Satin

Rob van Nieuwpoort
*rob@**cs.vu.nl***

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

ibis

vrije Universiteit

Optimizing for the grid

- Grids usually are hierarchical
  - Collections of clusters, supercomputers
  - Fast local links, slow wide-area links
- Optimize algorithms to exploit hierarchy
  - Message combining + latency hiding on wide-area links
  - Collective operations for wide-area systems
  - Successful for many applications

Satin master-worker

- Master-worker parallelism
  - Divide work into parts
  - Spawn job for each part
  - Solve parts in parallel
  - Combine results

Satin divide-and-conquer

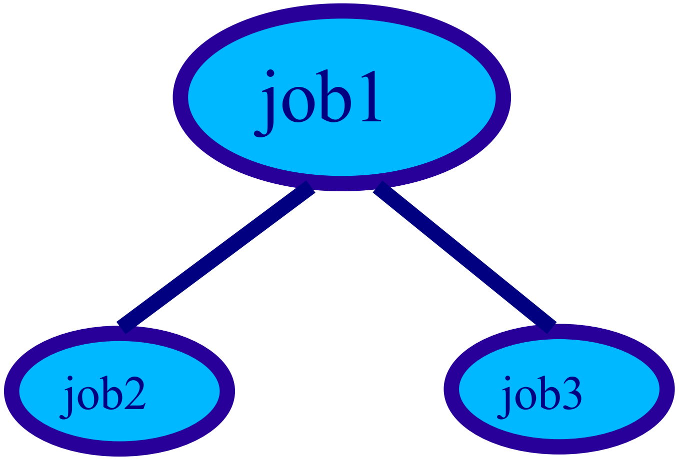
- Parallel Divide-and-conquer
  - Divide work into parts
  - Spawn job for each part
  - Solve parts in parallel
  - Combine results
  - But now recursively!
  - sub-problems split the work up further and spawn their own sub-jobs

Satin

- Divide-and-conquer

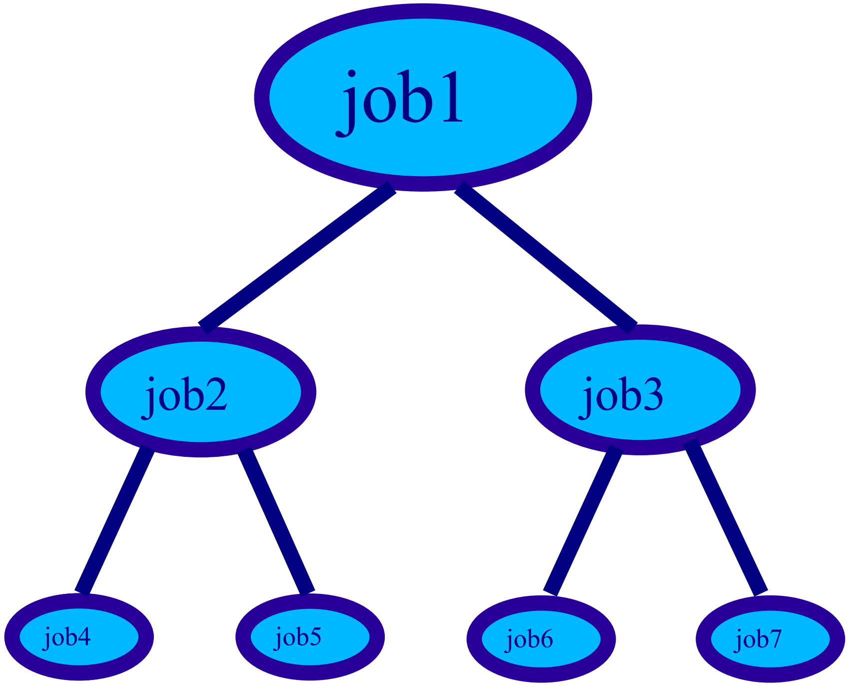
Satin

- Divide-and-conquer



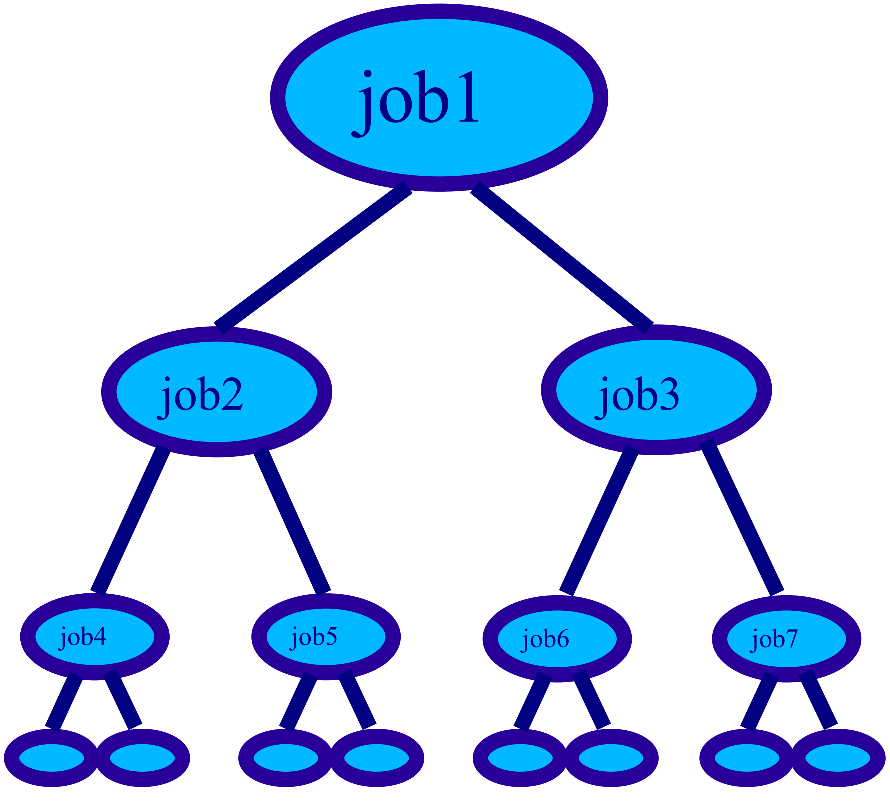
Satin

- Divide-and-conquer



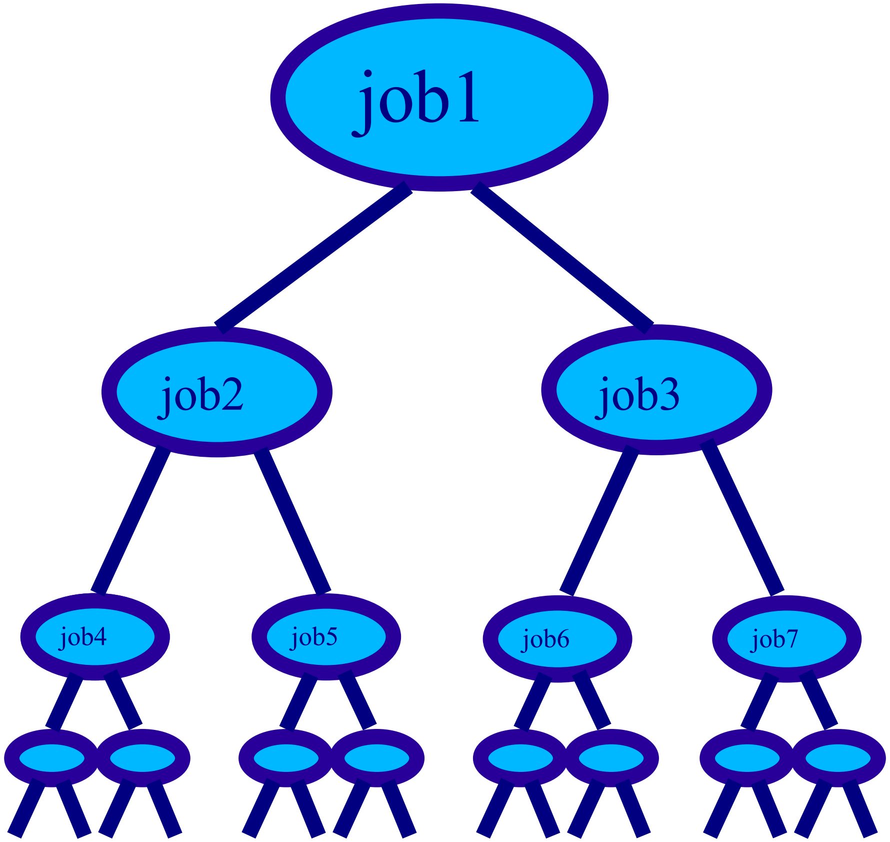
Satin

- Divide-and-conquer



Satin

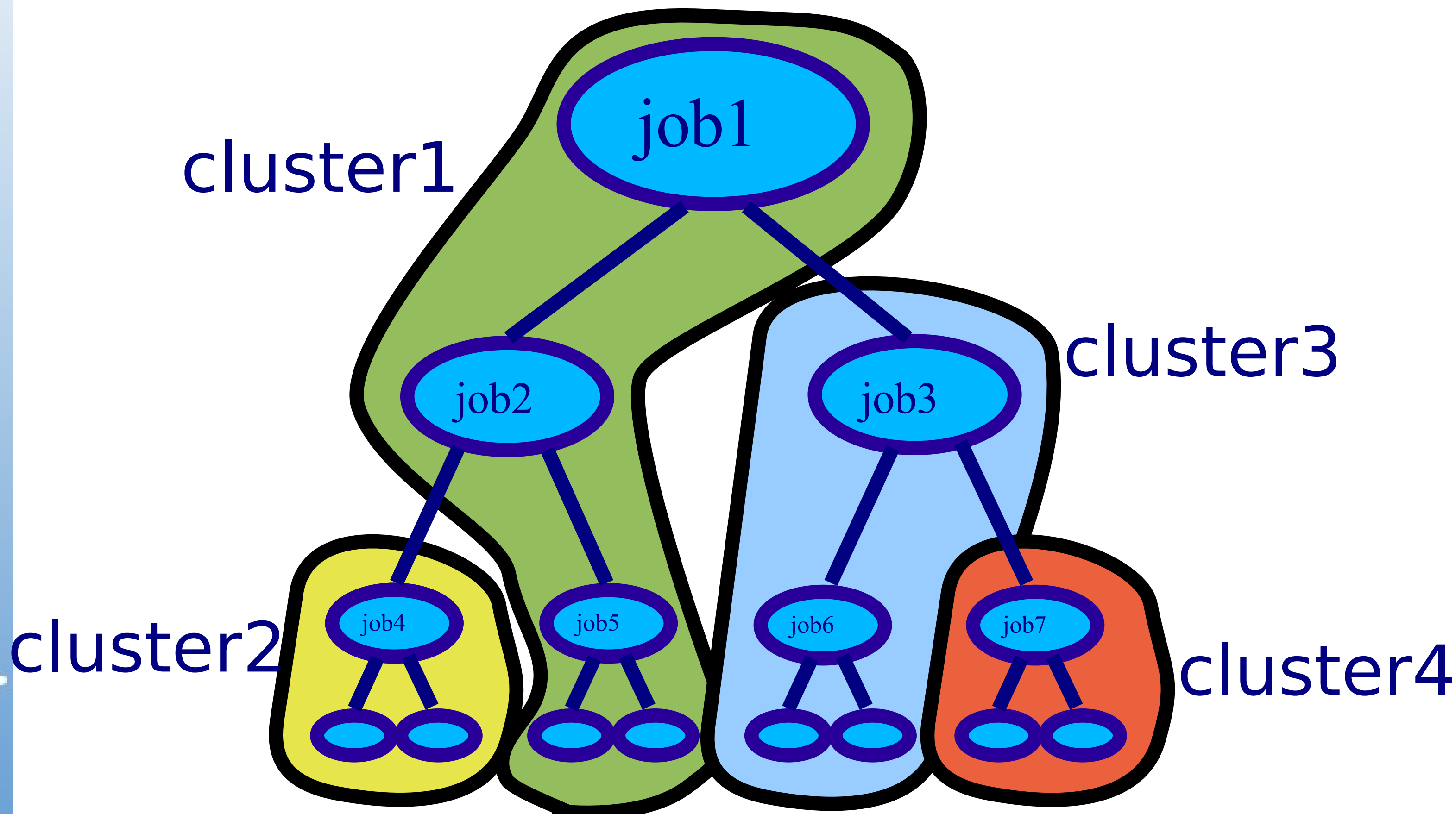
- Divide-and-conquer



... millions of jobs ...

Satin - Hierarchical

- Fits hierarchical structure of Grids



The Grid

- Satin explicitely targets the grid
- Different architectures --> Java
- Firewalls --> Ibis
- Slow networks (latency)
- Distributed memory
- Machines come and go
- Machines have different speeds
- Machines crash

Satin – Shared Objects

- Allow machines to share 'global data'
- Application controlled consistency (guard consistency)
- Allow different implementations
  - Special grid-aware multicast
  - Gossiping techniques
  - Point to point communication only

ibis

vrije Universiteit

Satin – Load Balancing

- Satin distributes jobs across machines
- Need load-balancing
  - Jobs can have different sizes
  - Machines have different speeds
- Load-balancing is done automatically

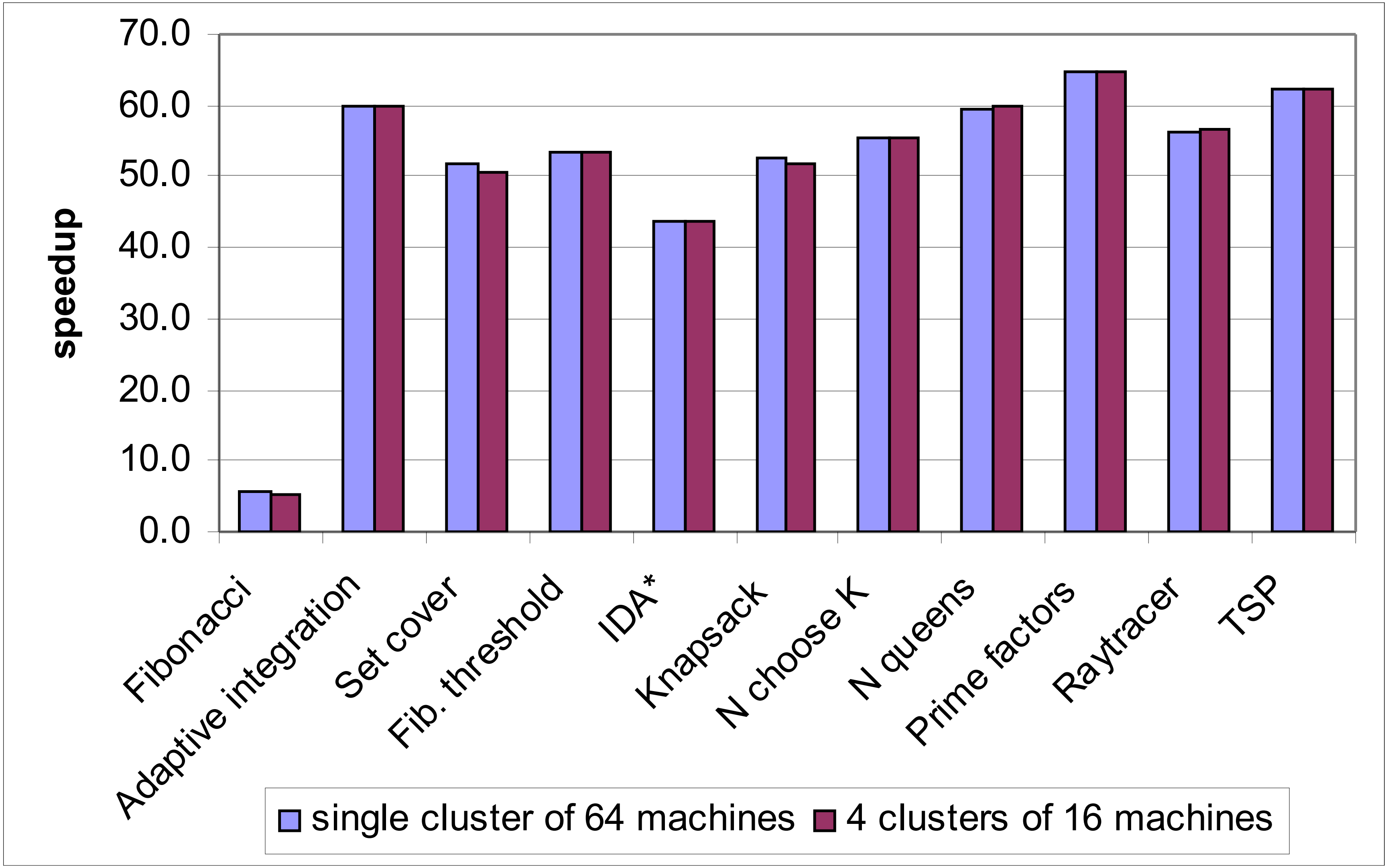
ibis

vrije Universiteit

Satin – Load Balancing

- Automatic load-balancing
  - On a cluster: Random Stealing
    - Randomly select other node, and 'steal' the largest pending job it has available.
    - Proven optimal in homogeneous systems.
  - multiple clusters / grid: Cluster-aware RS.
    - Randomly select node in other cluster and send asynchronous steal request
    - Do RS in local cluster while waiting for reply.
    - Not proven optimal, but works well

Satin – Load Balancing

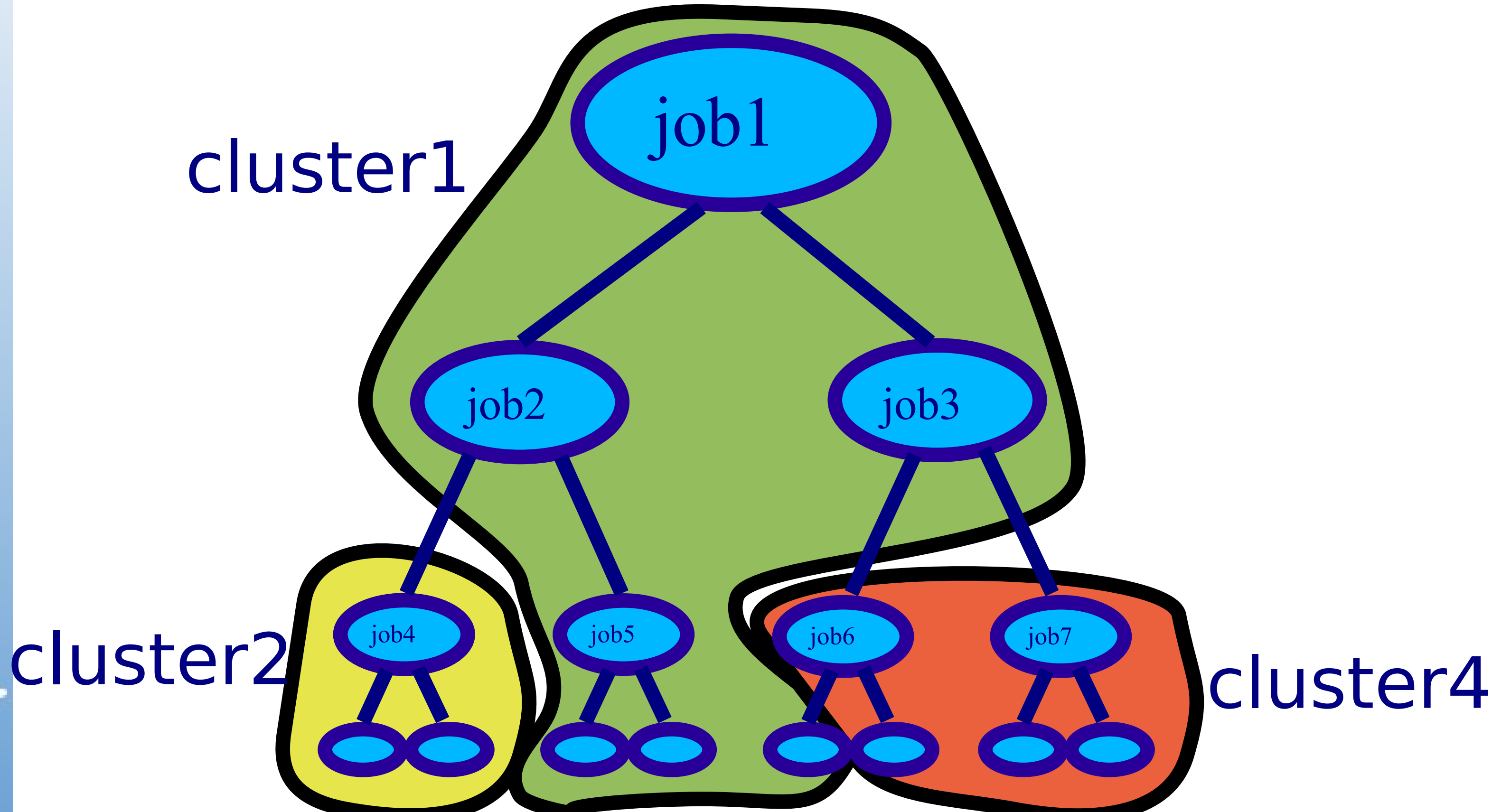


Satin – Malleability

- Add machines on the fly
  - User can add machines if compution is to slow or if more resource become available
- Remove machines on the fly
  - Machines can leave gracefully
  - Reservations can end
- Transparent for application

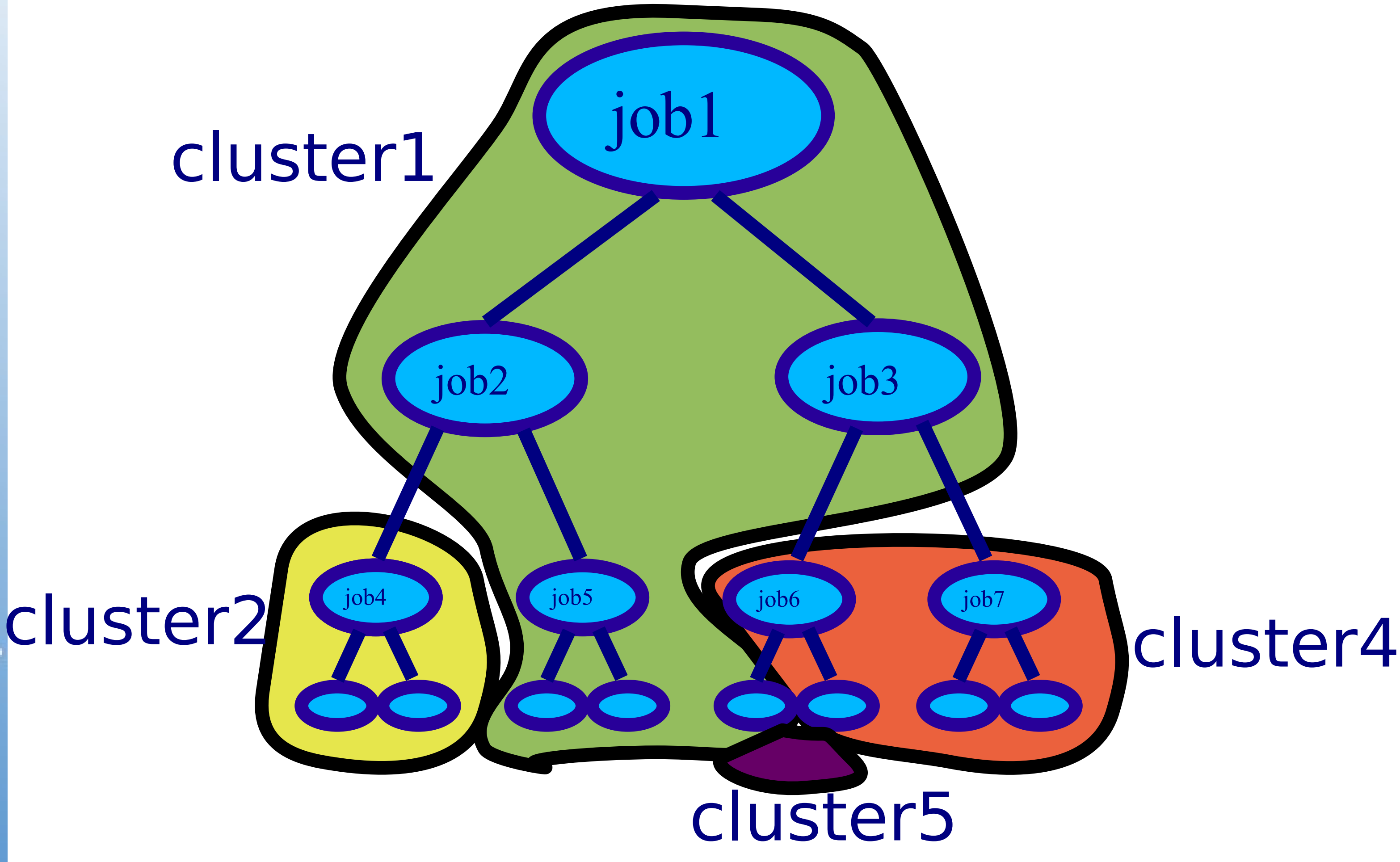
Satin – Malleability

- Transparently add machines



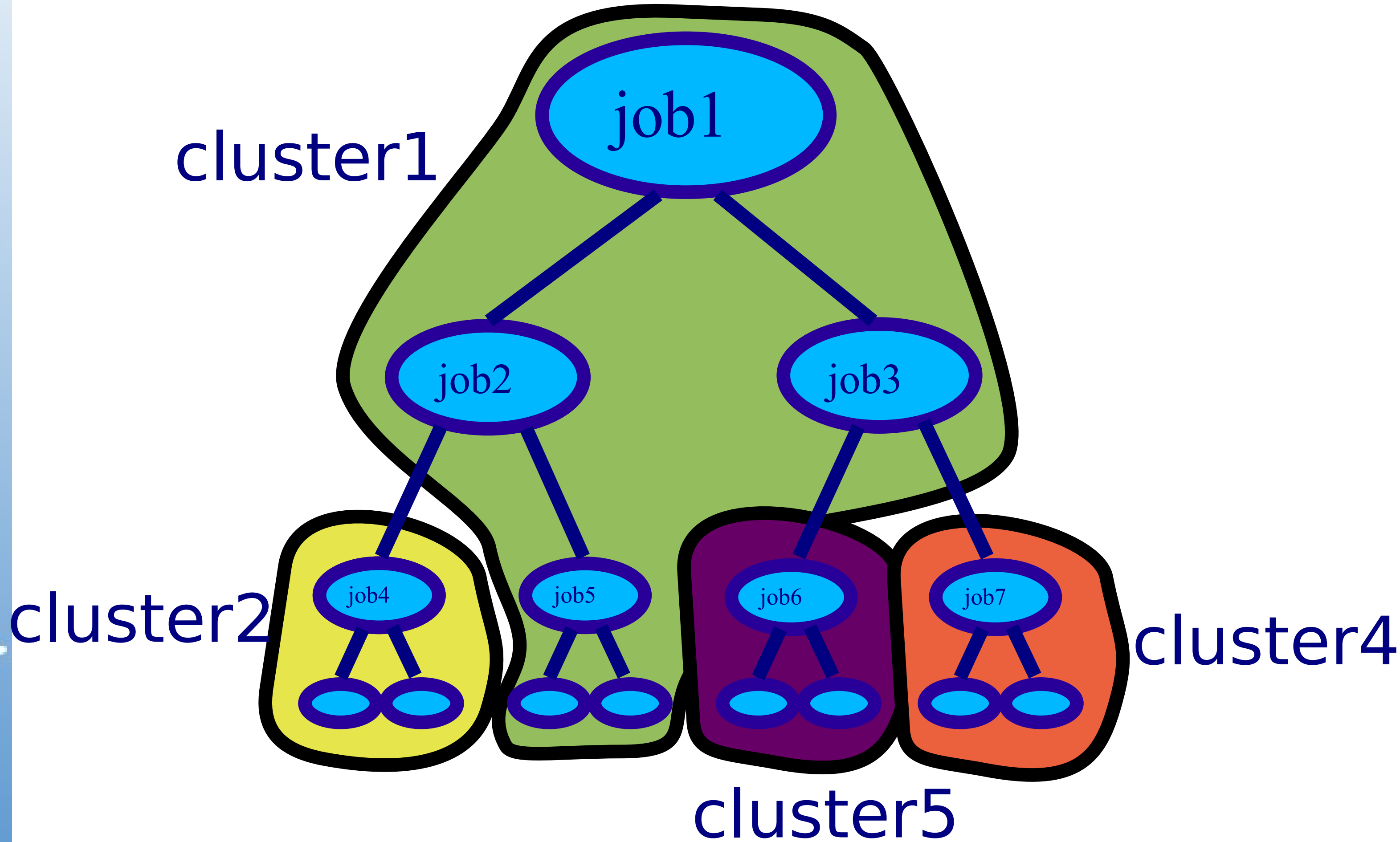
Satin – Malleability

- Transparently add machines



Satin – Malleability

- Transparently add machines

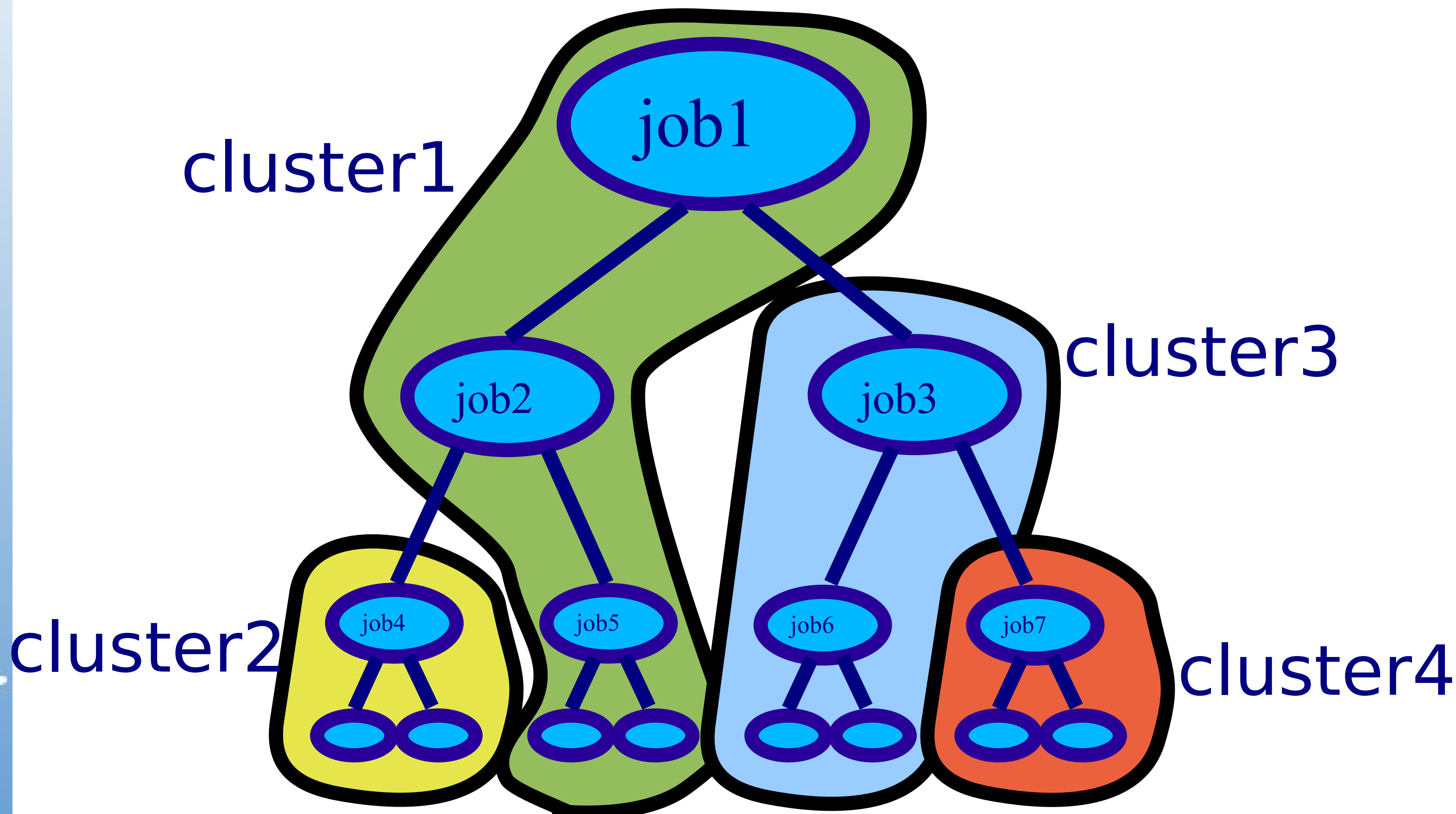


Satin – Fault Tolerance

- Machines can leave suddenly
  - Reservation can end without notification
  - Crashes
    - machines
    - network
    - software bugs
- Whole clusters can leave or crash
- The others continue the computation and automatically recompute lost work

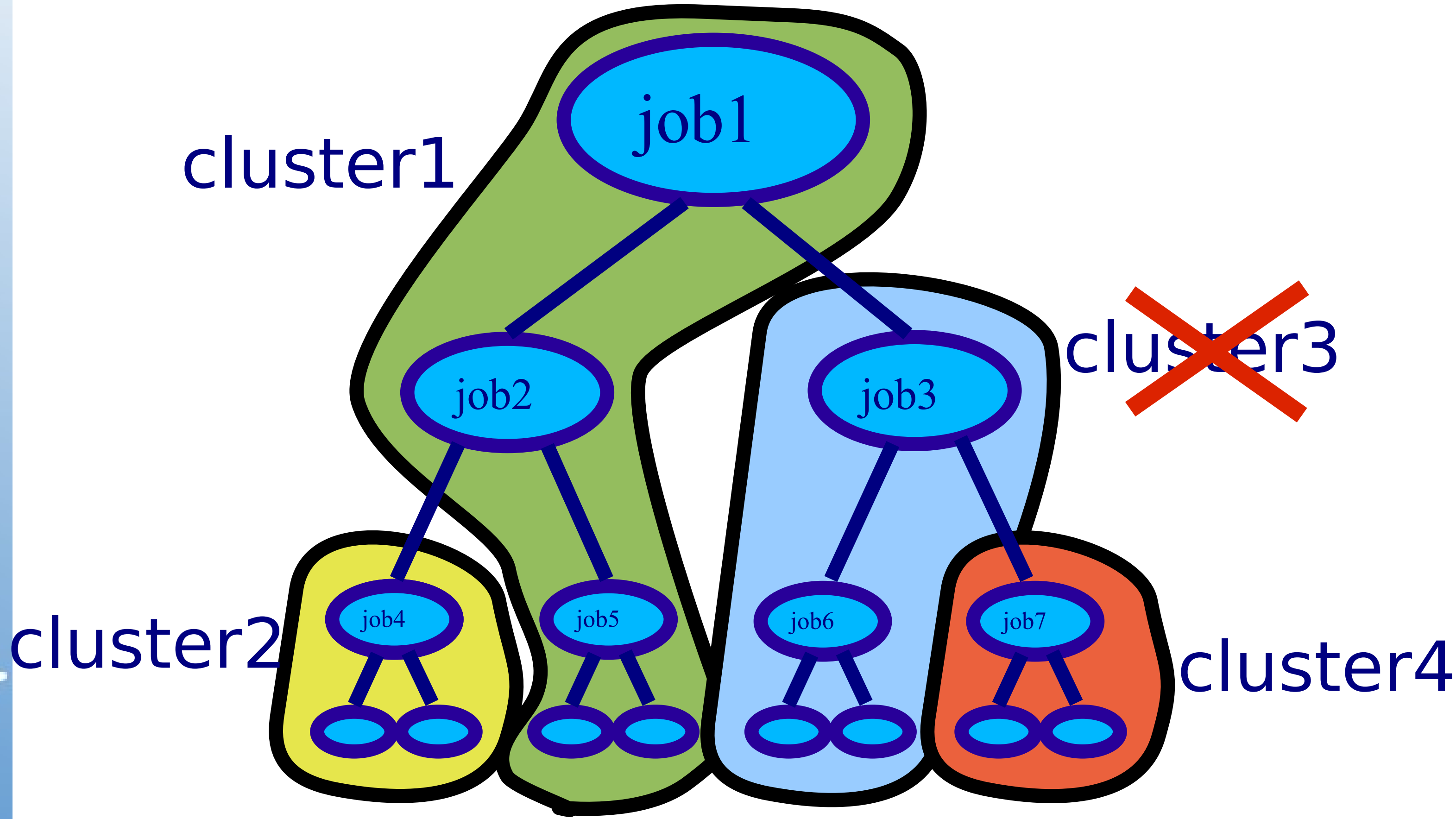
Satin – Fault Tolerance

- Transparent fault-tolerance



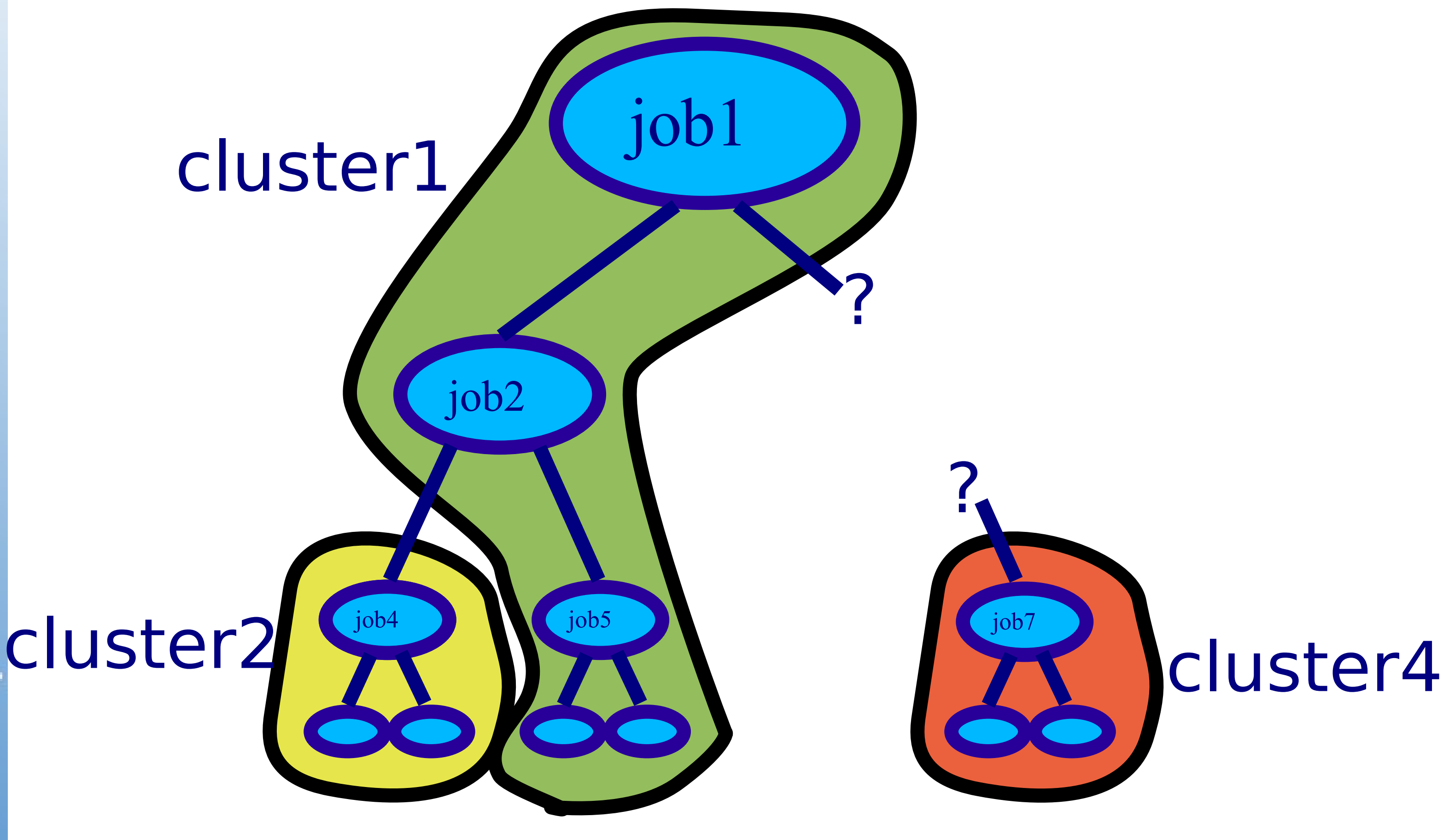
Satin – Fault Tolerance

- Transparent fault-tolerance



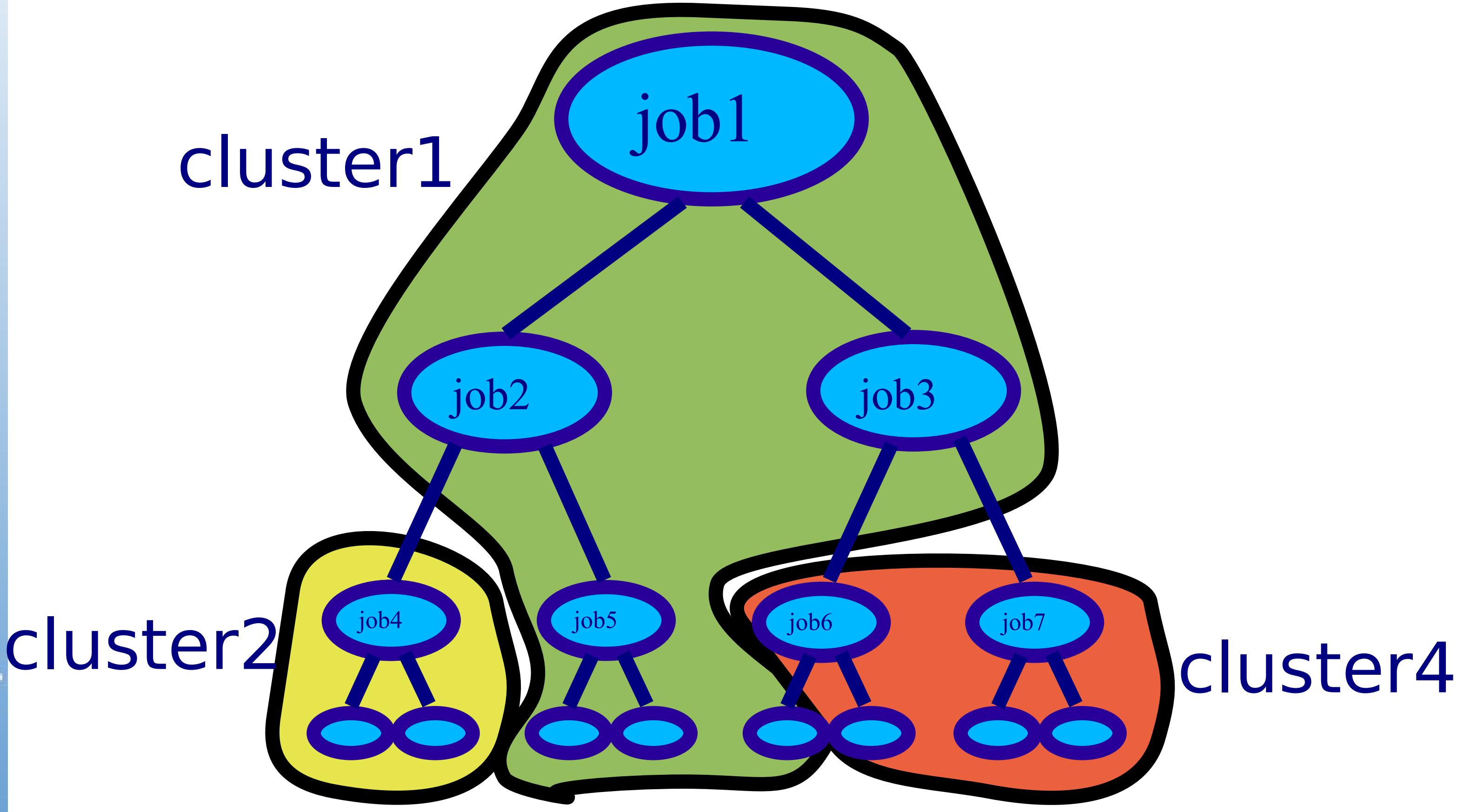
Satin – Fault Tolerance

- Transparent fault-tolerance

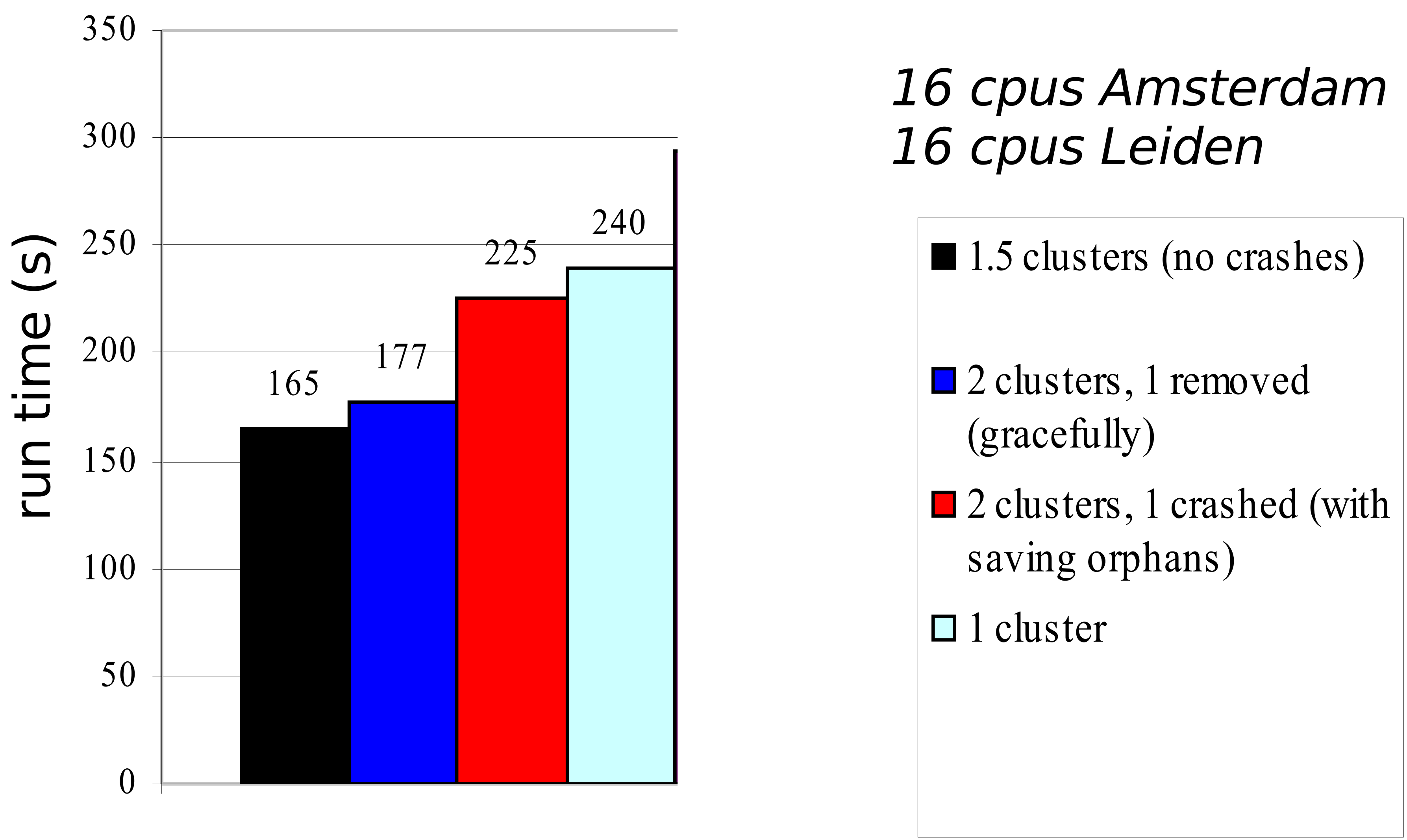


Satin – Fault Tolerance

- Transparent fault-tolerance



Satin – Fault Tolerance and Malleability Performance

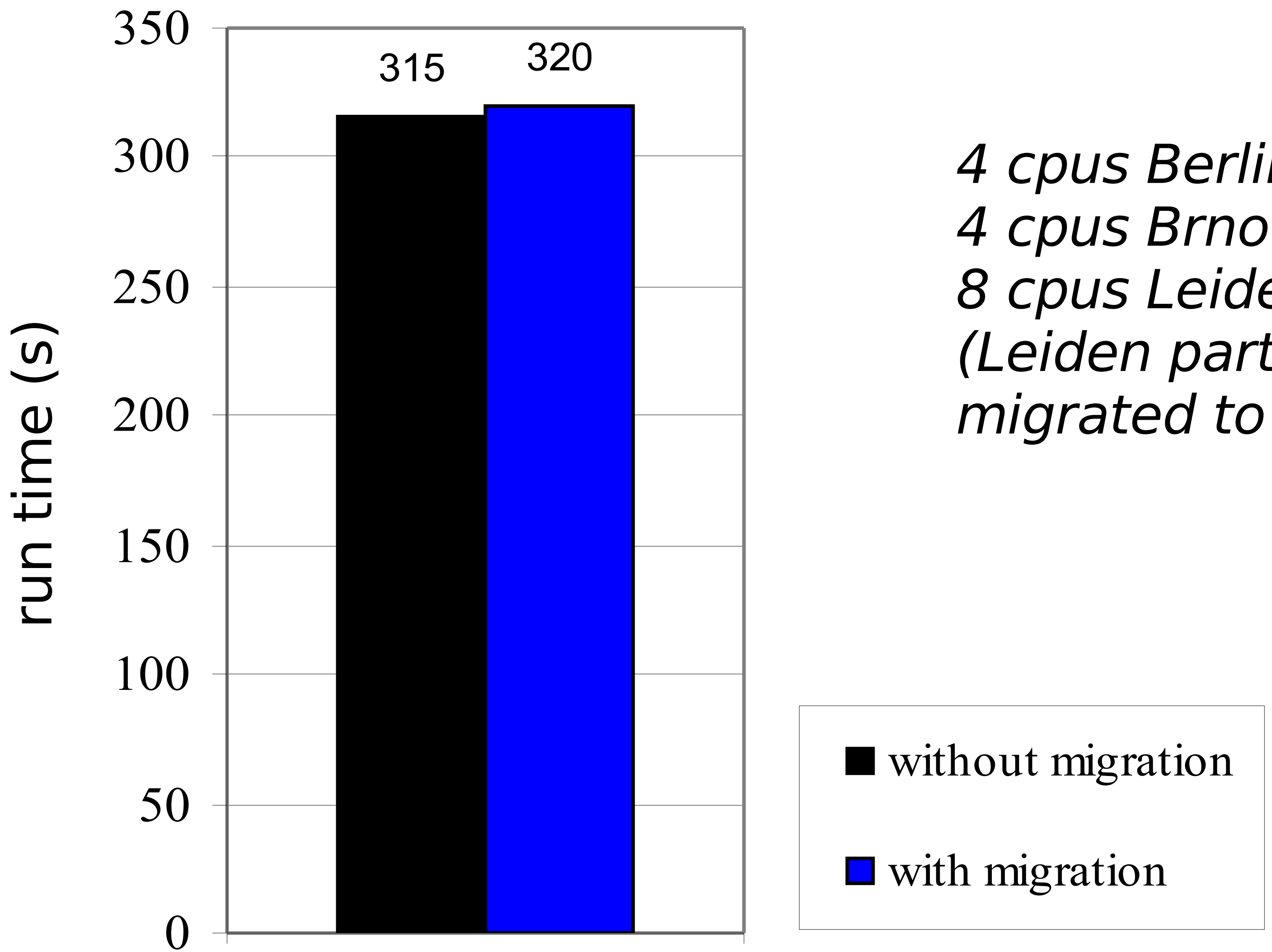


16 cpus Amsterdam
16 cpus Leiden

Satin – Migration

- Use malleability and fault tolerance
- Add new machines
- Remove old machines

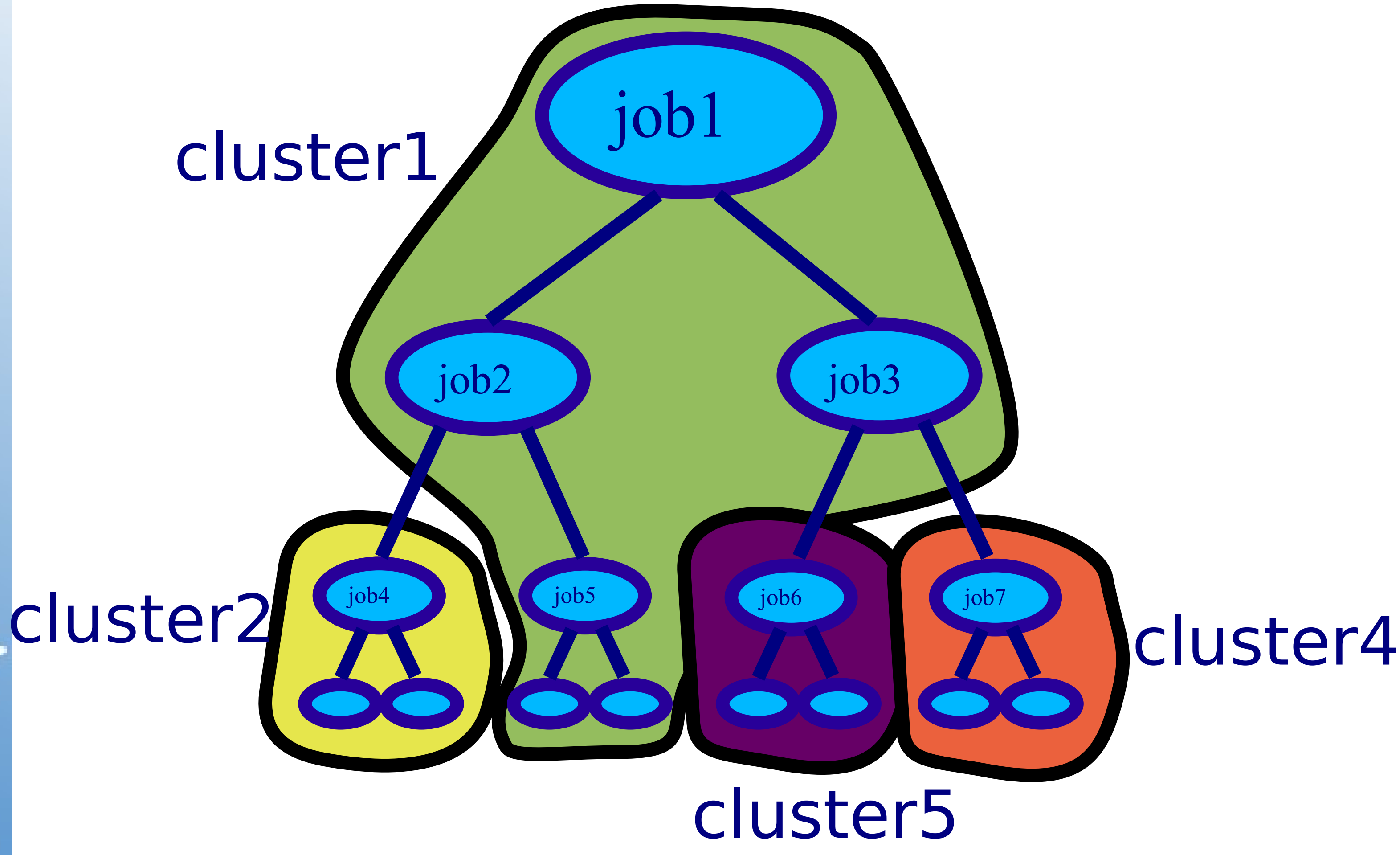
Satin - Migration



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

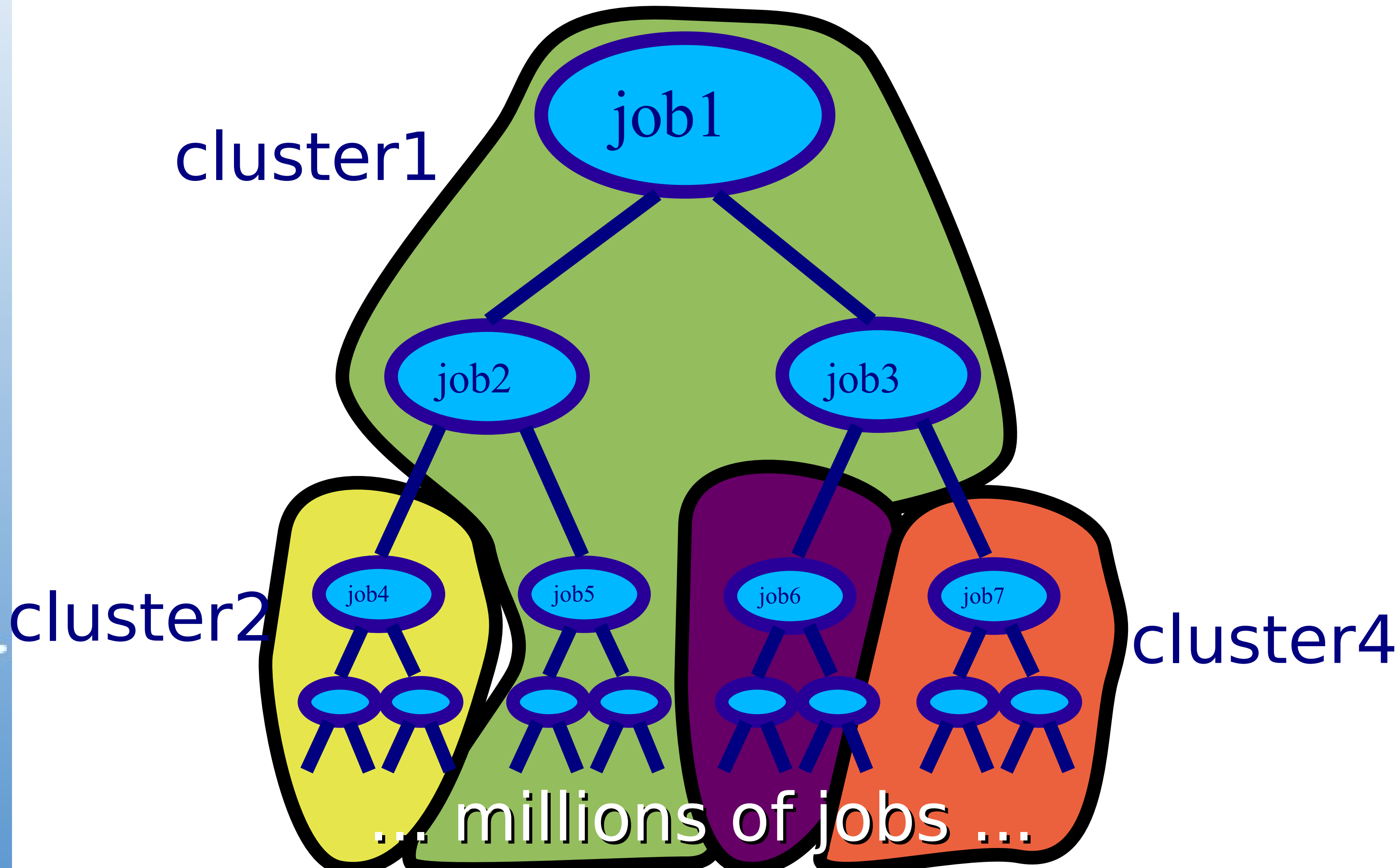
Satin – Adaptivity

- Automatically adapt number of machines



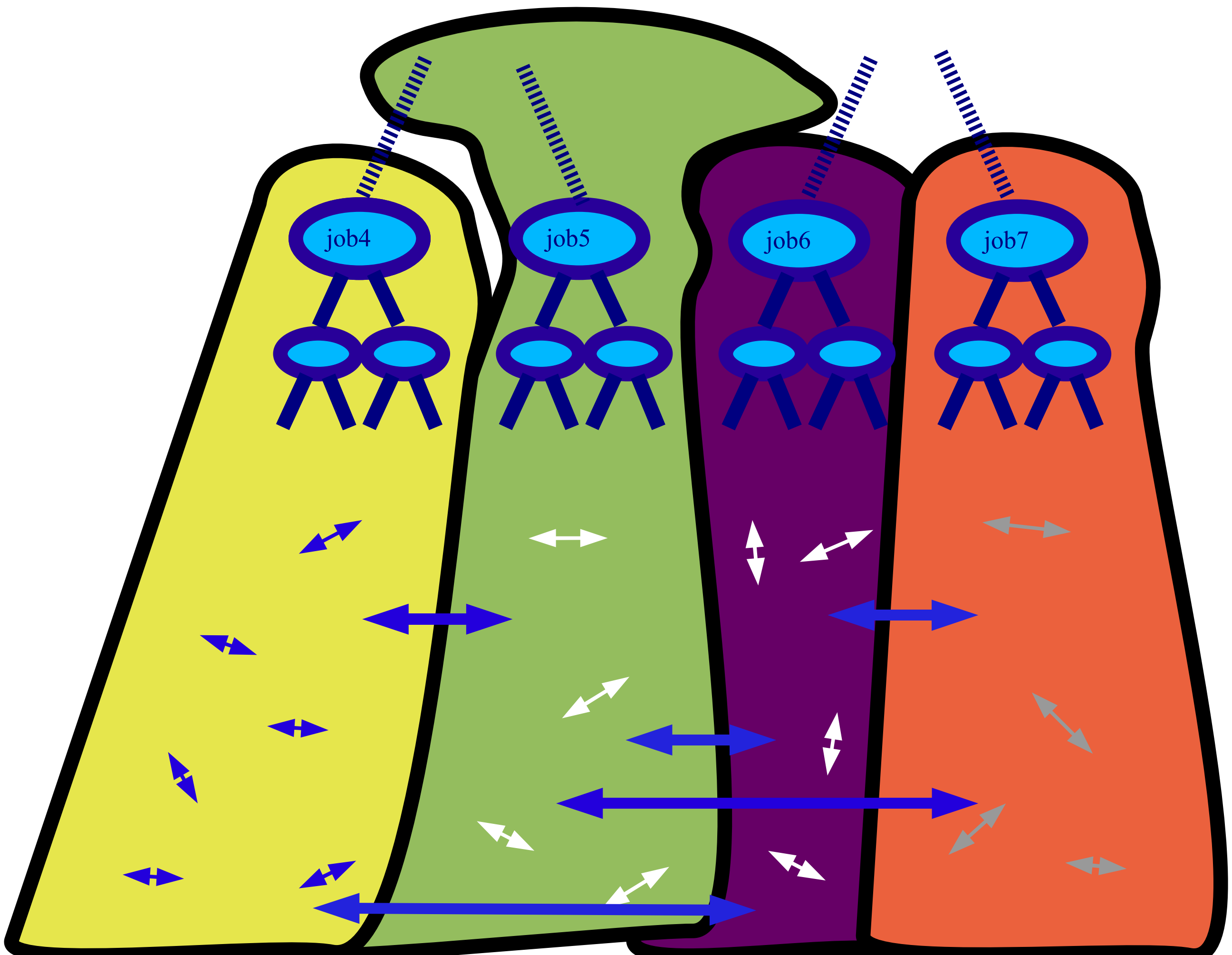
Satin – Adaptivity

- Automatically adapt number of machines



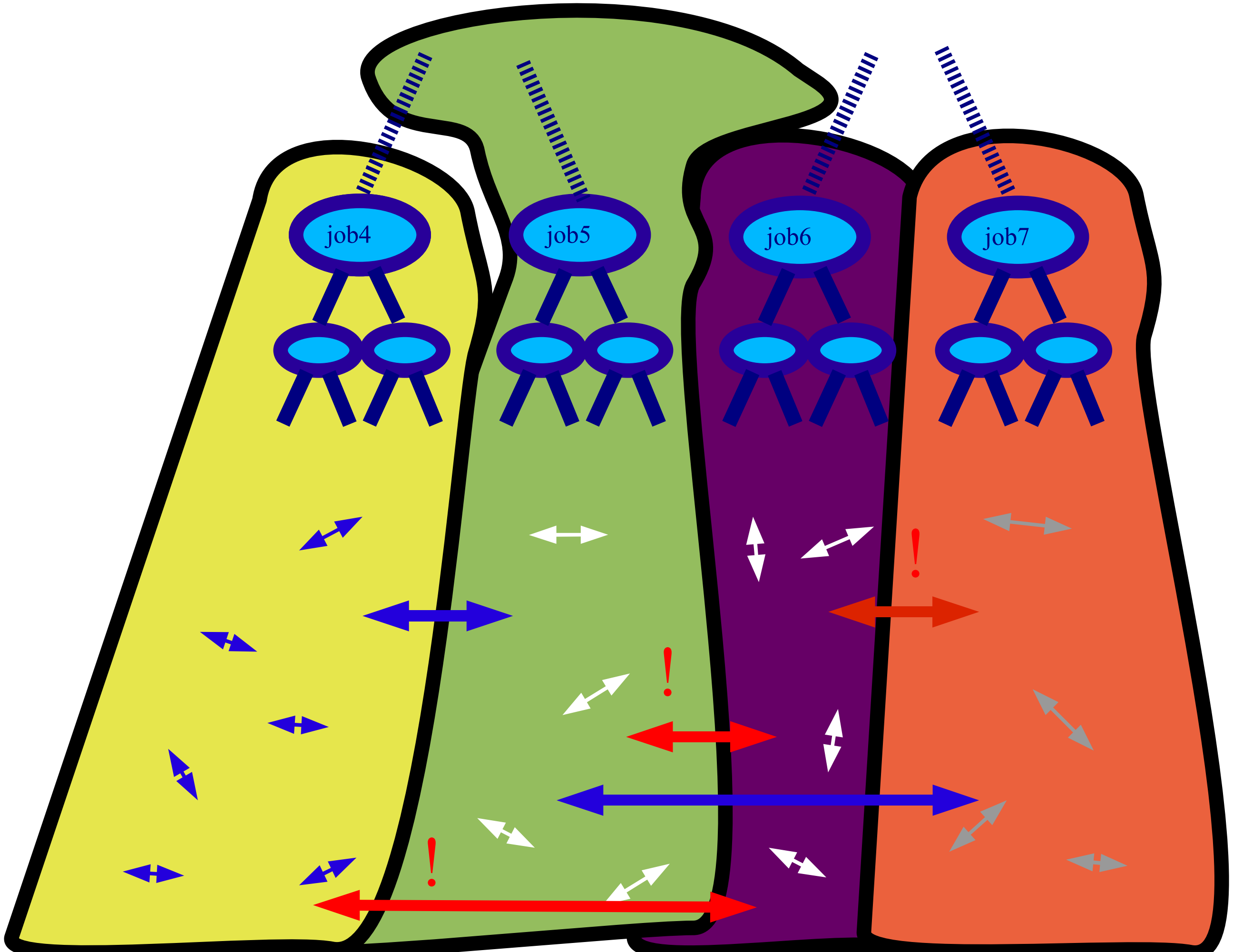
Satin – Adaptivity

- Measure overhead of each machine
  - Idle time + steal time



Satin – Adaptivity

- Find badly behaving machines/clusters

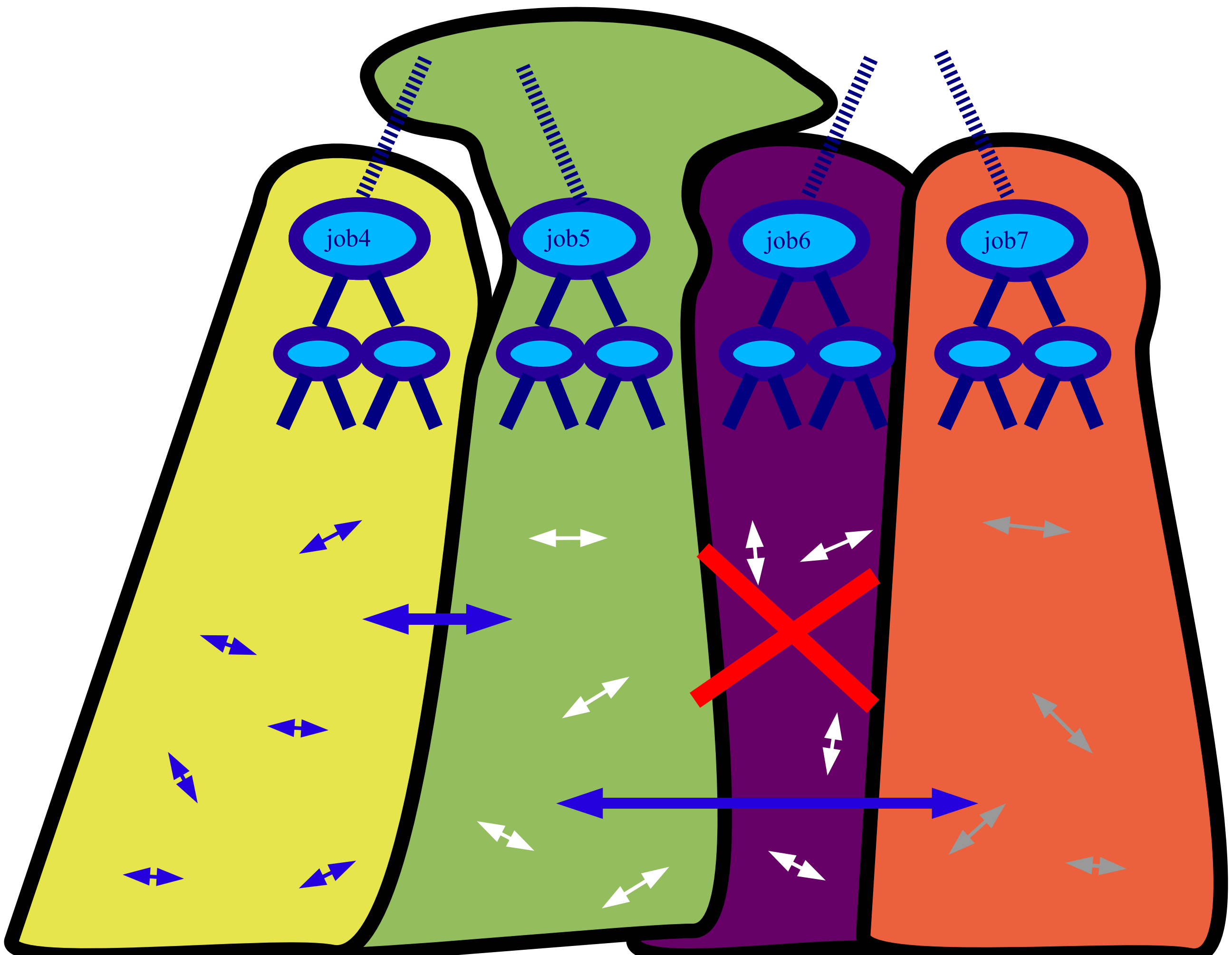


ibis

vrije Universiteit

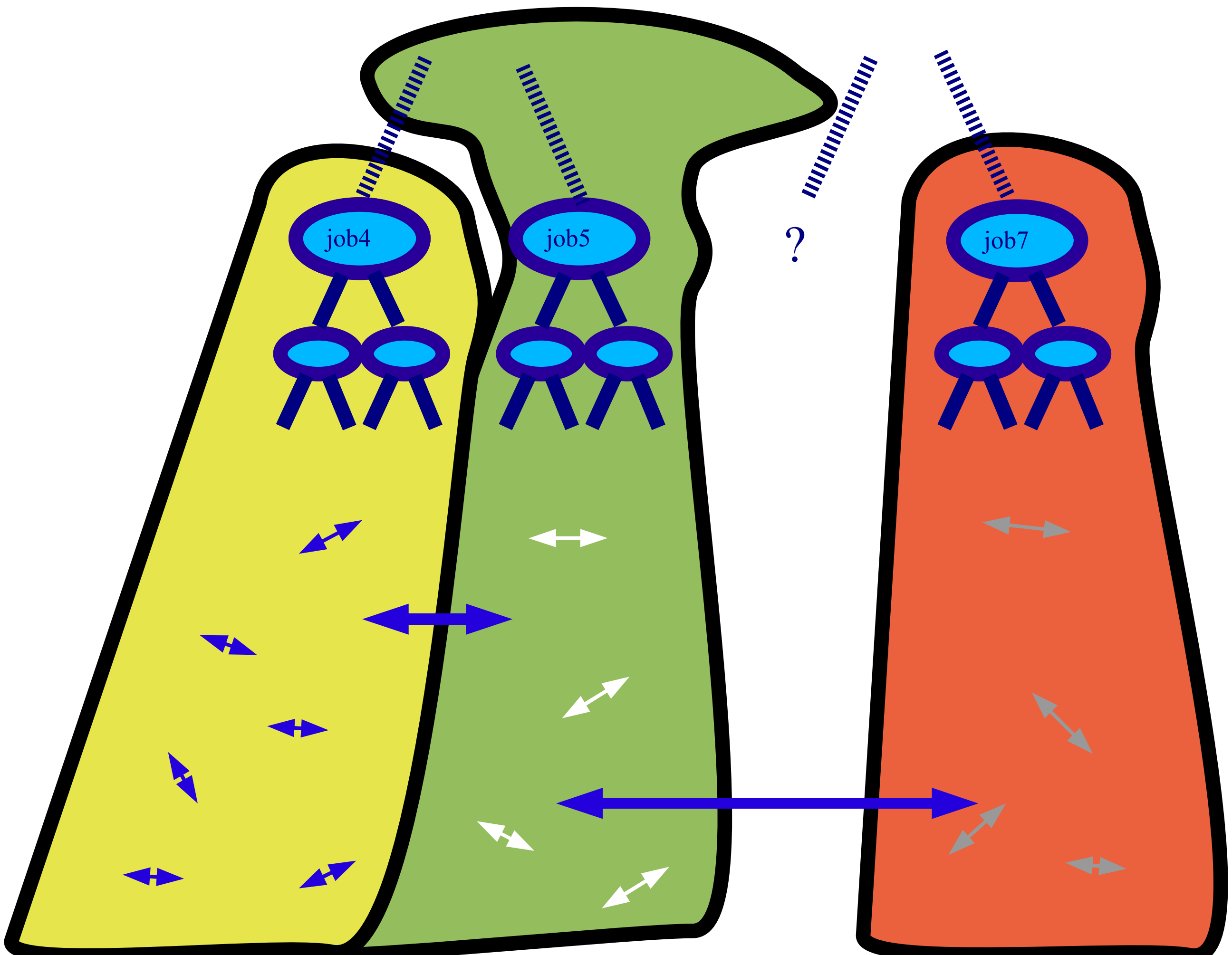
Satin – Adaptivity

- Remove them (or partly)



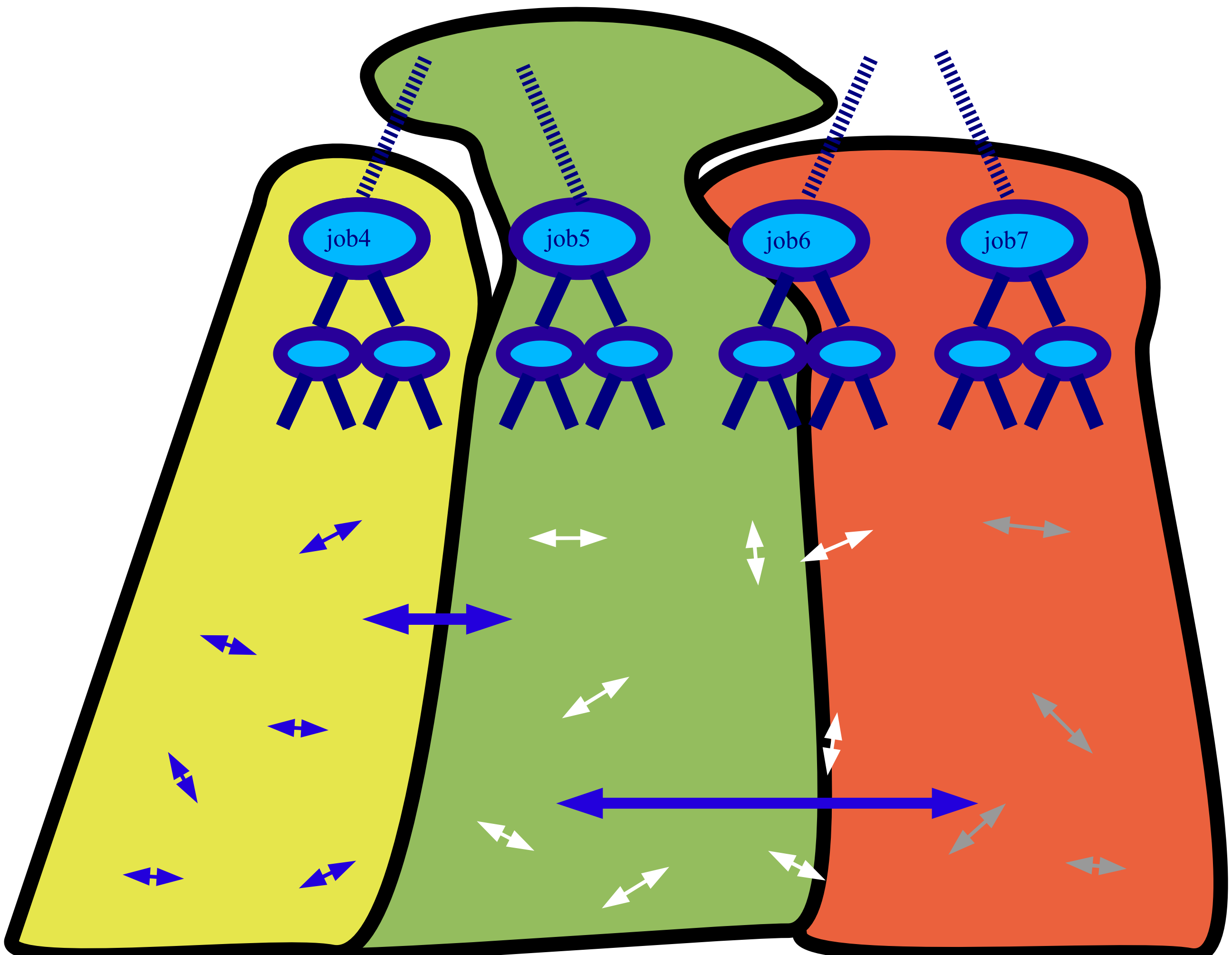
Satin – Adaptivity

- Remove them (or partly)



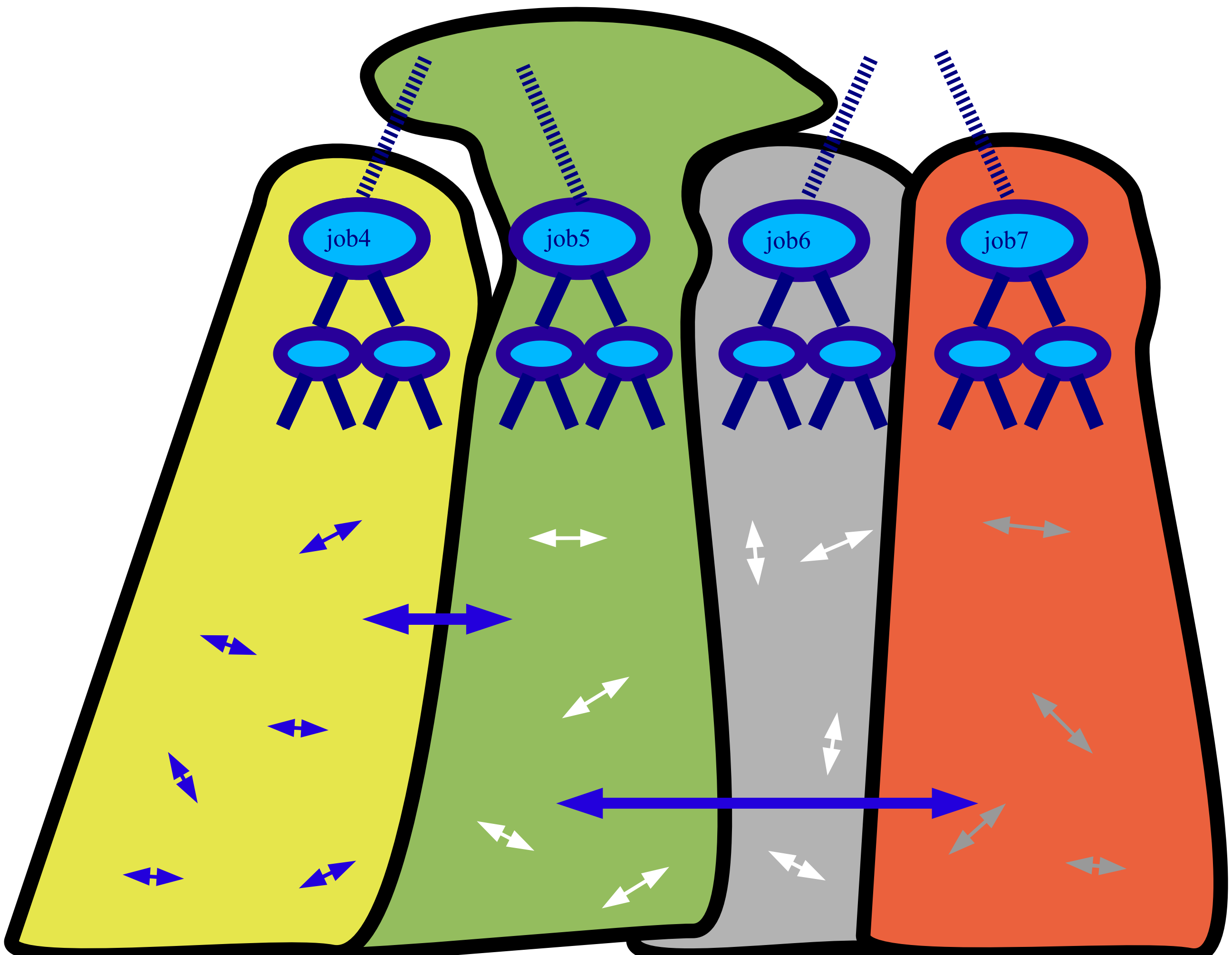
Satin – Adaptivity

- Satin will automatically recover



Satin – Adaptivity

- And may replace the machines/cluster



Summary: grid-enabling Satin

- The programming model itself
  - divide-and-conquer is inherently hierarchical: maps well on the grid
- Grid-aware load-balancing algorithm
  - Overlap wide-area communication with useful work
- Special consistency model for shared objects
  - Application defines consistency model
  - Allow different implementations
  - Special grid-aware multicast (if needed at all)
- Malleability, fault tolerance, migration, adaptivity
- Communicate through firewalls (thanks to Ibis)
- Portable (thanks to Java)

ibis

vrije Universiteit

Sequential Fibonacci

```
public long fib(int n) {
        if (n < 2) return n;

        long x = fib(n - 1);
        long y = fib(n – 2);

        return x + y;
}
```

Parallel Fibonacci

```
interface FibInterface extends ibis.satin.Spawnable {
    public long fib(int n);
}
public long fib(int n) {
        if (n < 2) return n;

        long x = fib(n - 1);
        long y = fib(n – 2);
        sync();
        return x + y;
}
```

Parallel Fibonacci

```
interface FibInterface extends ibis.satin.Spawnable {
    public long fib(int n);
}
public long fib(int n) {
        if (n < 2) return n;

        long x = fib(n - 1);
        long y = fib(n – 2);
        sync();
        return x + y;
}
```

Mark methods as

Spawnable.

They can run in parallel.

vrije Universiteit

Parallel Fibonacci

```
interface FibInterface extends ibis.satin.Spawnable {
    public long fib(int n);
}
public long fib(int n) {
        if (n < 2) return n;

        long x = fib(n - 1);
        long y = fib(n – 2);
        sync();
        return x + y;
}
```

Mark methods as Spawnable.

They can run in parallel.

Wait until spawned methods are done.

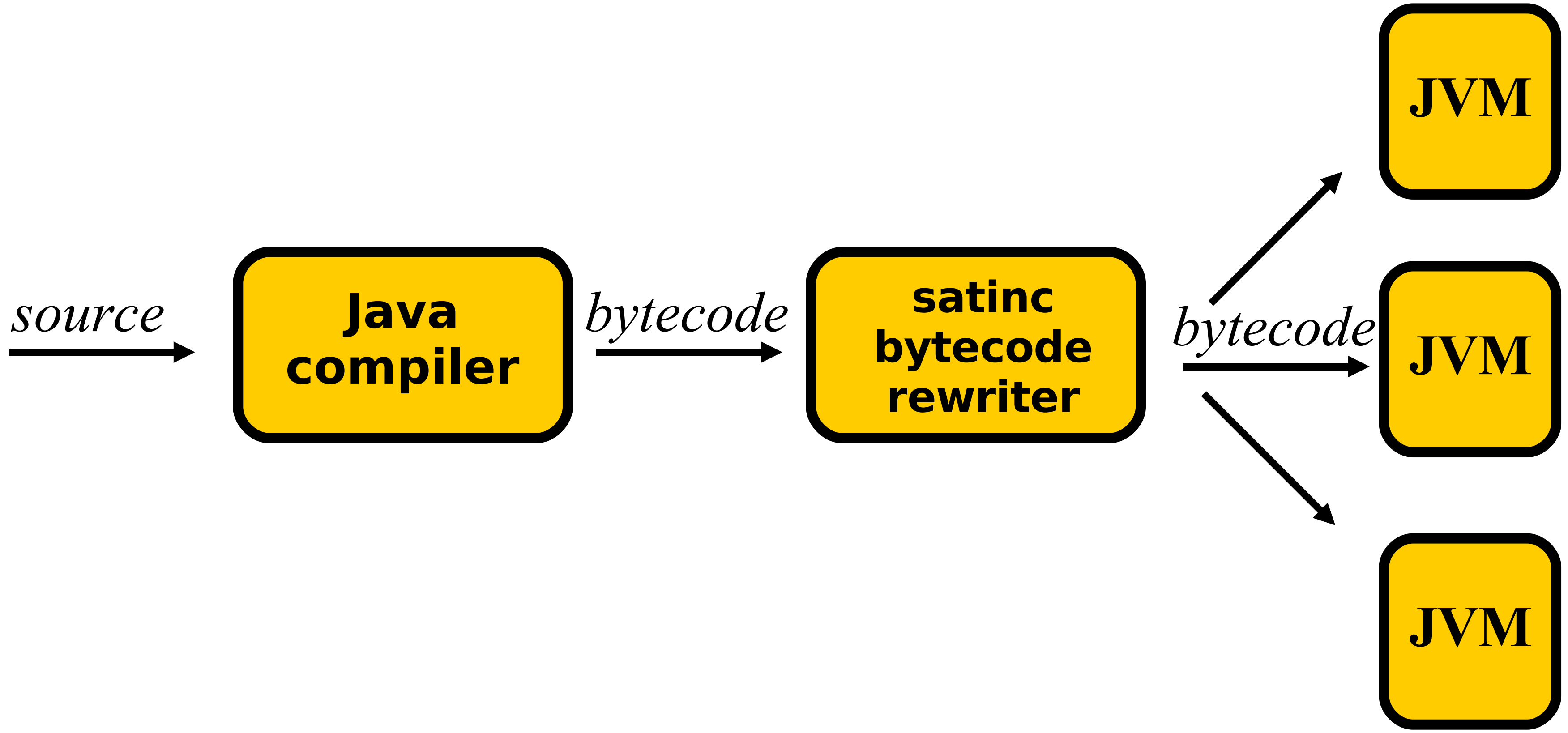
Spawning Jobs

- If a job is executed on a remote machine, the parameters are sent over the network (i.e. copied)
- If a job is executed locally, copying overhead should be avoided
  - 99% of all jobs run locally
- Parameter passing semantics
  - Cannot assume either call-by-value or call-by-reference
  - If a parameter is changed, copy it first

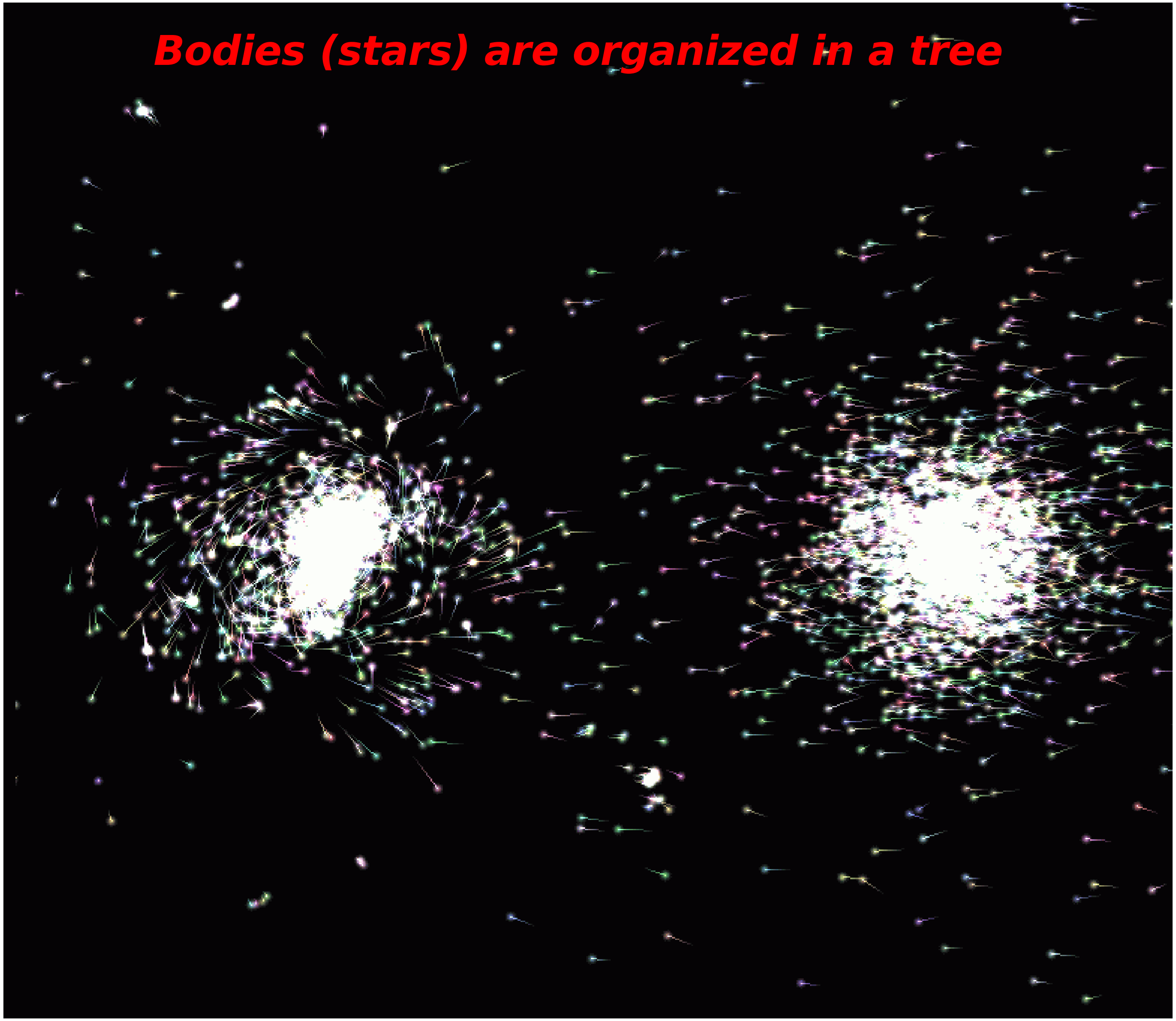
ibis

vrije Universiteit

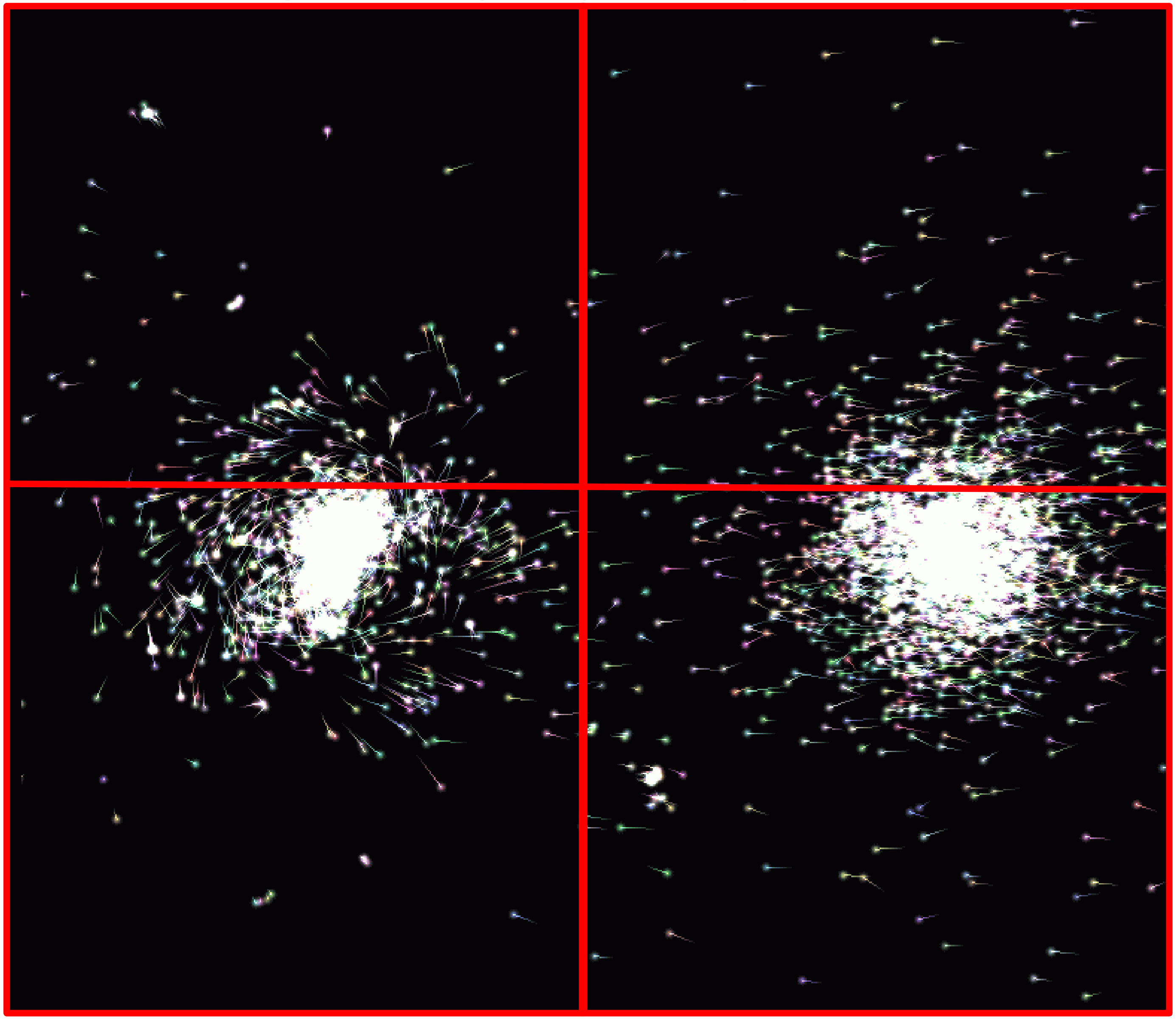
Compiling Satin Programs



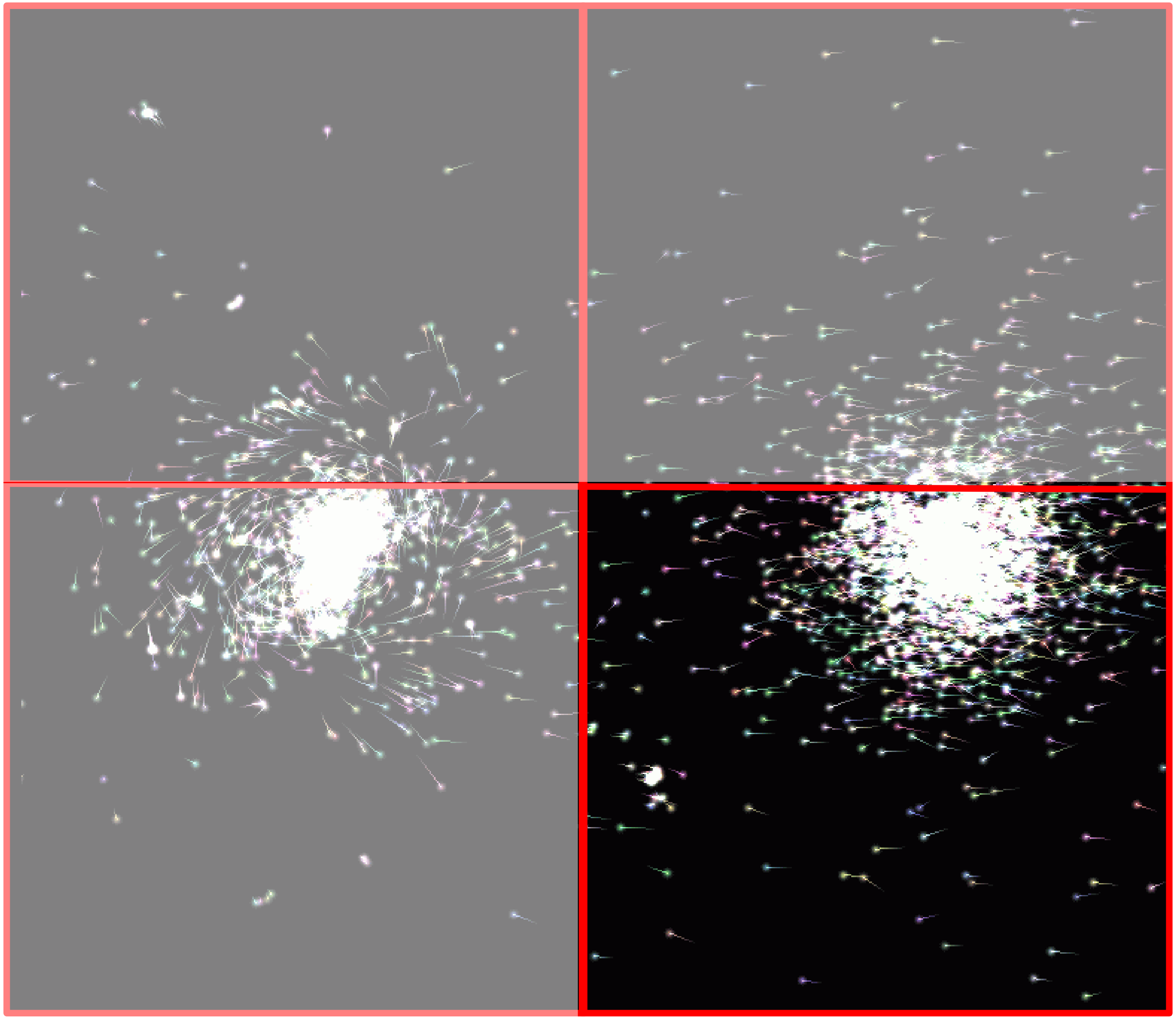
Barnes-Hut N-body simulation



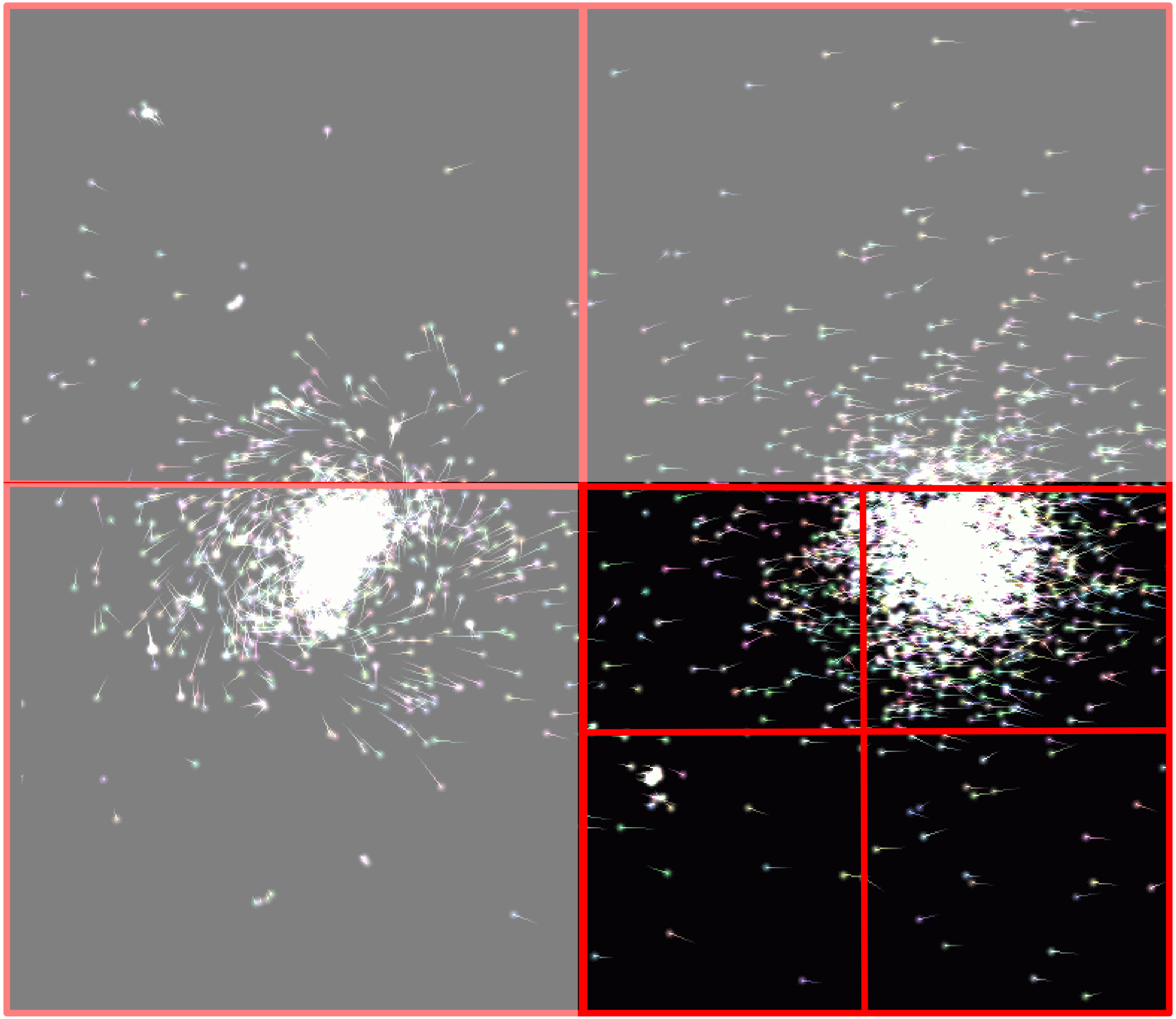
Barnes-Hut N-body simulation



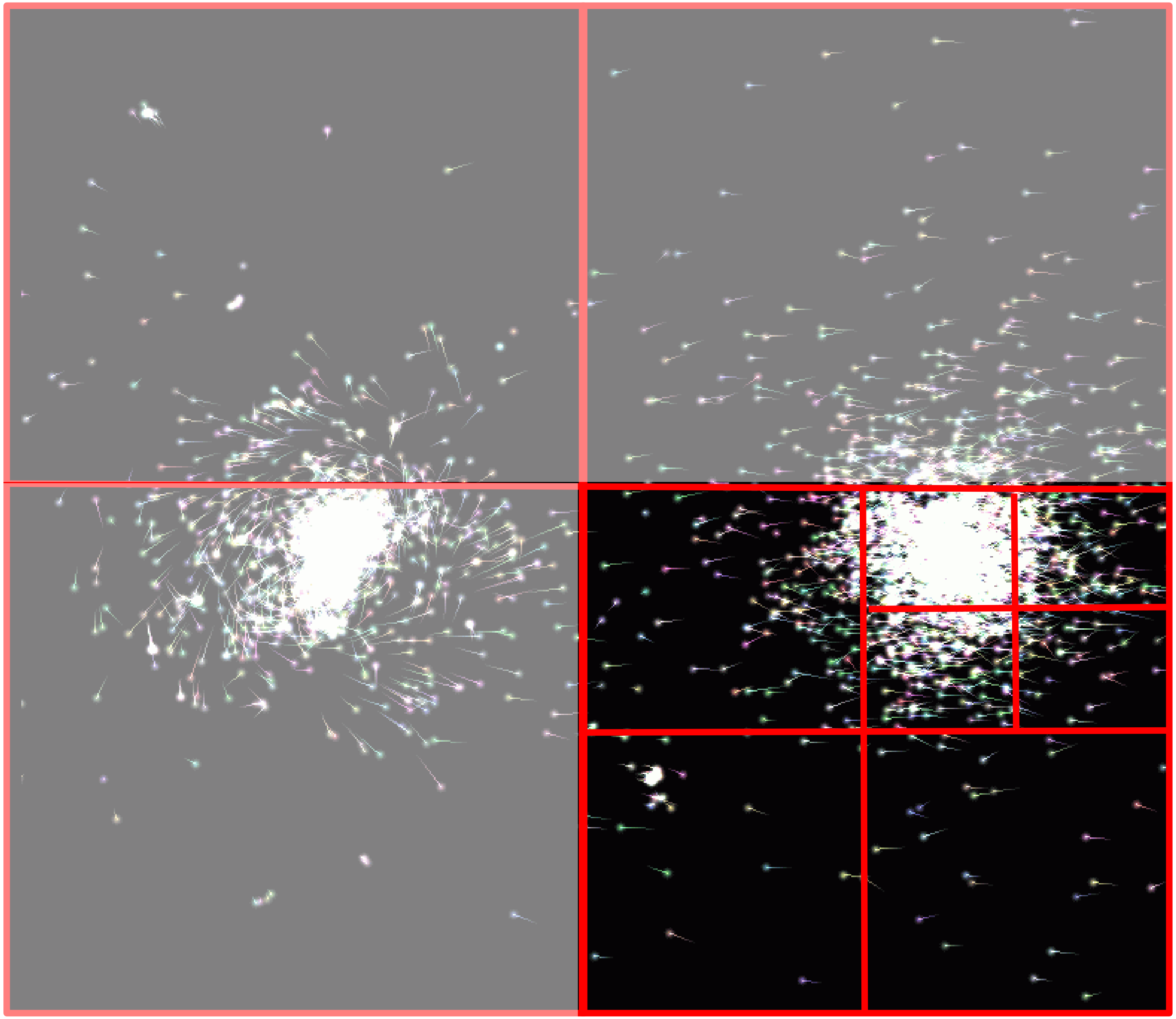
Barnes-Hut N-body simulation



Barnes-Hut N-body simulation



Barnes-Hut N-body simulation



Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;

  void updateBodies(BodyUpdates b, int iter) { // Global method.
    root.applyUpdates(b, iter); // Update bodies in our tree.
  }

  BodyTreeNode getRoot() { // Local method.
    return root;
  }
}

// Mark the computeForces method as a spawn operation.
interface BHSpawns extends satin.Spawnable {
  BodyUpdates computeForces(Subtree s, int iter, Bodies bodies);
}

class BarnesHut extends satin.SatinObject implements BHSpawns {
  boolean guard_computeForces(Subtree s, int iter, Bodies bodies) {
    return bodies.iter + 1 == iter;
  }

  // Spawnable method. The "bodies" parameter is a shared object.
  BodyUpdates computeForces(Subtree s, int iter, Bodies bodies) {
    if(s.hasNoChildren) {
      computeSequentially(s, iter, bodies.getRoot());
    } else { // Divide the work and spawn tasks (recursion step).
      for(int i=0; i<s.nrChildren; i++) {
        res[i] = computeForces(s.child[i], iter, bodies); // Spawn.
      }
      sync(); // Wait for the spawn operation to finish.

      return mergeSubresults(res); // Merge results and return.
    }
  }

  public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies(); // Create shared object.
    for (int iter = 0; iter < N; iter++) {
      results = bh.computeForces(root, iter, bodies); // Spawn.
      sync(); // Wait for the spawn operation to finish.
      bodies.update(results, iter); // Shared method invocation.
    }
  }
}
```

Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;

  void updateBodies(BodyUpdates b, int iter) { // Global method.
    root.applyUpdates(b, iter); // Update bodies in our tree.
  }

  BodyTreeNode getRoot() { // Local method.
```

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
    void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
    BodyTreeNode root;

    void updateBodies(BodyUpdates b, int iter) {
      root.applyUpdates(b, iter);
    }

    BodyTreeNode getRoot() {
        return root;
    }
}
```

Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;

  void updateBodies(BodyUpdates b, int iter) { // Global method.
    root.applyUpdates(b, iter); // Update bodies in our tree.
  }

  BodyTreeNode getRoot() { // Local method.
    return root;
  }
}

// Mark the computeForces method as a spawn operation.
interface BHSpawns extends satin.Spawnable {
  BodyUpdates computeForces(Subtree s, int iter, Bodies bodies);
}

class BarnesHut extends satin.SatinObject implements BHSpawns {
  boolean guard_computeForces(Subtree s, int iter, Bodies bodies) {
```


```
// Mark the computeForces method as a spawn operation.
interface BHSpawns extends satin.Spawnable {
  BodyUpdates computeForces(Subtree s, int iter, Bodies bodies);
}
```


```
      return mergeSubresults(res); // Merge results and return.
    }
  }

  public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies(); // Create shared object.
    for (int iter = 0; iter < N; iter++) {
      results = bh.computeForces(root, iter, bodies); // Spawn.
      sync(); // Wait for the spawn operation to finish.
      bodies.update(results, iter); // Shared method invocation.
    }
  }
}
```

Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;
```

```
public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies();
    for (int iter = 0; iter < N; iter++) {
        results = bh.computeForces(root, iter, bodies);
        sync();
        bodies.updateBodies(results, iter);
    }
}
```

```
      }
      sync(); // Wait for the spawn operation to finish.

      return mergeSubresults(res); // Merge results and return.
    }
  }
```

```
  public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies(); // Create shared object.
    for (int iter = 0; iter < N; iter++) {
      results = bh.computeForces(root, iter, bodies); // Spawn.
      sync(); // Wait for the spawn operation to finish.
      bodies.update(results, iter); // Shared method invocation.
    }
  }
}
```

Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;

  void updateBodies(BodyUpdates b, int iter) { // Global method
```

```
BodyUpdates computeForces(Subtree s, int iter, Bodies bodies) {
    if(s.hasNoChildren) {
        computeSequentially(s, iter, bodies.getRoot());
    } else {
        for(int i=0; i<s.nrChildren; i++) {
           res[i] = computeForces(s.child[i], iter, bodies);
        }
        sync();

        return mergeSubresults(res);
    }
}
```

```
  }

  public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies(); // Create shared object.
    for (int iter = 0; iter < N; iter++) {
      results = bh.computeForces(root, iter, bodies); // Spawn.
      sync(); // Wait for the spawn operation to finish.
      bodies.update(results, iter); // Shared method invocation.
    }
  }
}
```

ibis

vrije Universiteit

Barnes-Hut Nbody code

```
// Marker interface that defines updateBodies as a global method.
interface BodiesInterface extends satin.GlobalMethods {
  void updateBodies(BodyUpdates b, int iter);
}

// A shared object containing the tree of bodies.
class Bodies extends satin.SharedObject implements BodiesInterface {
  BodyTreeNode root;

  void updateBodies(BodyUpdates b, int iter) { // Global method.
    root.applyUpdates(b, iter); // Update bodies in our tree.
```

```
boolean guard_computeForces(Subtree s, int iter, Bodies bodies) {
  return iter == bodies.iter + 1;
}
```

```
}

class BarnesHut extends satin.SatinObject implements BHSpawns {
  boolean guard_computeForces(Subtree s, int iter, Bodies bodies) {
    return bodies.iter + 1 == iter;
  }

  // Spawnable method. The "bodies" parameter is a shared object.
  BodyUpdates computeForces(Subtree s, int iter, Bodies bodies) {
    if(s.hasNoChildren) {
      computeSequentially(s, iter, bodies.getRoot());
    } else { // Divide the work and spawn tasks (recursion step).
      for(int i=0; i<s.nrChildren; i++) {
        res[i] = computeForces(s.child[i], iter, bodies); // Spawn.
      }
      sync(); // Wait for the spawn operation to finish.

      return mergeSubresults(res); // Merge results and return.
    }
  }

  public static void main(String[] args) {
    BarnesHut bh = new BarnesHut();
    Bodies bodies = new Bodies(); // Create shared object.
    for (int iter = 0; iter < N; iter++) {
      results = bh.computeForces(root, iter, bodies); // Spawn.
      sync(); // Wait for the spawn operation to finish.
      bodies.update(results, iter); // Shared method invocation.
    }
  }
}
```

Satin Applications

- VLSI routing
- Satisfiability solver
- Gene sequencing
- N-body simulations
- Grammar-based text analysis
- Game-tree search
- Raytracing
- Numerical functions
- ...

vrije Universiteit

Conclusion

- Satin provides a powerful programming model
- Extremely easy to use
- Satin is optimized for grid applications
- Satin allows applications to transparently deal with grid issues
  - load balancing, malleability, migration, fault-tolerance, adaptivity, firewalls, heterogeneity

ibis

vrije Universiteit