently hierarchical
- Spawn: put work in queue

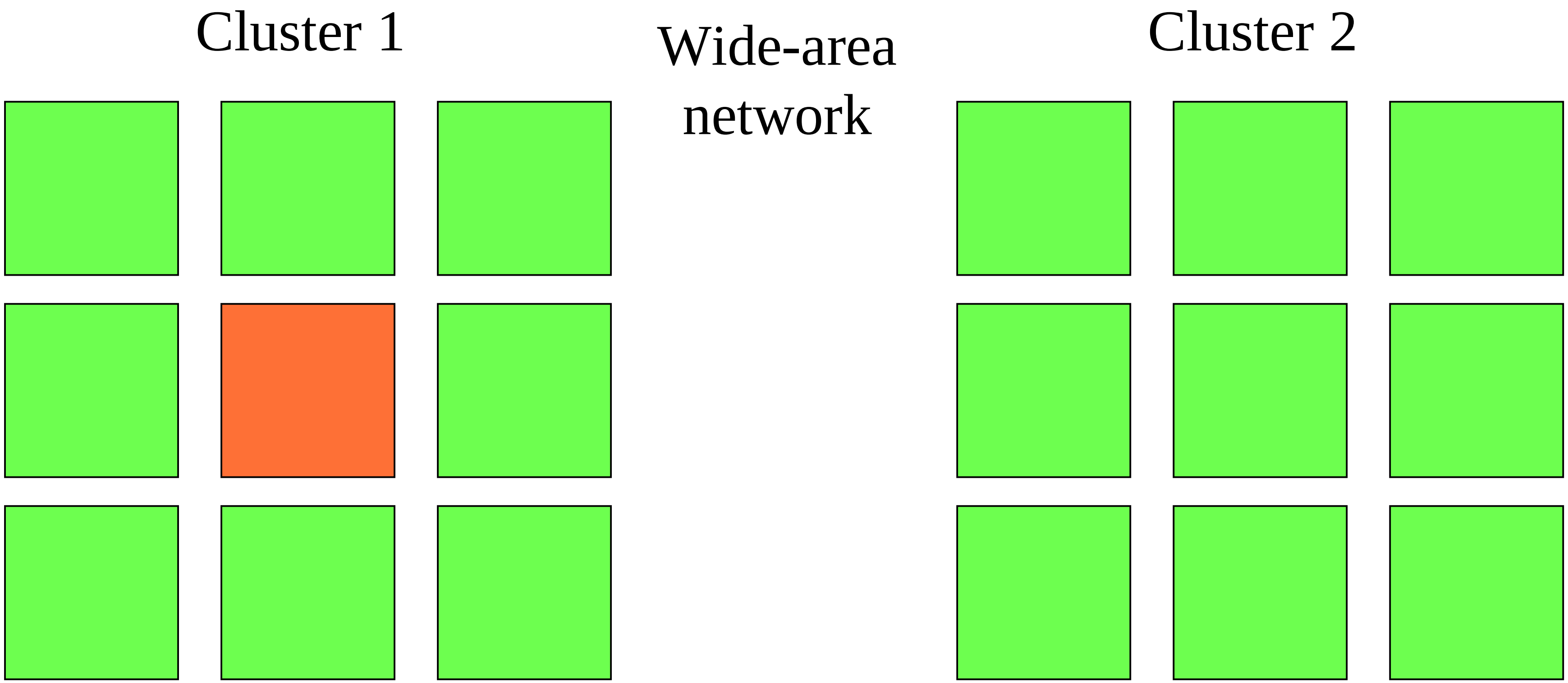


- Sync:
  - Run work from queue
  - Do load balancing (moving work between queues)

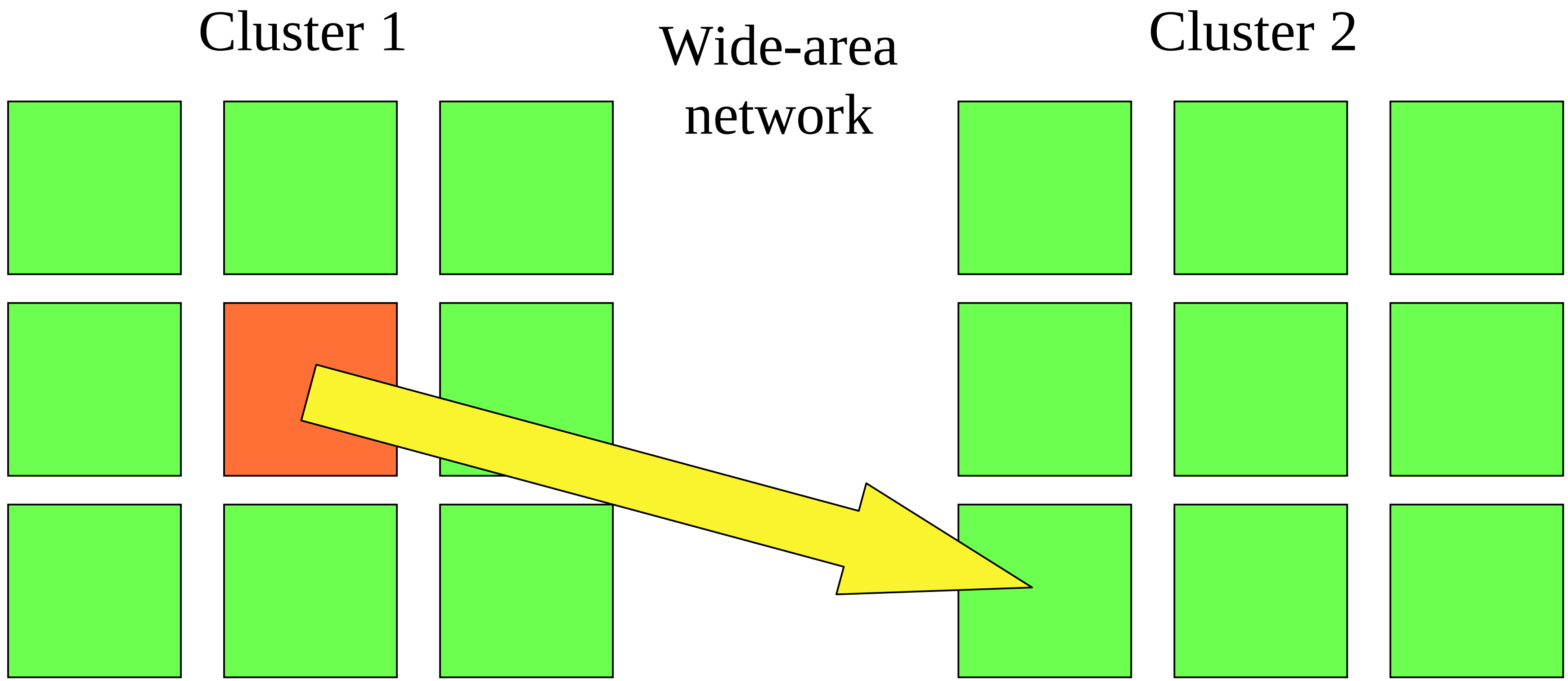
Load balancing Satin programs

- Random stealing (RS) has been proven to be optimal in space, time and communication
- However, RS does not consider communication costs
- Fine in a cluster : costs are the same, regardless of the target
- Does not work in grid environment
  - With 4 clusters, 75% of steals are wide-area
  - Synchronous algorithm

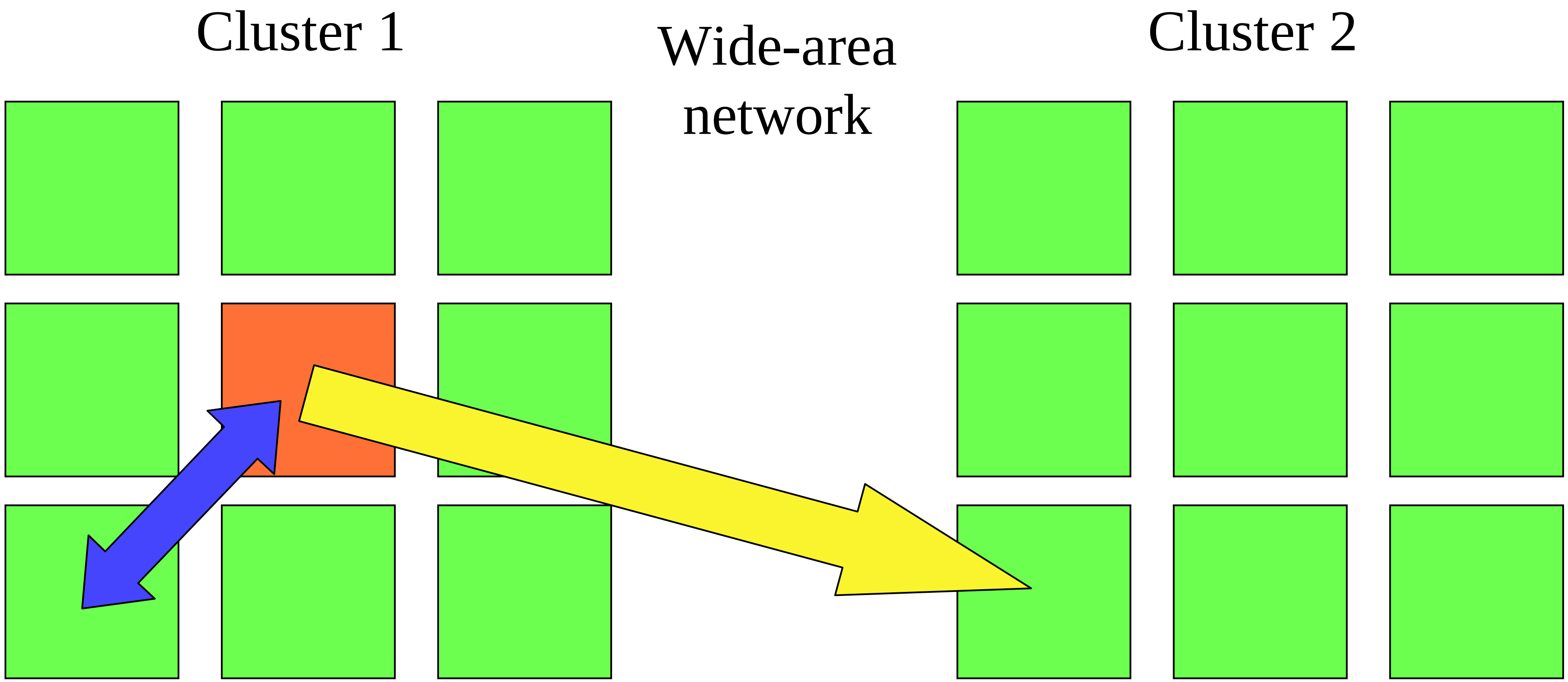
Cluster-aware Random Stealing (CRS)



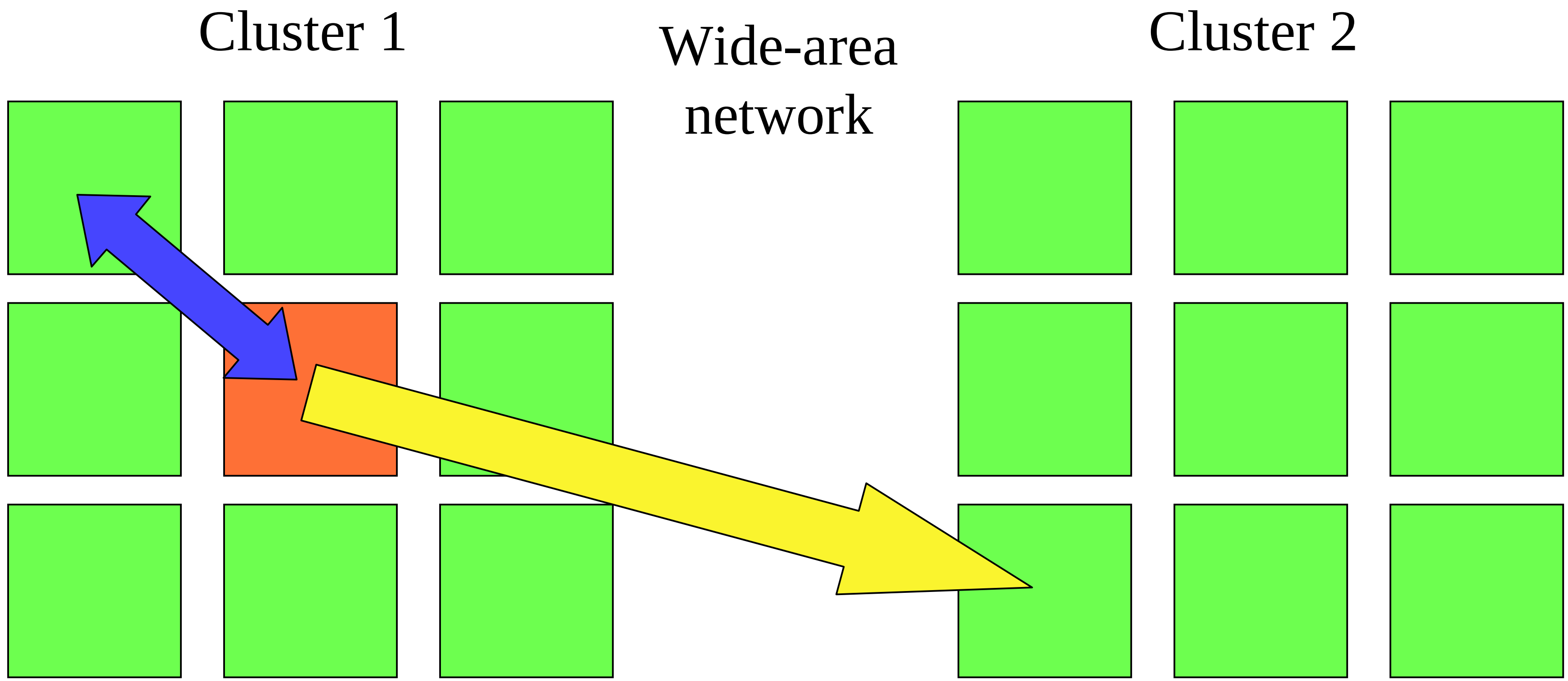
Cluster-aware Random Stealing (CRS)



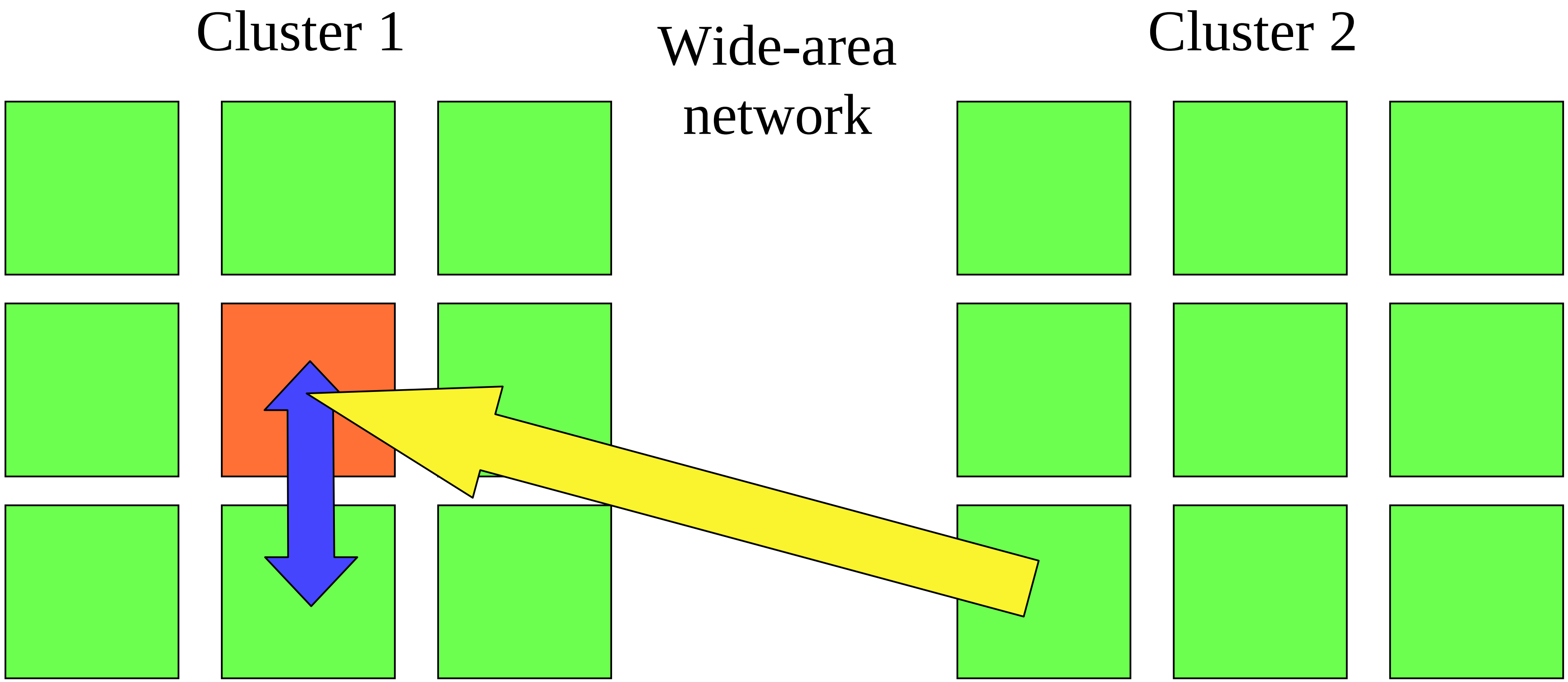
Cluster-aware Random Stealing (CRS)



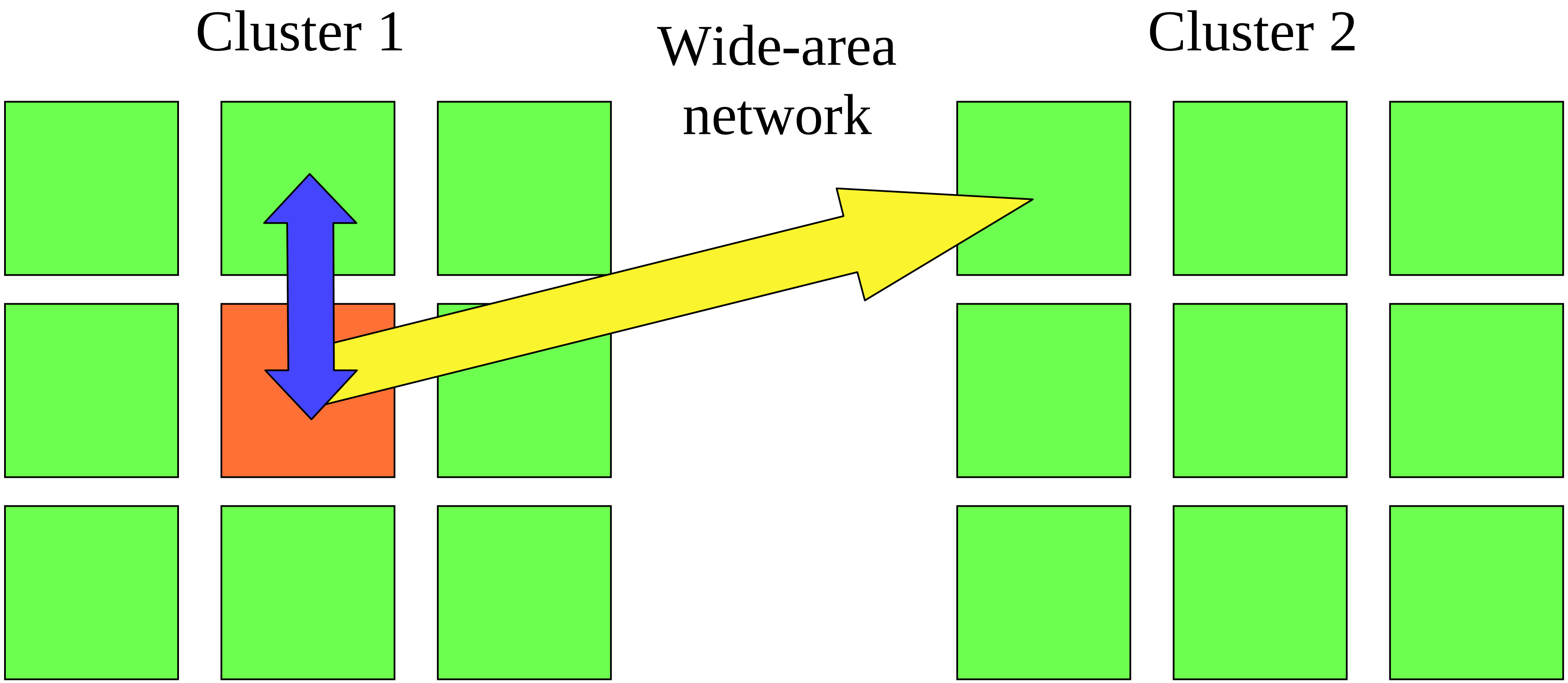
Cluster-aware Random Stealing (CRS)



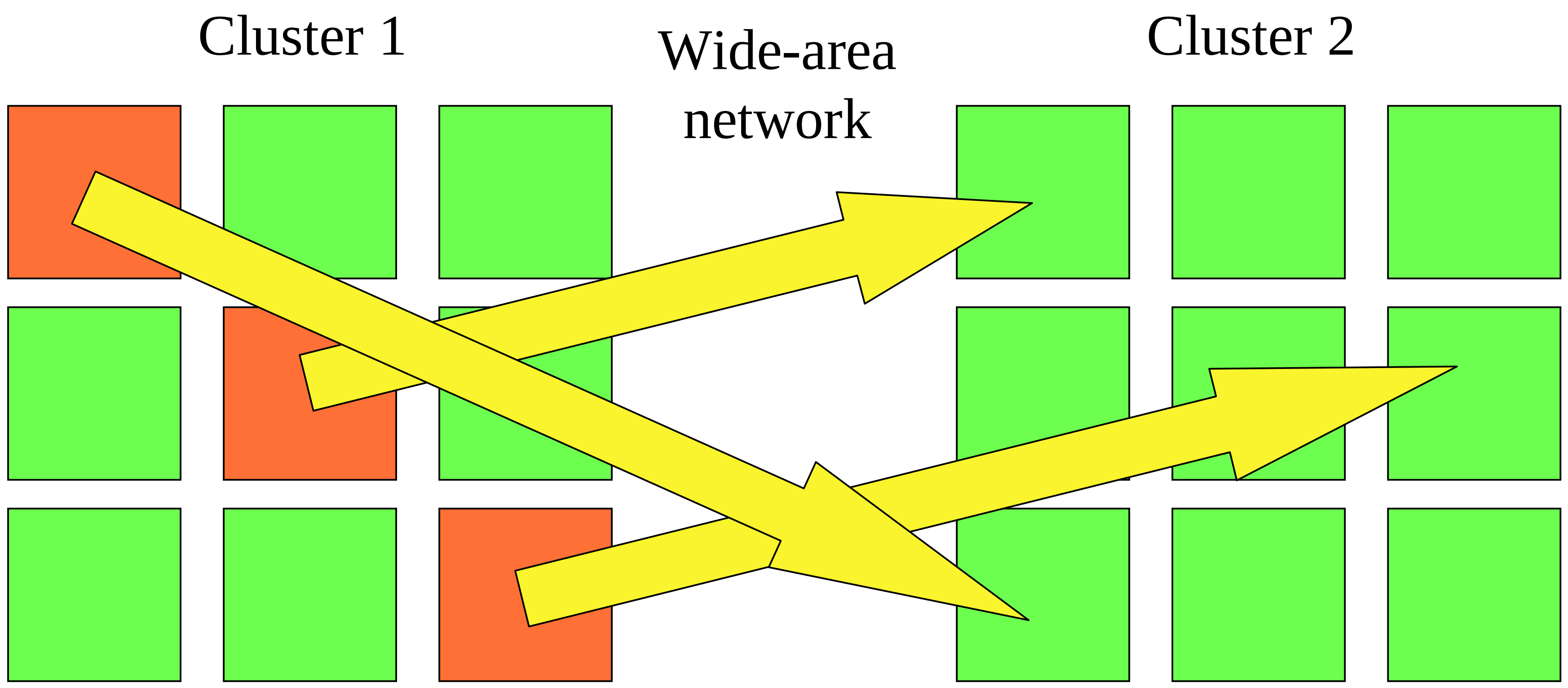
Cluster-aware Random Stealing (CRS)



Cluster-aware Random Stealing (CRS)

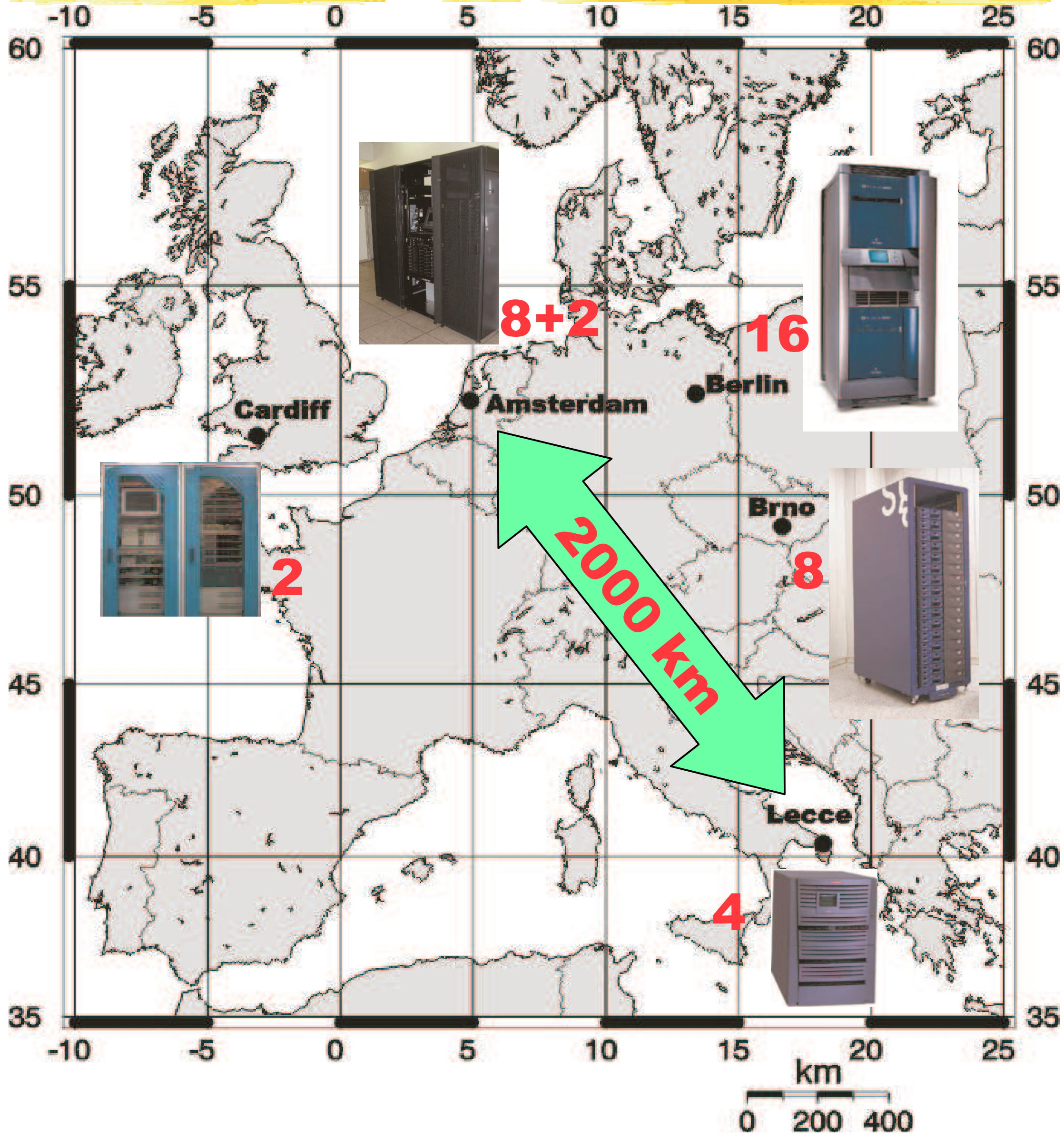


Cluster-aware Random Stealing (CRS)



Satin raytracer on a real grid

- GridLab testbed: 5 cities in Europe
- 40 machines
- Distance 2000 km / 1250 miles
- Factor of 10 difference in CPU speeds
- Latencies: 0.2 - 210 ms daytime, 0.2 - 66 ms night
- Bandwidths: 9 KB/s – 11 MB/s
- Three orders of magnitude difference in communication speeds



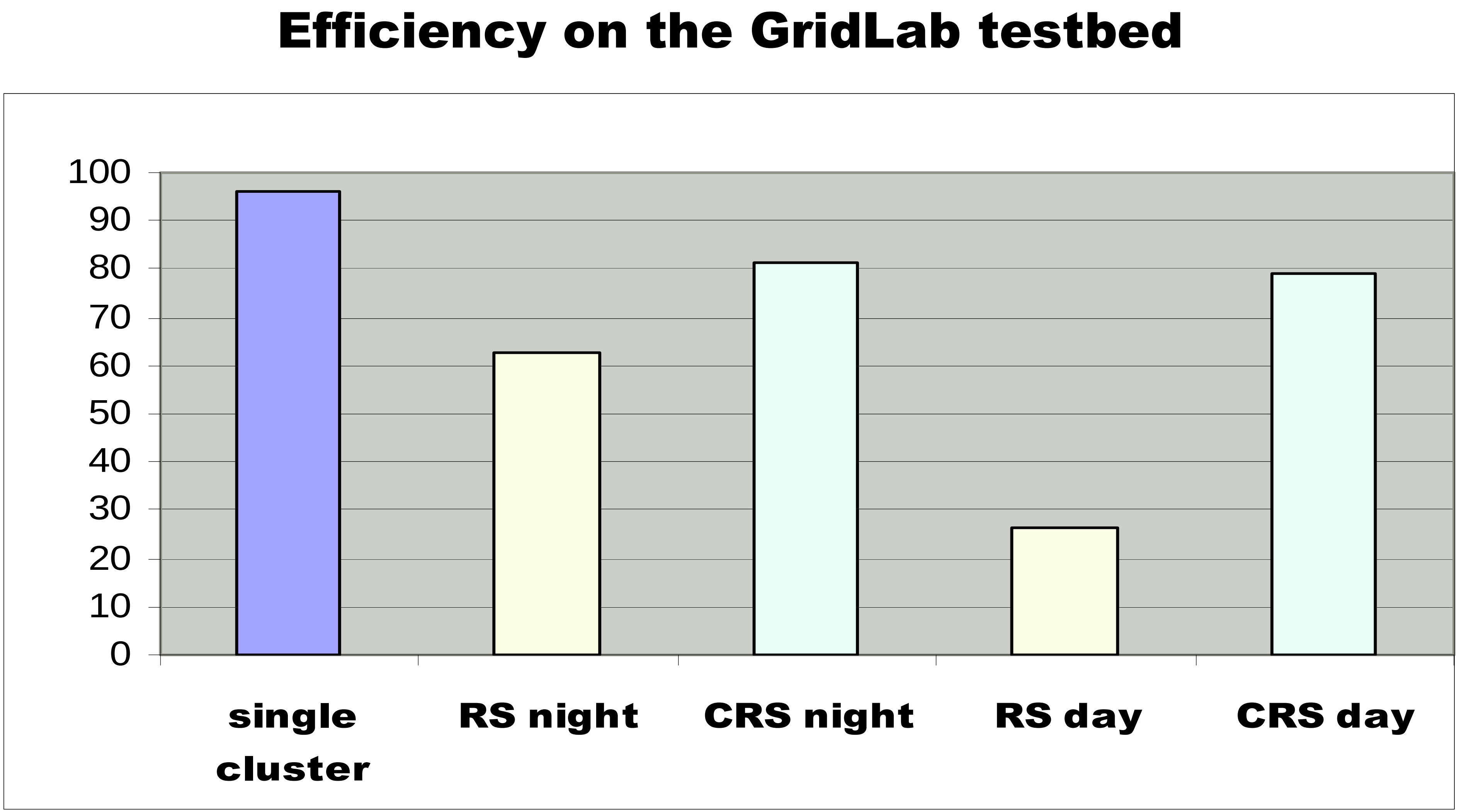
Configuration

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

Performance measurements

- Problem: how to define efficiency on a grid?
- Our approach:
  - Benchmark each CPU with Raytracer on small input
  - Normalize CPU speeds (relative to 1 GHz P3 node)
- Our case:
  - 40 CPUs, equivalent to 24.7 1 GHz P3 nodes
- Define:
  - T_perfect = sequential time on 1 GHz P3 node / 24.7
  - efficiency = T_perfect / actual runtime
  - Also compare against single 25-node 1 GHz P3 cluster

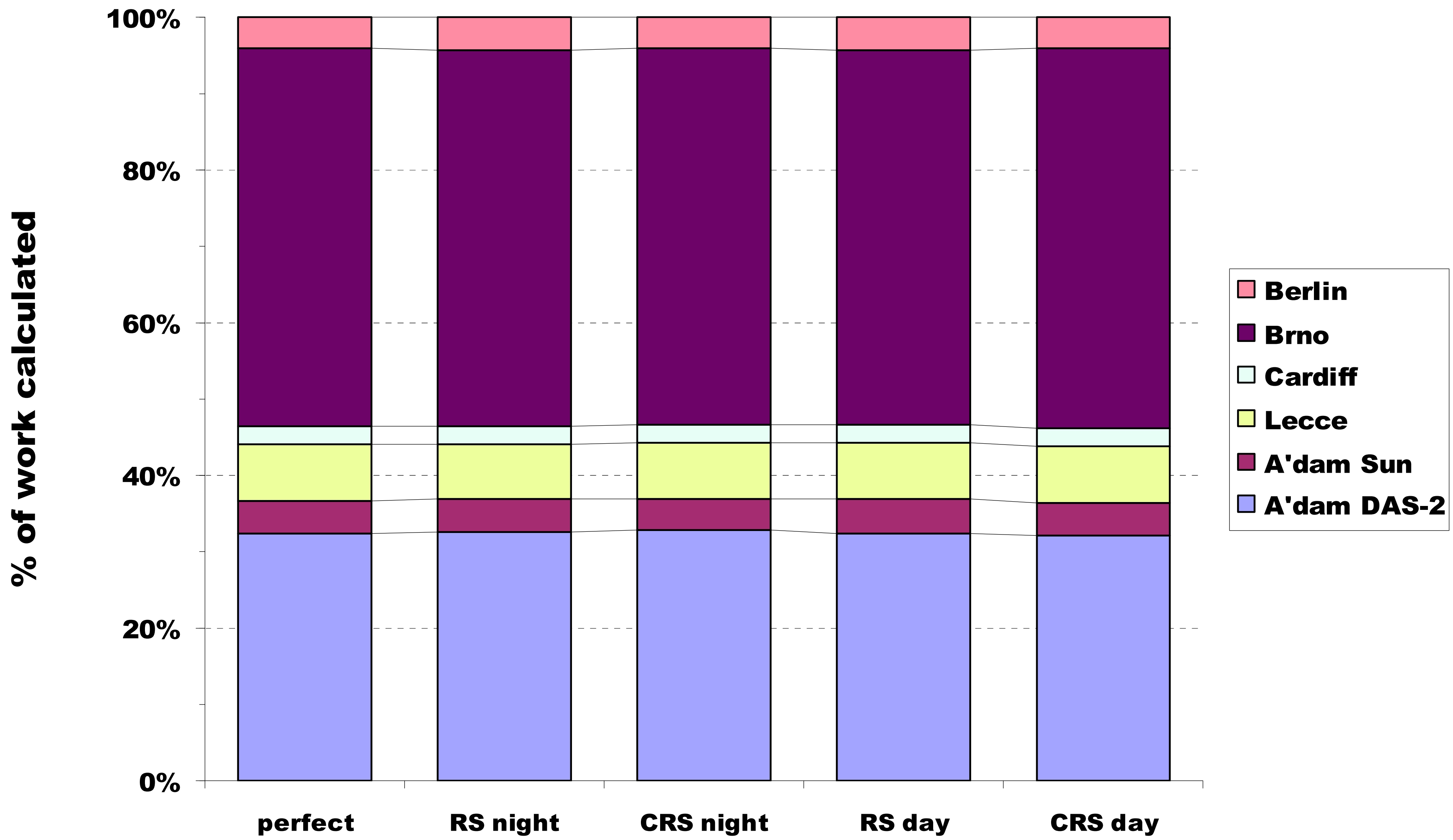
Performance on GridLab testbed



Communication statistics

	intra cluster		inter cluster	
	messages	Mbyte	messages	Mbyte
nighttime				
RS	3,218	41.8	11,473	137.3
CRS	1,353,295	131.7	12,153	86.0
daytime				
RS	56,686	18.9	149,634	154.1
CRS	2,148,348	130.7	10,115	82.1
single cluster	45,458	155.6	n.a.	n.a.

Work distribution



Experiences

- No support for co-allocation yet
  - done manually
- Firewall problems everywhere
  - Currently: use a range of site-specific open ports
  - Future: use multiplexing over ssh connections
- Java indeed runs everywhere

  modulo bugs in (old) JVMs
  - IBM 1.3 JDK: bug in versioning mechanism
  - Origin 1.3 JDK: bug in thread synchronization

Conclusion

- Ibis: a programming environment for grids
- RMI, group communication, divide&conquer
- Portable
  - Using Java's "write once, run everywhere" property
- Efficient
  - Reasonably efficient "run everywhere" solution
  - Optimized solutions for special cases
- Satin: divide-and-conquer parallelism
- Experience on *real* European grid testbed
- Portable and efficient with special grid-aware load-balancing algorithm

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

Cluster-aware Random Stealing (CRS)

- Use Random Stealing inside clusters

- When a node becomes idle, it will:
  - Send an *asynchronous* wide-area steal request
  - Do random steals locally, execute stolen jobs
  - Make sure only 1 wide-area steal attempt is in progress at a time (simulate synchronous RS)

- Prefetching adapts
  - More idle nodes → more prefetching

SOR: Java/Ibis versus C/MPI