Javelin++: Scalability Issues in Global Computing

Michael O. Neary, Sean P. Brydon, Paul Kmiec,
Sami Rollins, and Peter Cappello

Department of Computer Science
University of California, Santa Barbara

ACM Java Grande
San Francisco, CA, June 13, 1999

A Successful Global Computing Application

Taken from the SETI@home website:

- To our users: SETI@home is off to a great start! We now have over 500,000 users, many more than we expected. This has caused some temporary problems:
  - Our servers are occasionally overloaded. If you get a "Can't connect to server" message, the problem is probably at our end, not yours.
  - ...

Motivation (cont.)

- Common issues:
  - Scalability
  - Fault Tolerance
  - Correctness
  - Incentive
- All dealt with in application-specific manner
- Javelin++ Goal:
  - Provide a scalable, unified framework to support such applications

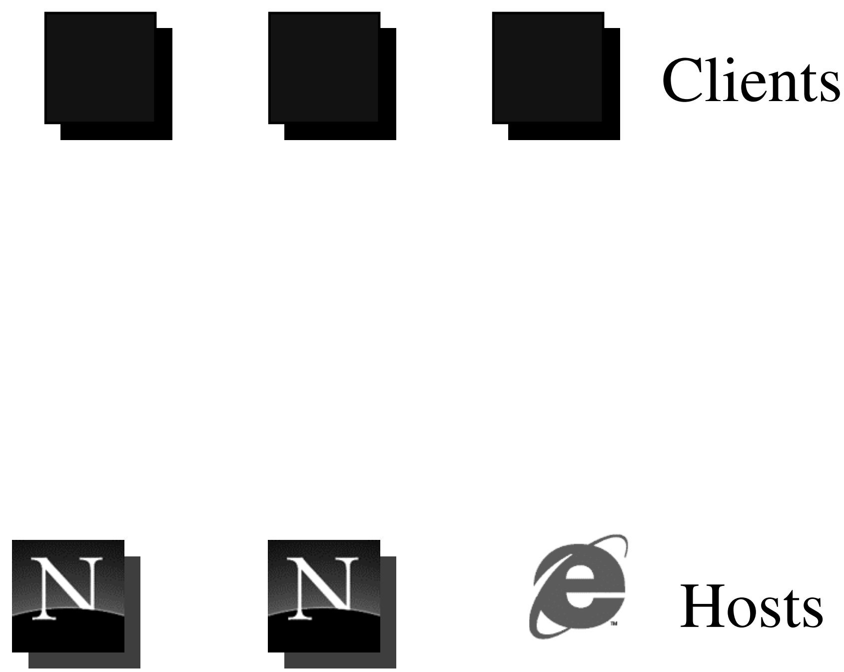
Outline

- Javelin Recap
- Javelin++ Enhancements
- Code Distribution Scalability
- Execution Scalability
- Experimental Results
- Fault Tolerance: Eager Scheduling
- Future Work & Conclusion

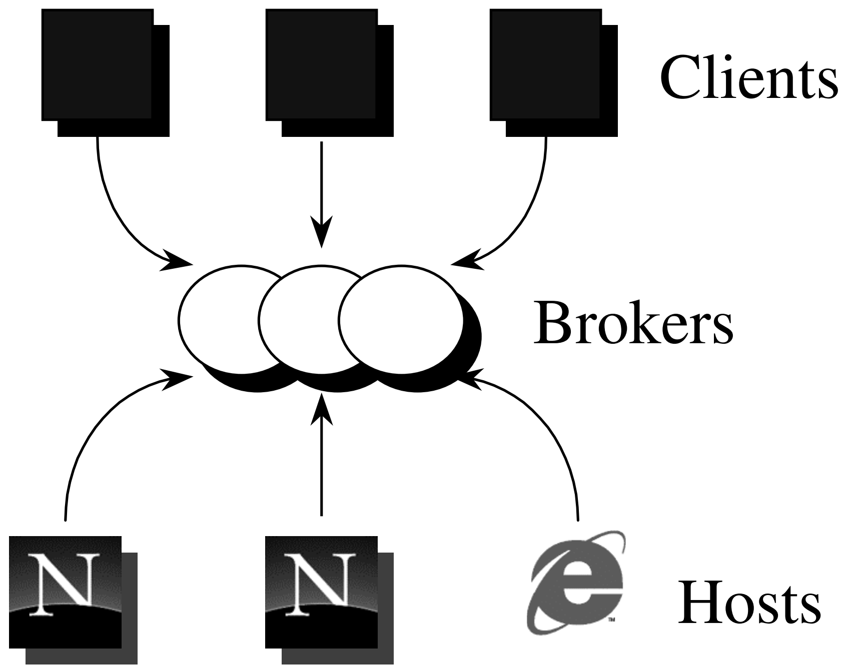
Javelin: A Java-Based Infrastructure

Clients

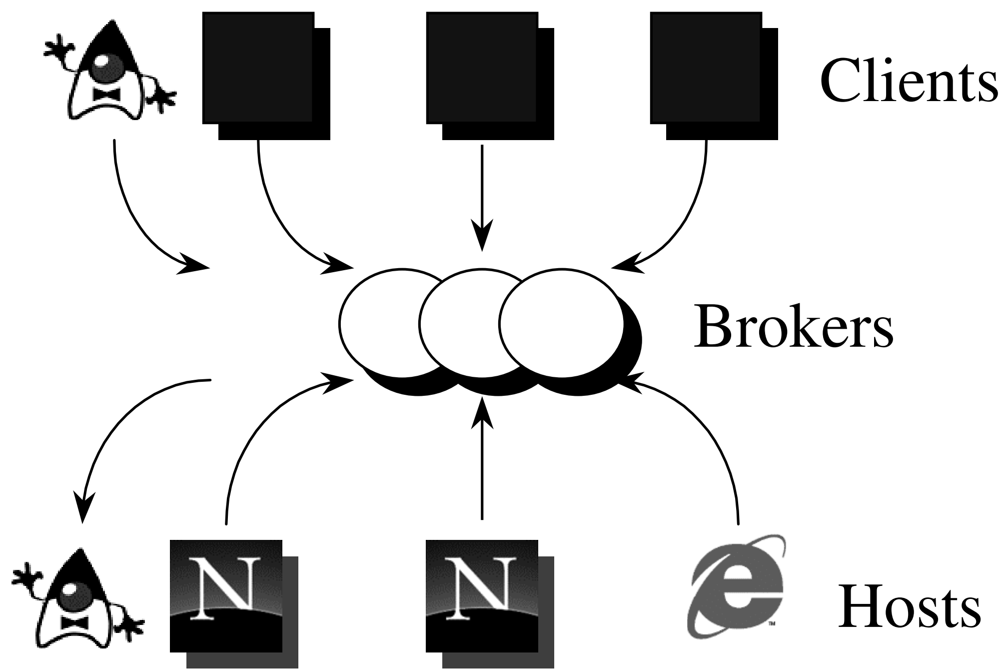
Javelin: A Java-Based Infrastructure



Javelin: A Java-Based Infrastructure



Javelin: A Java-Based Infrastructure

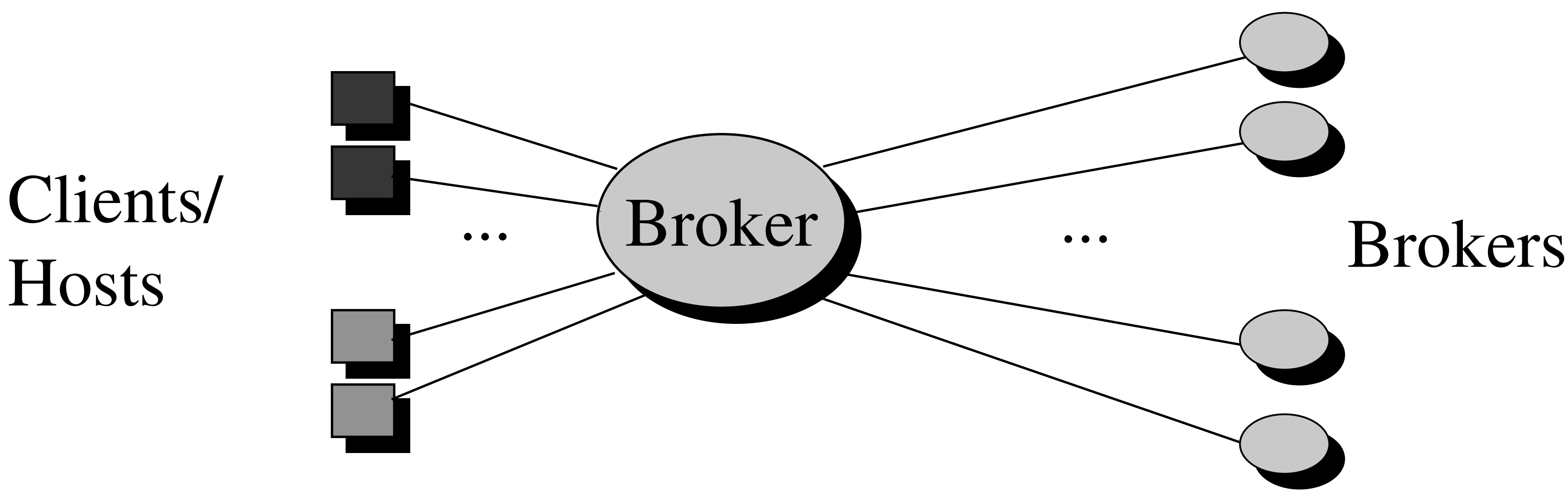


Javelin++ Enhancements

- Broker network
- 2 different work stealing schedulers
  - probabilistic & deterministic
- Eager scheduling
  - application transparent fault tolerance
- Javelin++ API
- Java applications, Java RMI

Network Structure

- fixed number of clients/hosts per broker
- graph of brokers with bounded degree
- potentially unlimited number of hosts

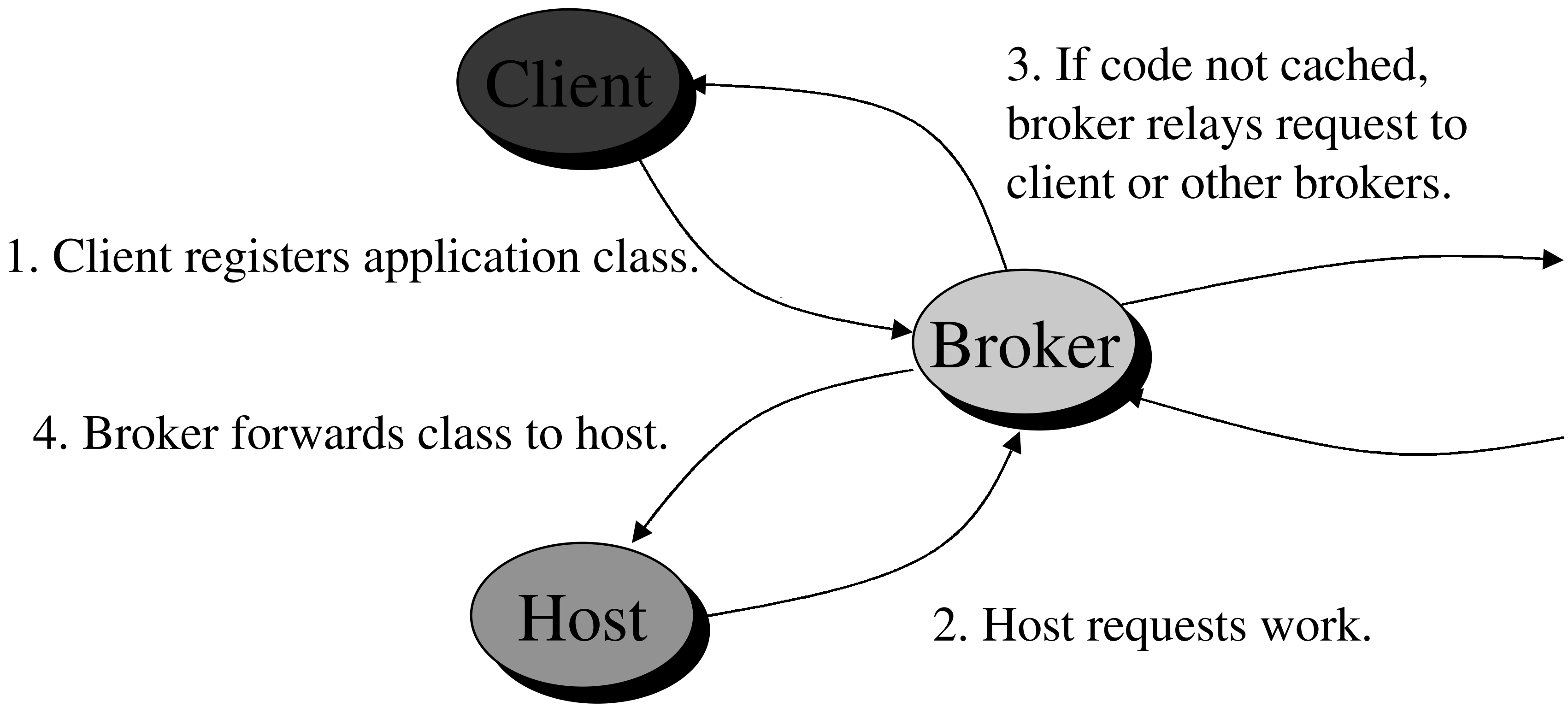


Key Scalability Problems

- Host allocation and code distribution
  - How do clients find hosts ?
  - How does code get shipped ?
- Data communication at runtime
  - How is data exchanged between hosts at execution time ?

Code Distribution Scalability

- Implementation extends the *ClassLoader*, uses Java RMI for code shipping



Outline

- Javelin Recap
- Javelin++ Enhancements
- Code Distribution Scalability
- Execution Scalability
- Experimental Results
- Fault Tolerance: Eager Scheduling
- Future Work & Conclusion

Execution Scalability

- Programming model:
  - Adaptive Parallelism
  - Master-Worker with Subcontracting
- Cilk-like Work Stealing
- Main data structure local to each host:
  - Double Ended Queue (*Deque*)
  - Local host picks off chunks on one side of deque
  - Work stealing requests pick off chunks at other side

Deterministic Work Stealing

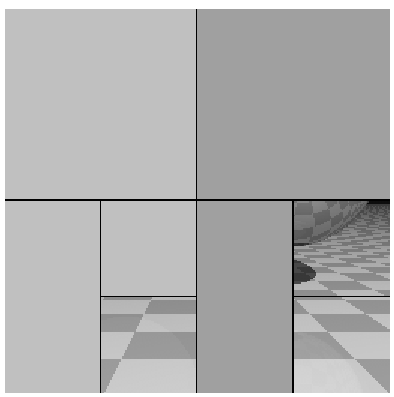
- Hosts are organized in a *balanced tree*:
  - client located at the root
  - client maintains data structure for host tree
  - new hosts are assigned leaf position
  - tree fanout chosen by application
- Work counter heuristic:
  - each host maintains counter for itself & its children
  - upper bound estimate for work left in subtree

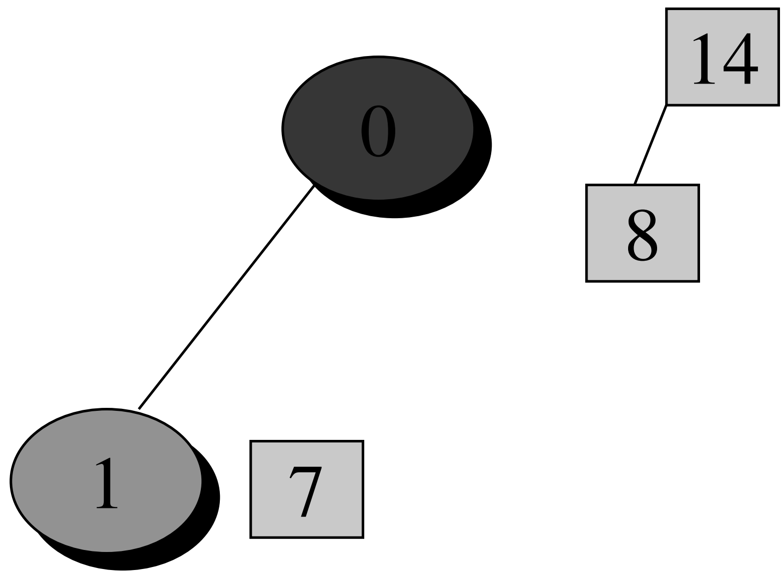
Deterministic Work Stealing

- Client (Host 0) starts computation

Deterministic Work Stealing

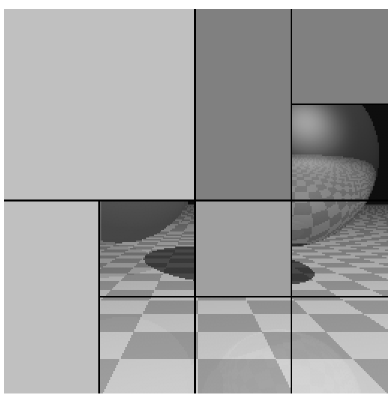
- Host 1 joins computation

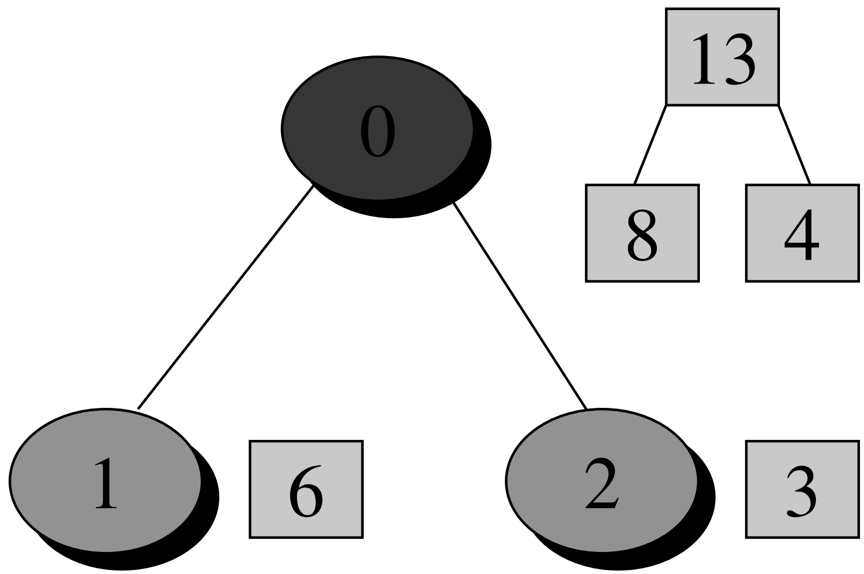




Deterministic Work Stealing

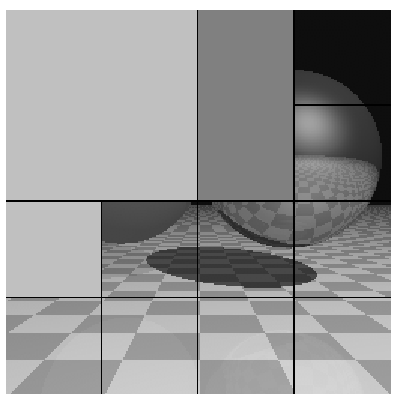
- Host 2 joins computation

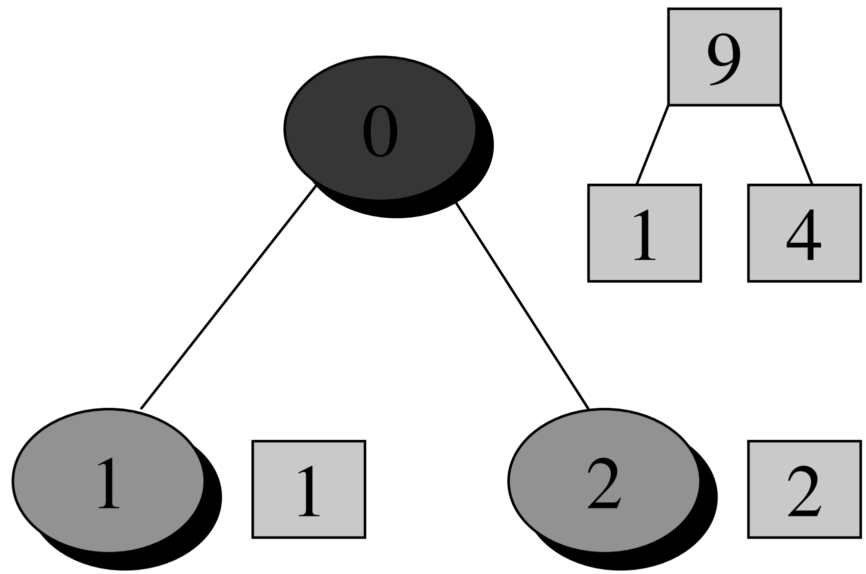




Deterministic Work Stealing

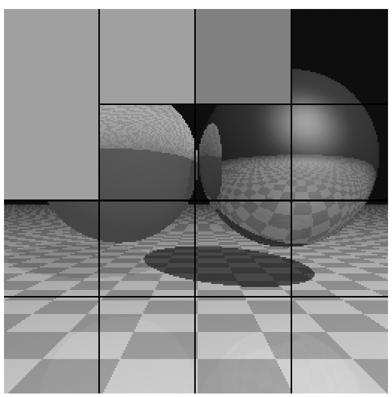
- Client steals work from host 1

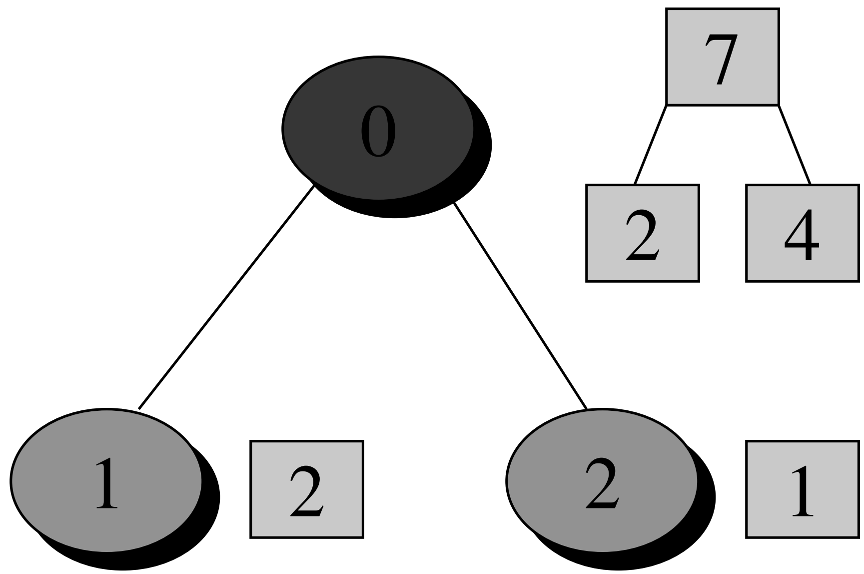




Deterministic Work Stealing

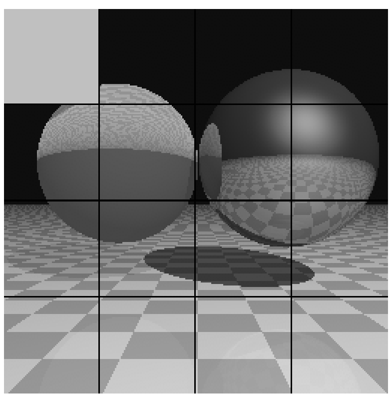
- Host 1 steals work from client

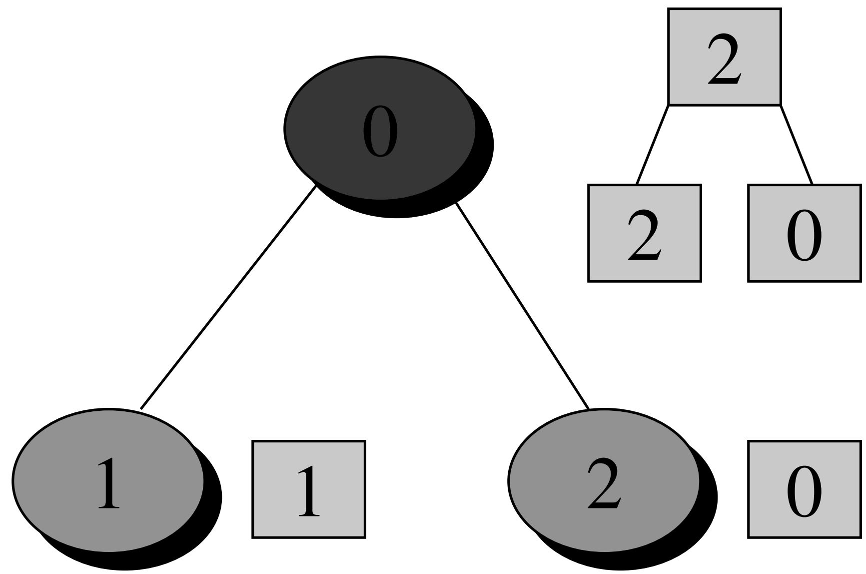




Deterministic Work Stealing

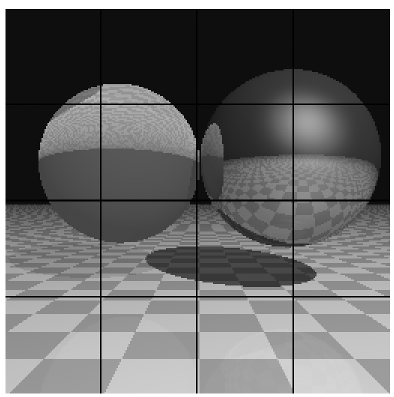
- Host 2 requests work from client

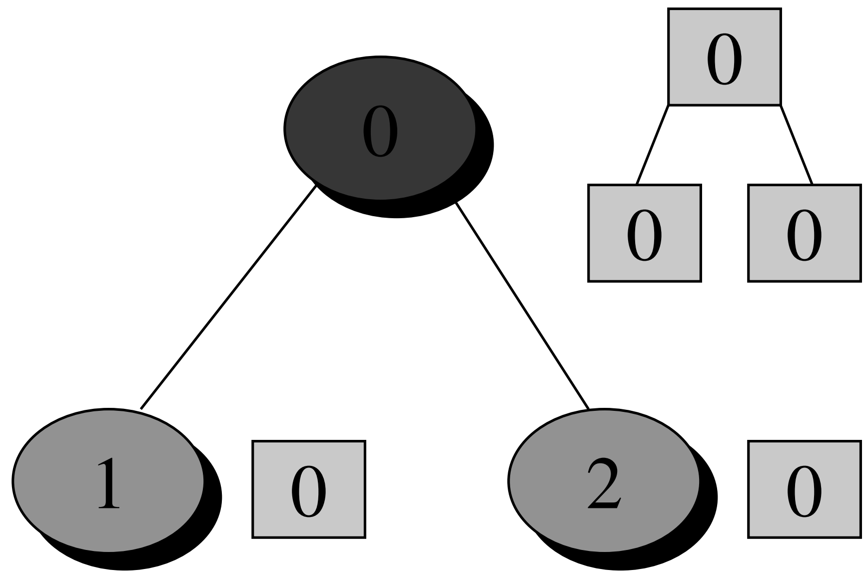




Deterministic Work Stealing

- Picture complete!





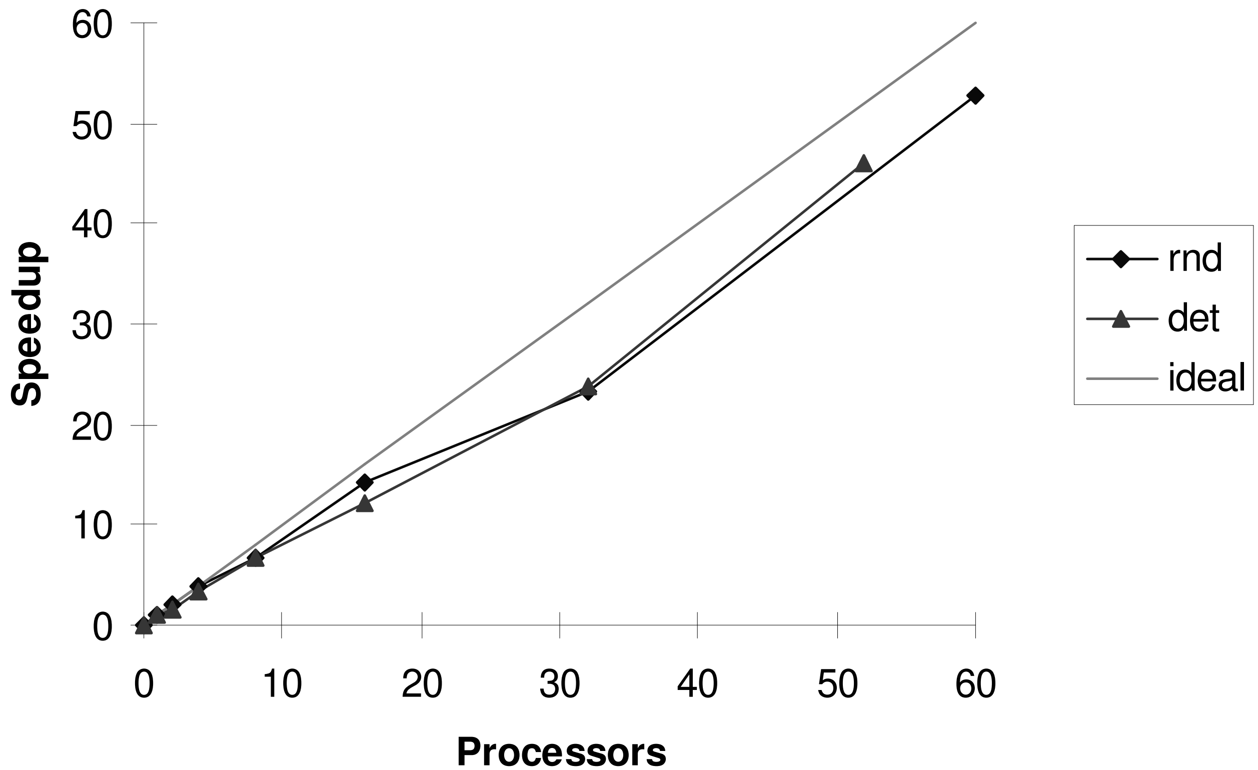
Probabilistic Work Stealing

- When local deque is empty:
  - host selects *random target* from list of hosts
  - repeat until steal request successful
- Host addresses piggybacked on every message
  - provides information exchange
- Disadvantage at end of computation:
  - chance of hitting work decreases as number of hosts increases
  - generates many useless messages

Experimental Setup

- Environment:
  - 12 Pentium II 350 MHz
  - 41 Pentium 166 MHz
  - 10 UltraSparc (143 - 400 MHz)
- Raytracing application:
  - 994 graphical objects
  - image size 1024x1024 pixels
  - min chunk size 32x32 pixels = 1024 chunks
- Machines & network under normal workload

Experimental Results



Outline

- Javelin Recap
- Javelin++ Enhancements
- Code Distribution Scalability
- Execution Scalability
- Experimental Results
- Fault Tolerance: Eager Scheduling
- Future Work & Conclusion

Eager Scheduling

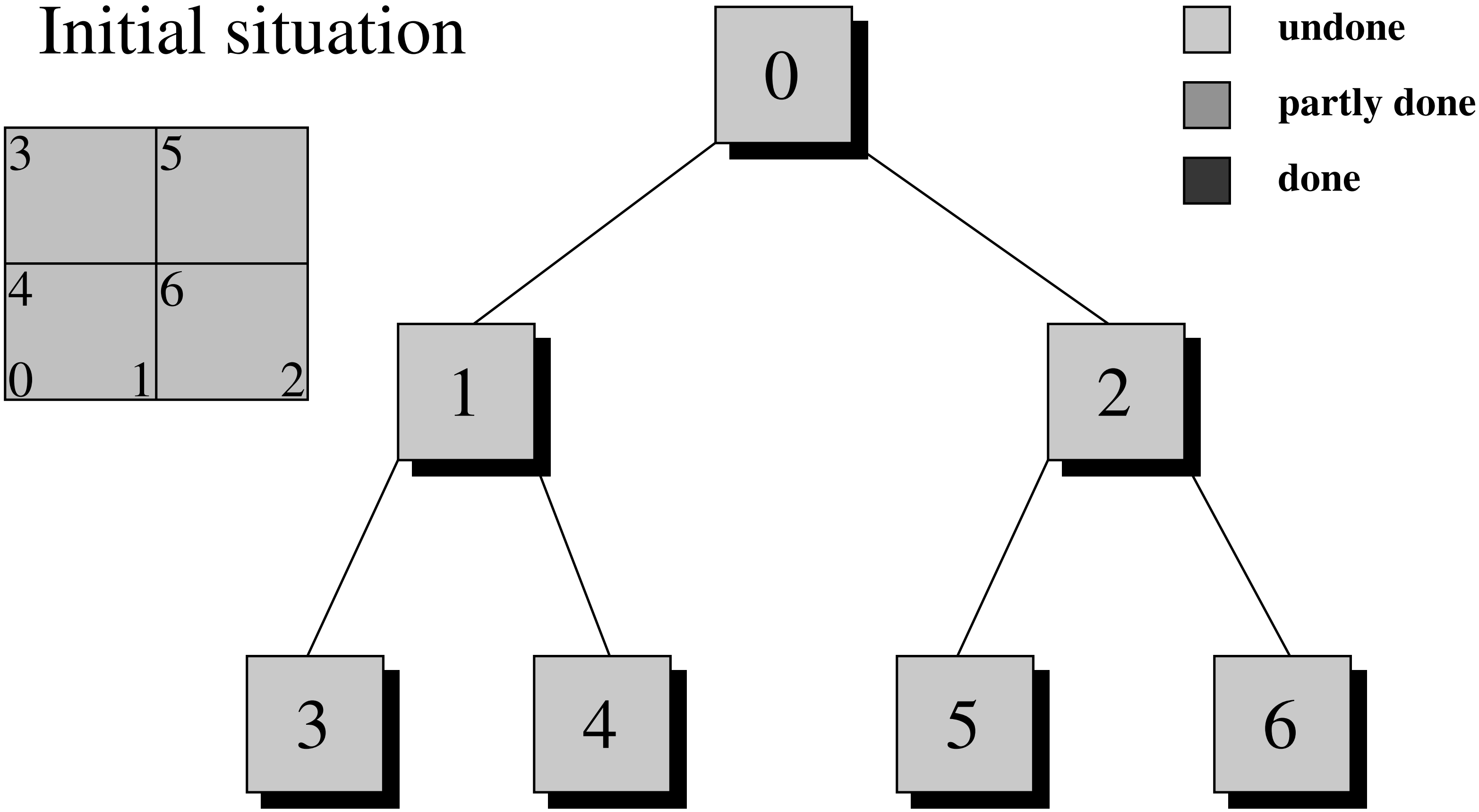
- Principle:
  - faster processors subsume work of slower processors
- Used successfully in *Charlotte* (NYU)
- Properties:
  - Low overhead in absence of failures
  - Load balancing & fits adaptive model
  - Robust: sustains n-1 failures among n hosts

Javelin++ Eager Scheduling

- Client maintains *problem tree*
  - atomic chunks at leaf nodes
- When work stealing fails:
  - undone problem chunks are selected from tree and rescheduled
  - larger undone chunks have priority
- Result processing is done bottom-up

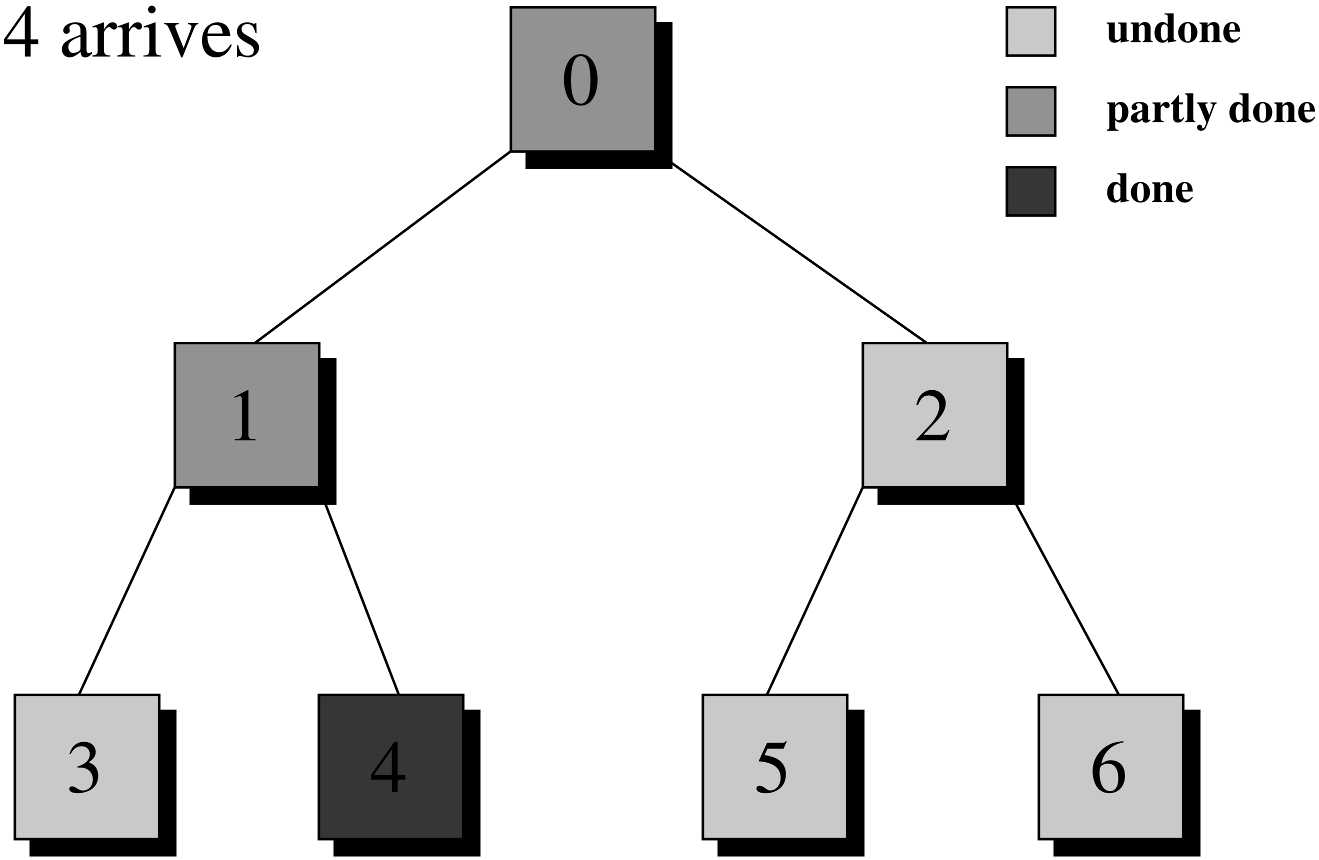
Eager Scheduling Example

Initial situation



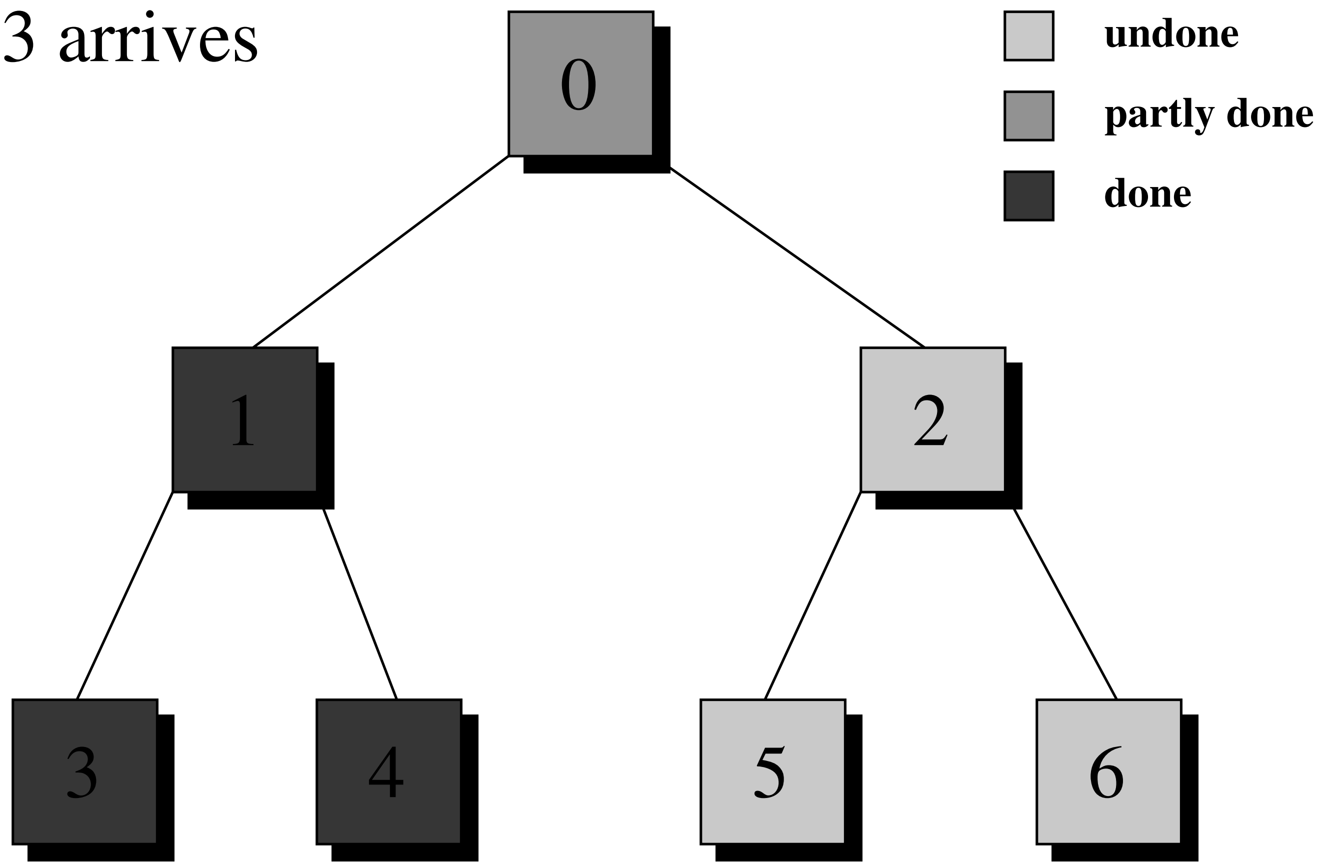
Eager Scheduling Example

Result 4 arrives



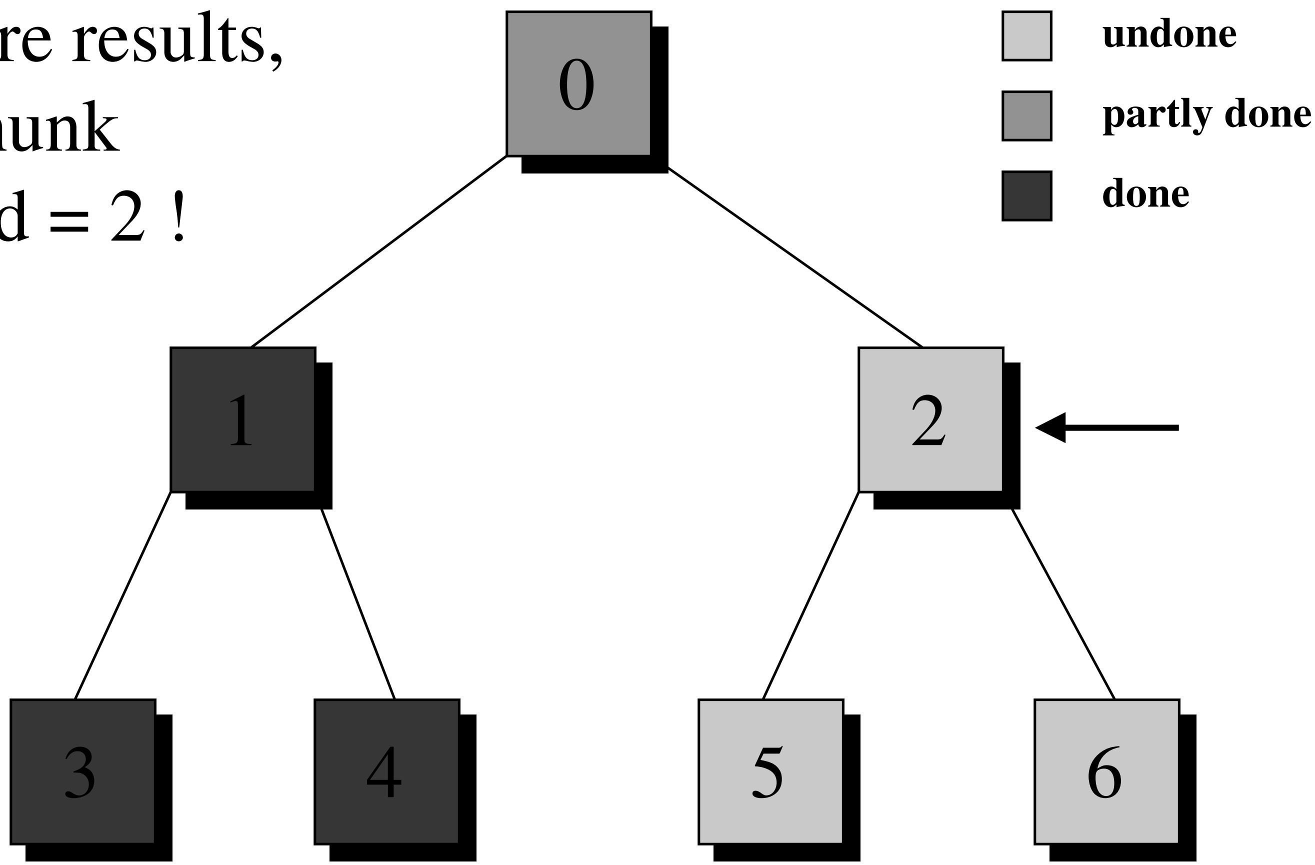
Eager Scheduling Example

Result 3 arrives



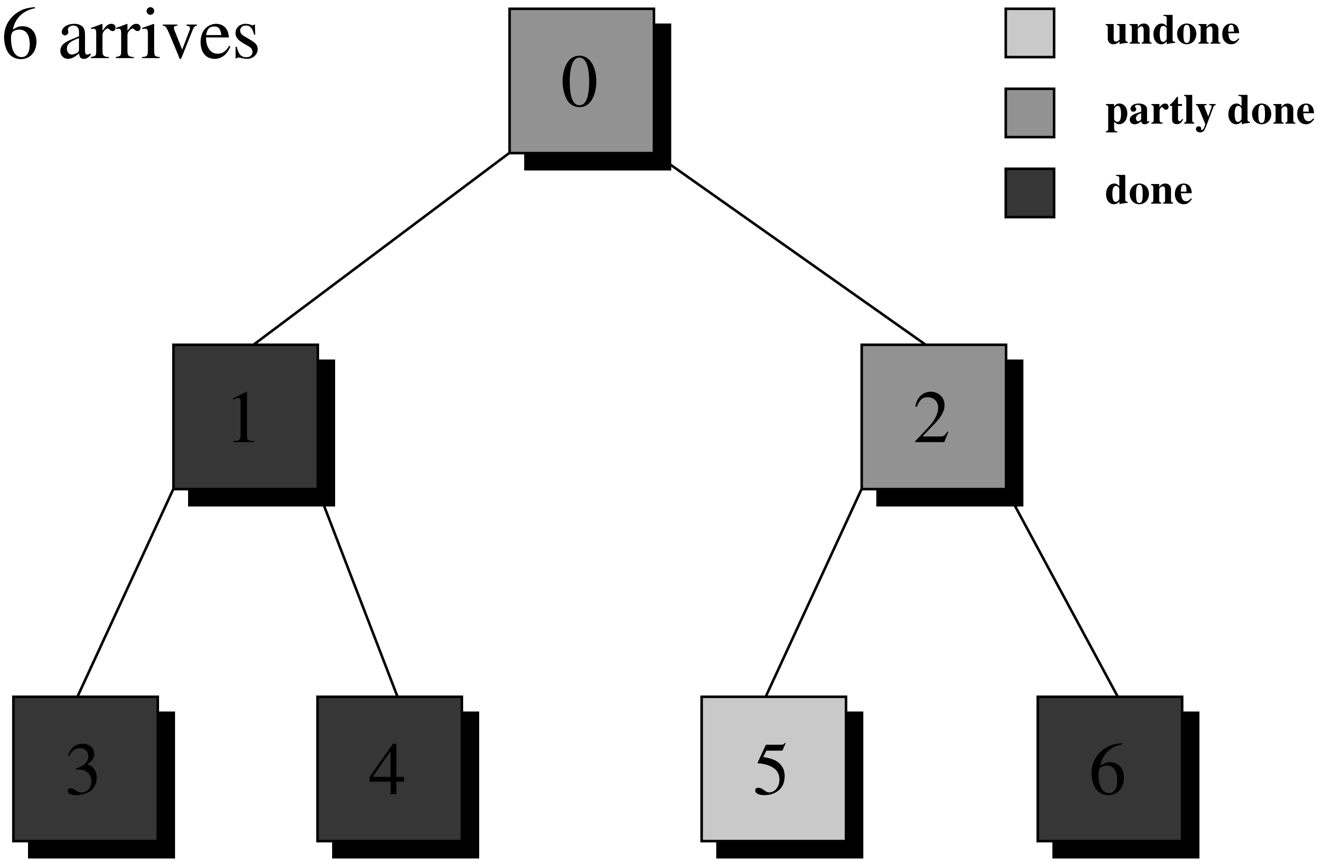
Eager Scheduling Example

No more results,
next chunk
selected = 2 !



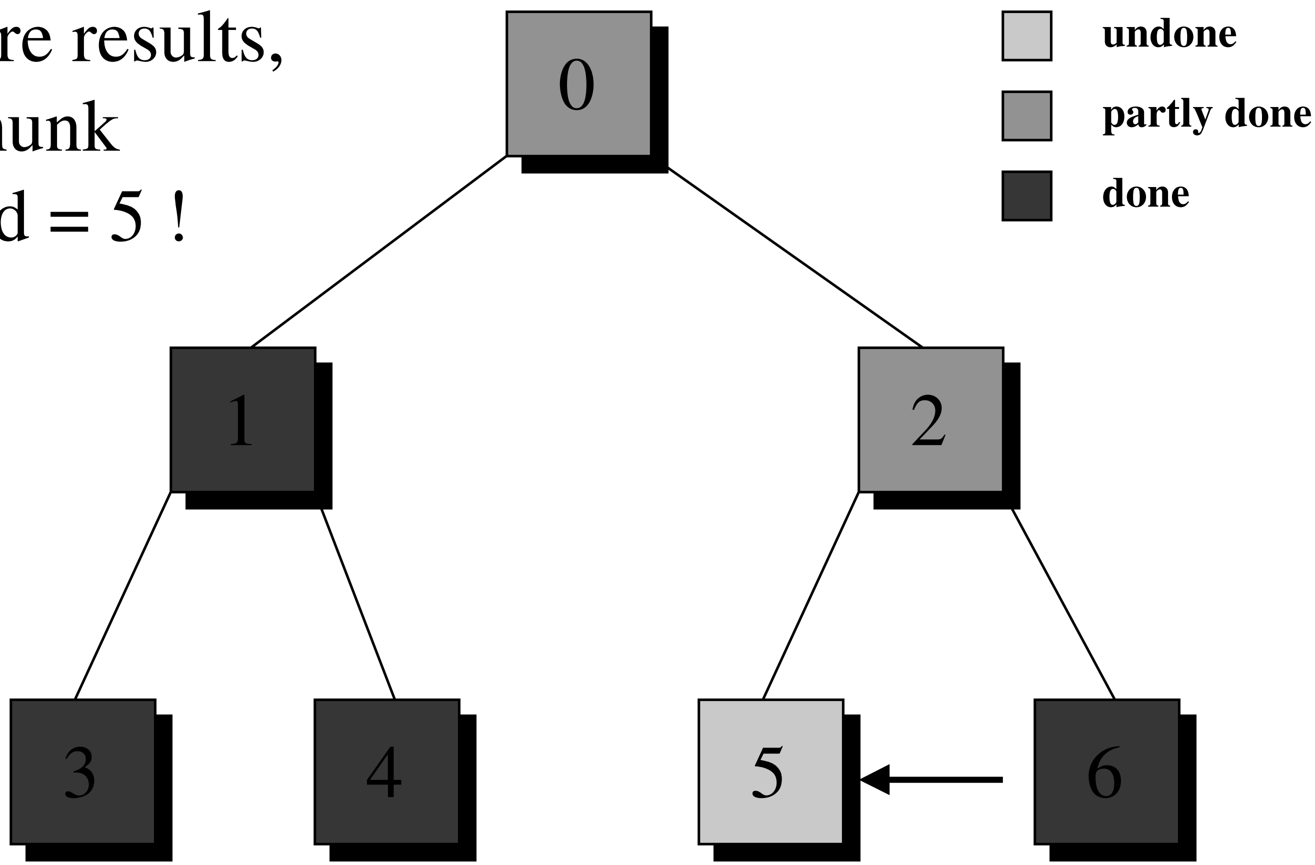
Eager Scheduling Example

Result 6 arrives

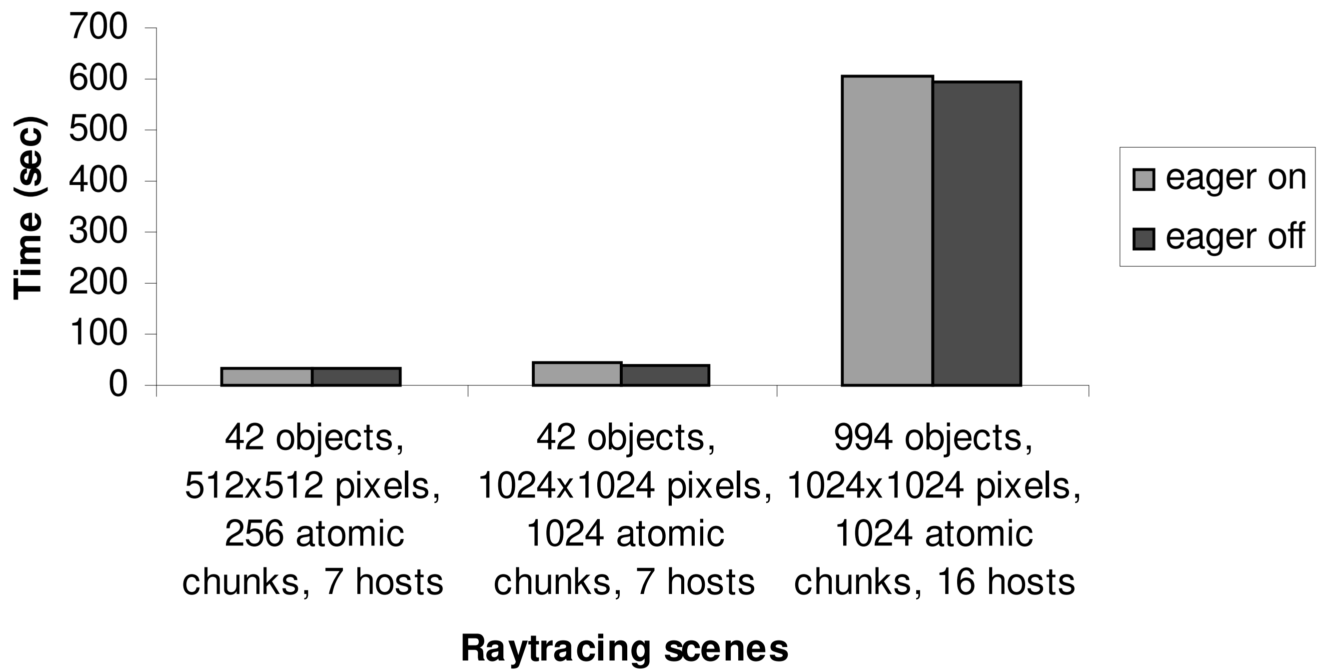


Eager Scheduling Example

No more results,
next chunk
selected = 5 !

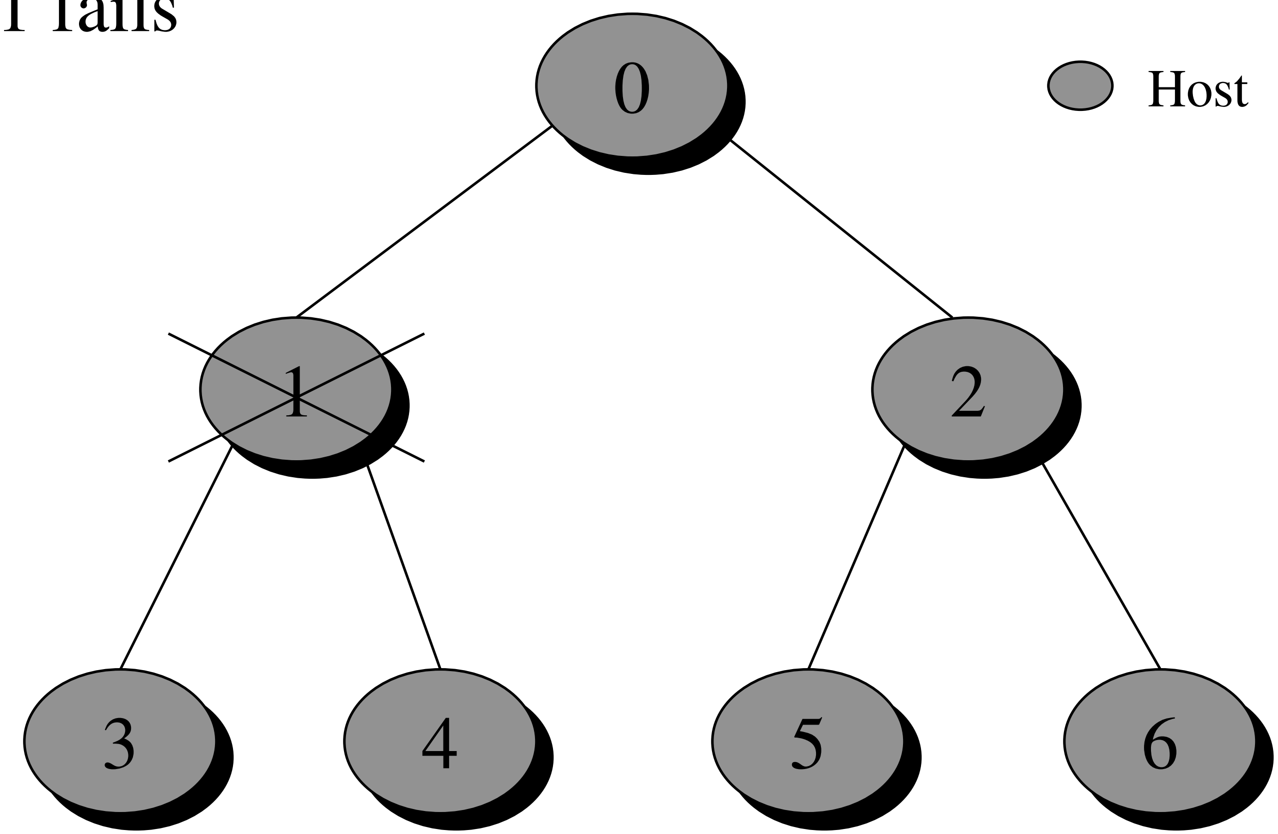


Eager Scheduling Overhead



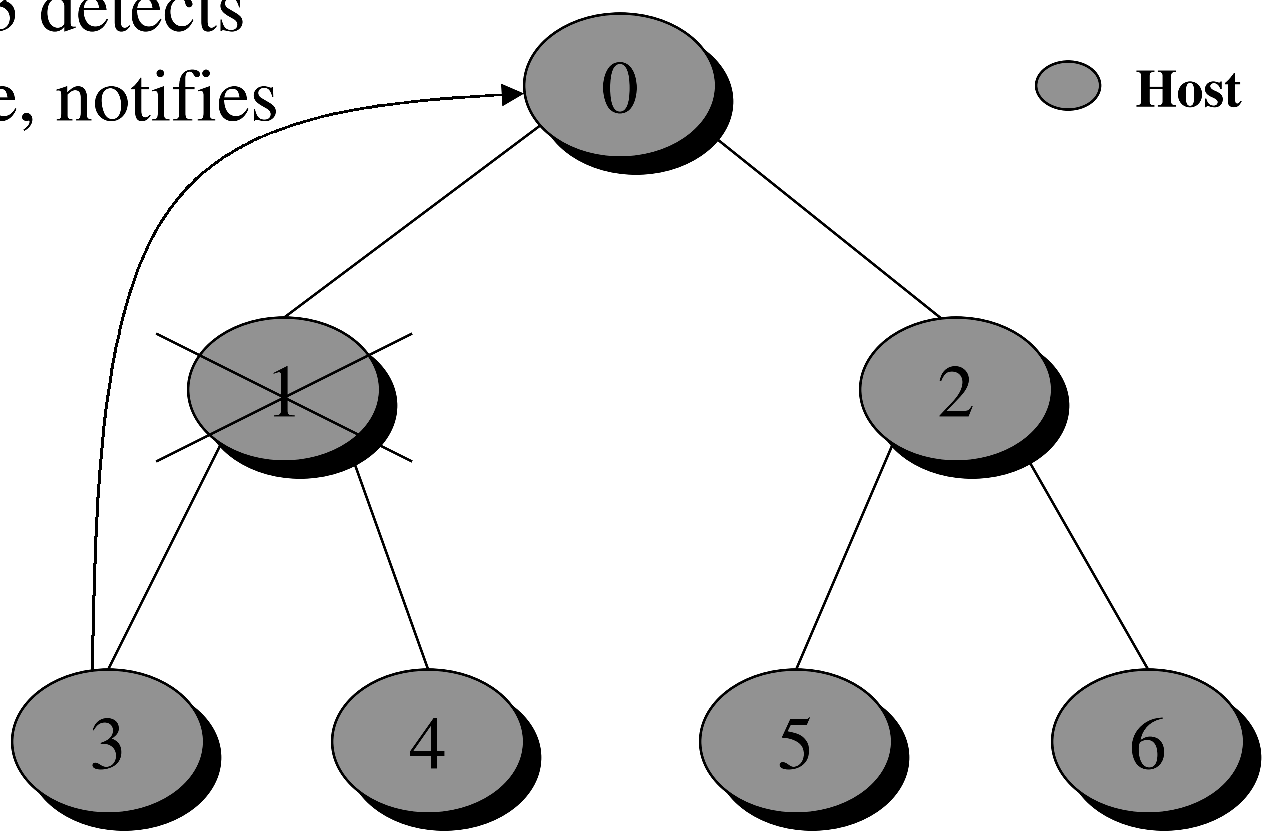
Fixing the Deterministic Tree

Host 1 fails



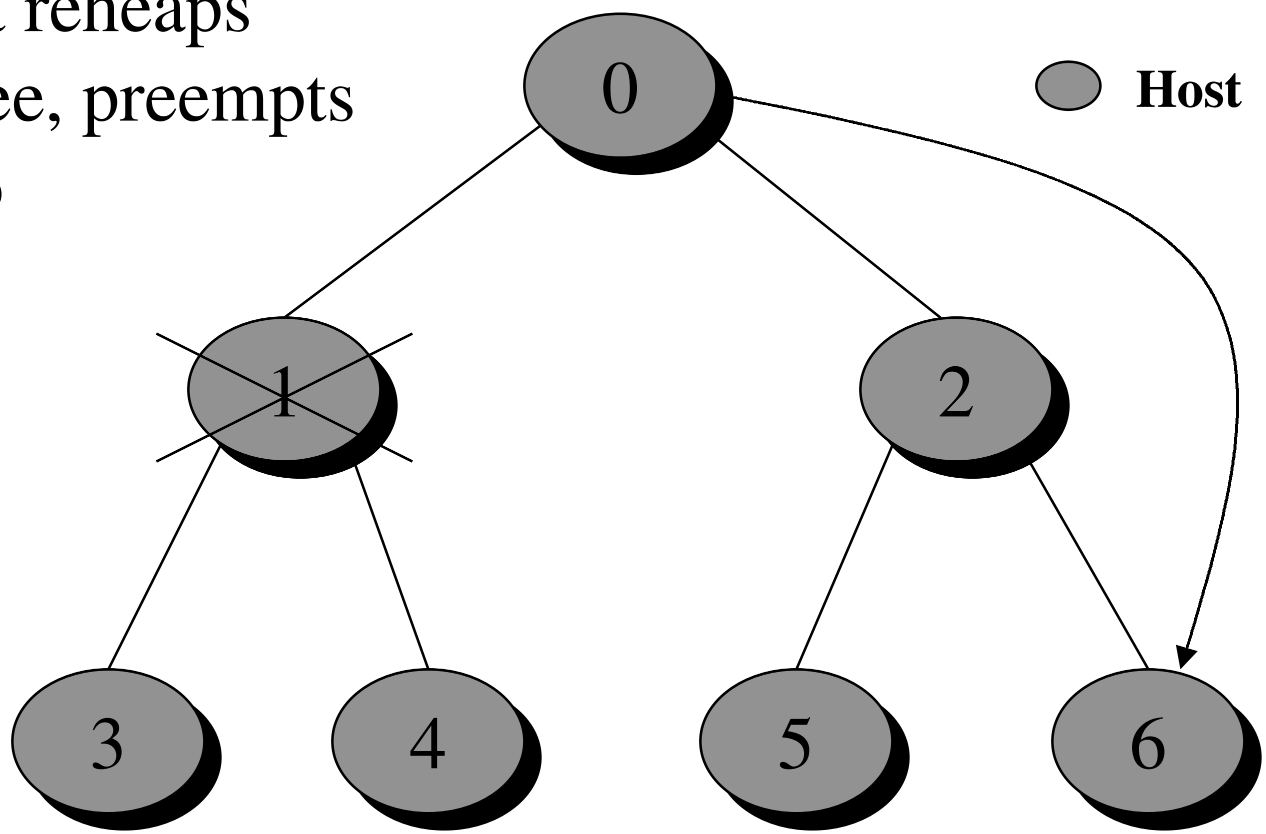
Fixing the Deterministic Tree

Host 3 detects failure, notifies client



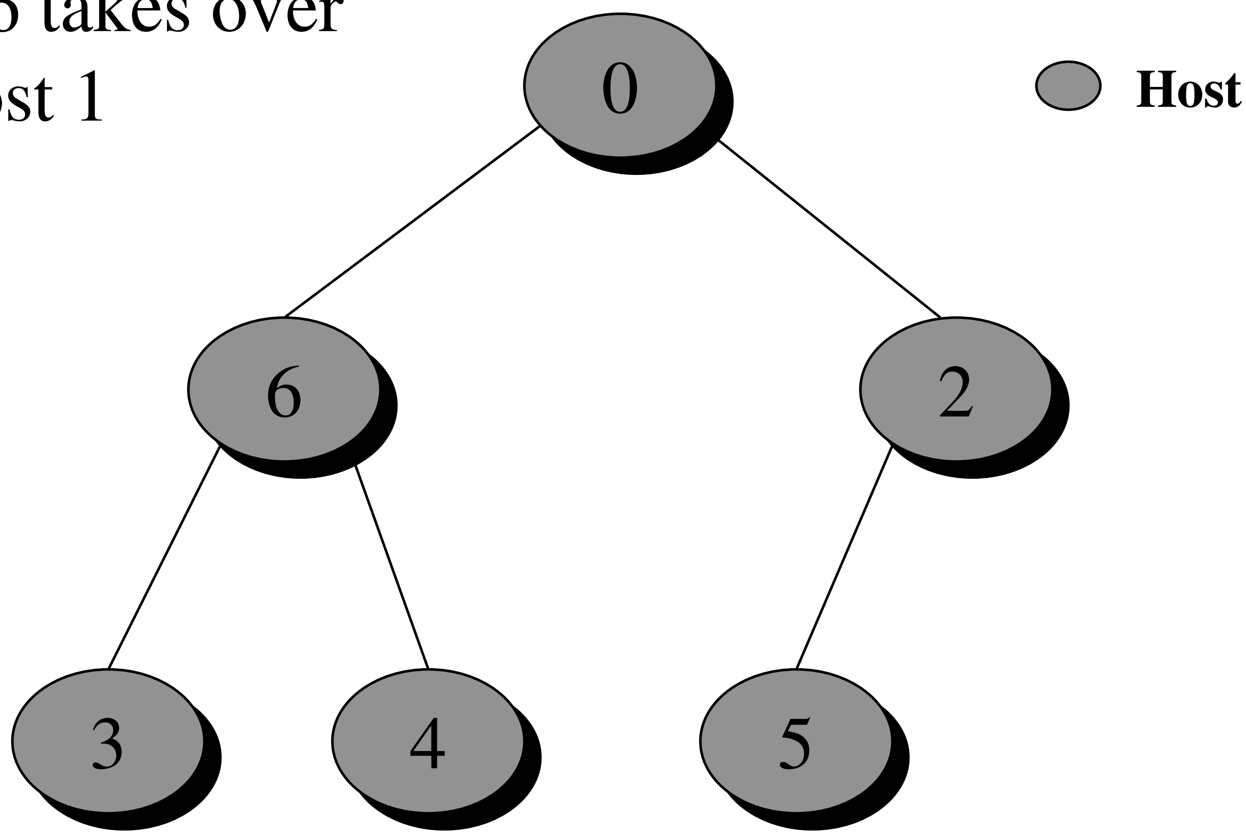
Fixing the Deterministic Tree

Client reheaps the tree, preempts host 6



Fixing the Deterministic Tree

Host 6 takes over for host 1



Conclusion

- Javelin++ offers various scalability and fault tolerance enhancements over Javelin
- Experimental results validate our approach on a small scale
- Once fault tolerance is fully achieved, large scale experiments will be possible