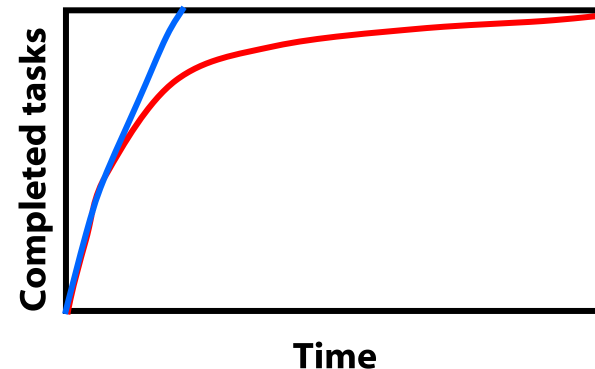
Loose Synchronization for Large-Scale Networked Systems

Jeannie Albrecht, Christopher Tuttle,
Alex C. Snoeren, and Amin Vahdat

June 3, 2006

Introduction

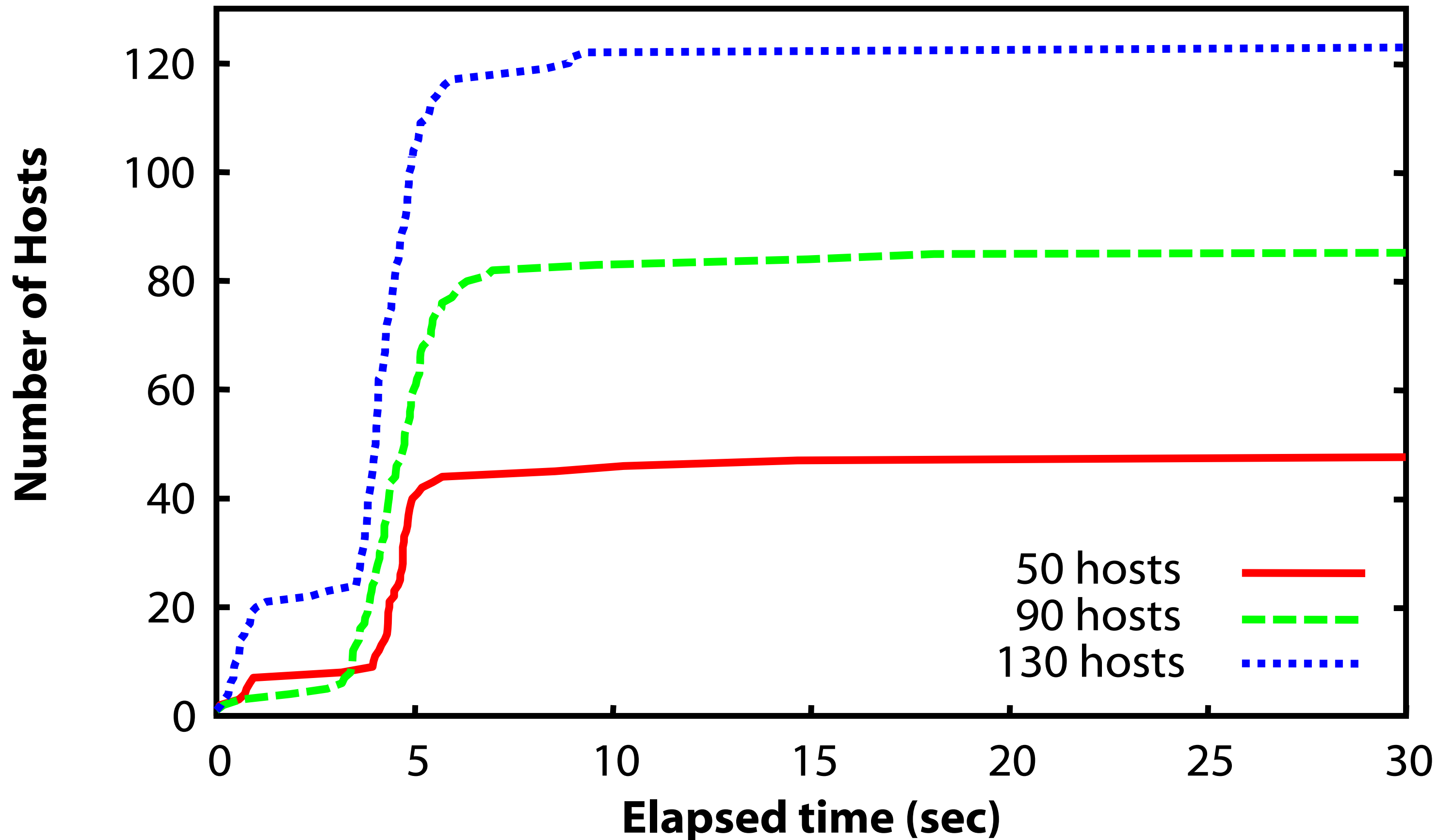
- Challenge: Controlling distributed applications in heterogeneous, failure-prone networked environments
  - Distributed systems tend to be volatile
  - Unpredictable network congestion
  - All hosts do not perform equally
  - Performance suffers due to small set of slow hosts/links



- Contribution: Unified abstraction for managing unpredictable failures and performance variation
  - Detect slow/failed hosts, remap computations, etc.

Bullet

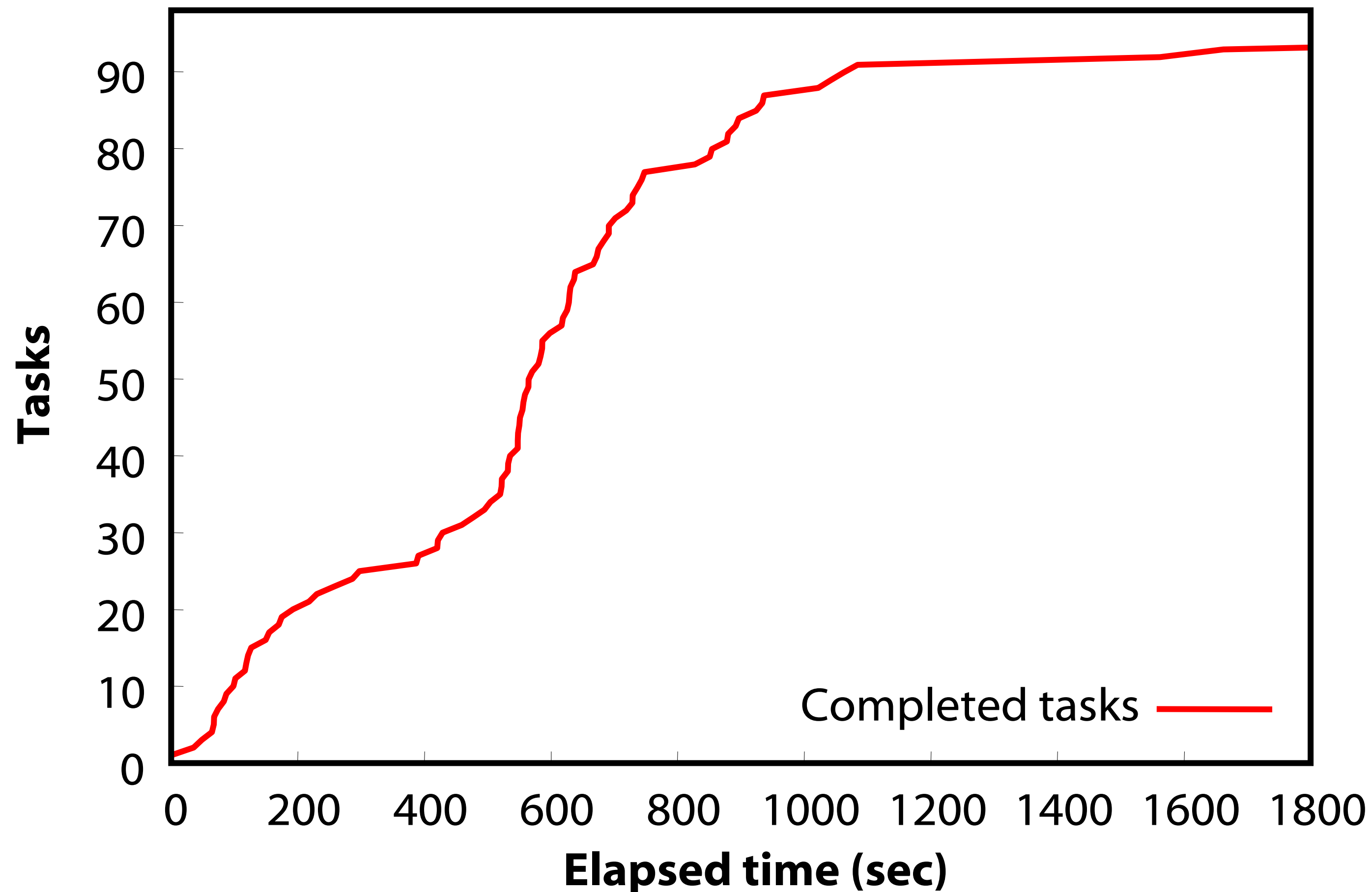
[SOSP 03]



- Overlay based file distribution
- 30+ seconds for 50/90/130 hosts to connect

EMAN

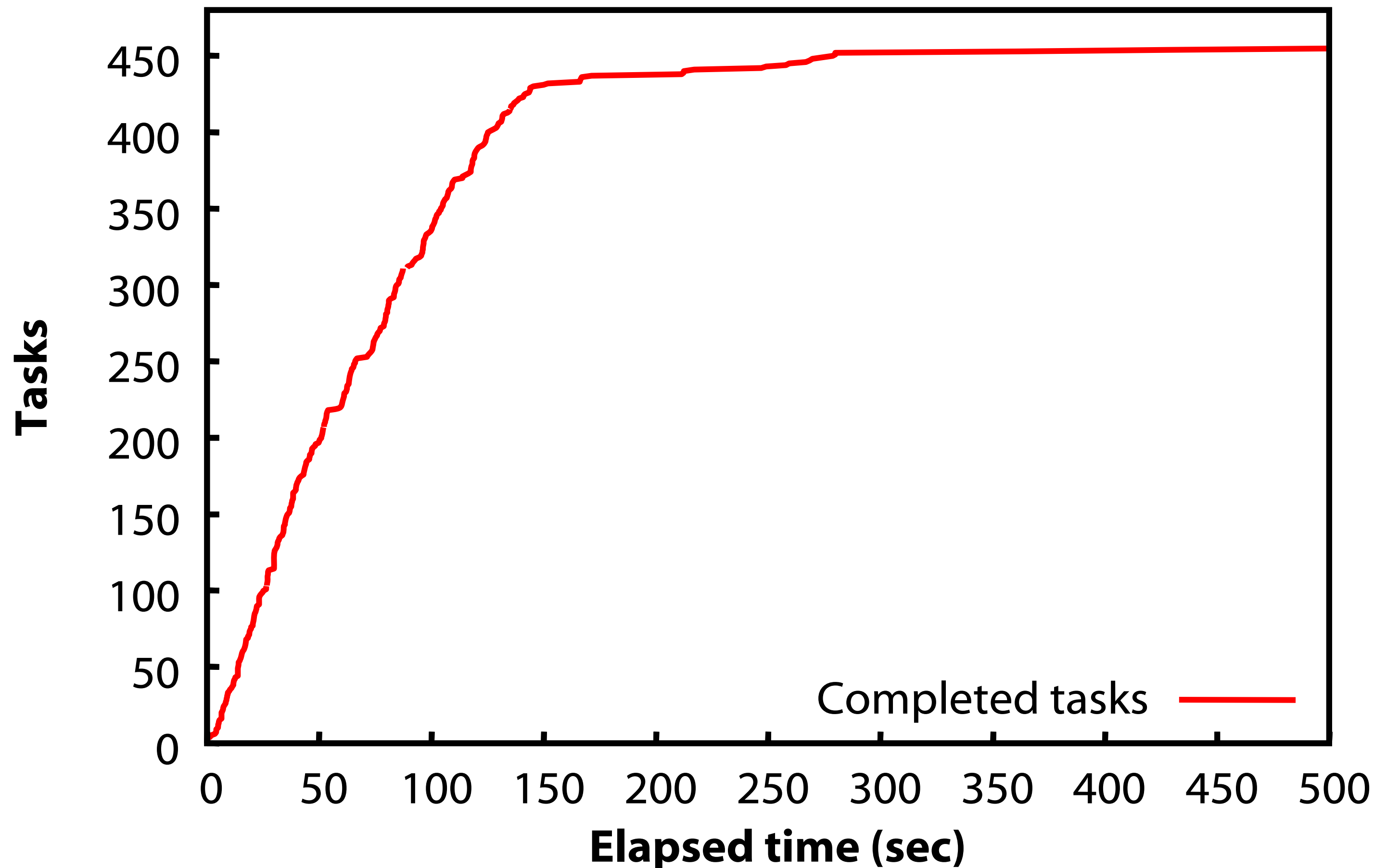
[JSB 99]



- Electron Micrograph Analysis
- 2700+ seconds to complete 98 tasks on 98 hosts

MapReduce

[OSDI 04]



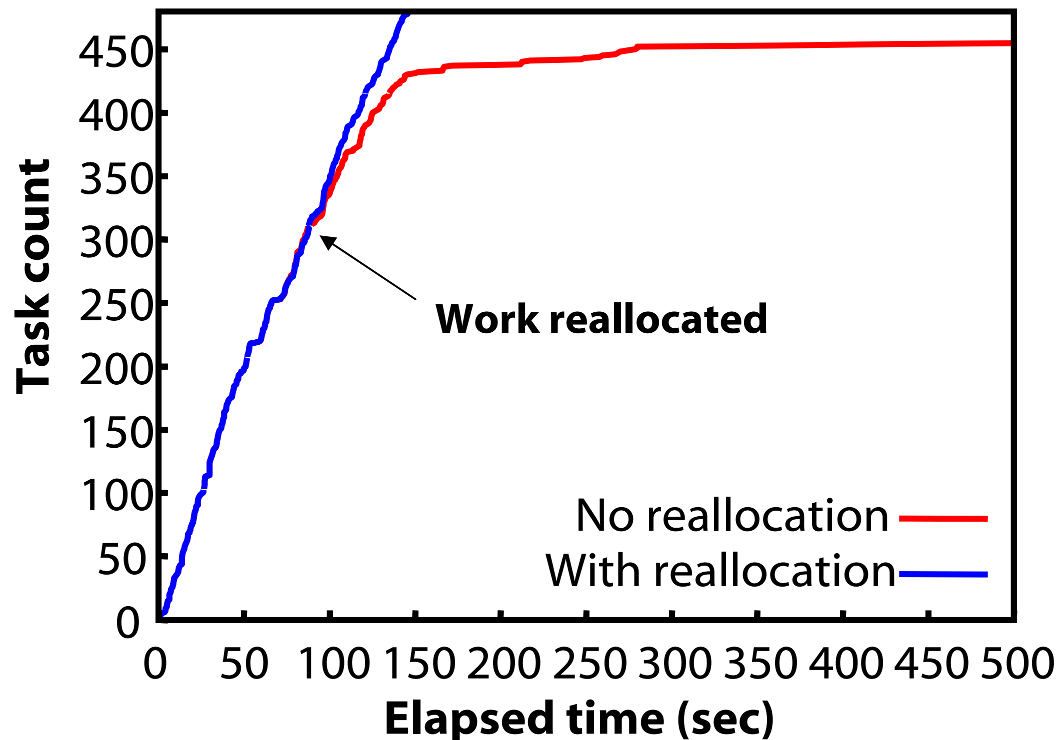
- Application-specific data processing
- 2500+ seconds to complete 480 map tasks on 30 hosts

Application Characteristics

- Bullet, EMAN, and MapReduce belong to a specific class of applications
  - Support mid-computation reconfiguration
  - Support dynamically degrading computation
- Some applications do not support reconfiguration
  - Degrading computation may reduce accuracy
  - Require specific number of hosts
- For applications that support reconfiguration, we improve performance by coping with stragglers

Dealing with Stragglers

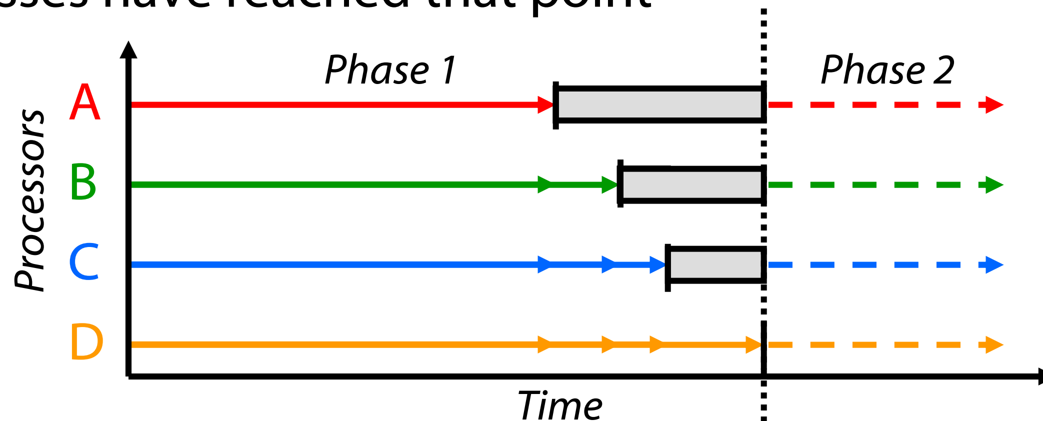
- MapReduce explicitly dealt with stragglers
  - Detected slow hosts and reallocated work to fast hosts



- Need a general technique for detecting stragglers
- Solution: Partial barriers

Synchronization Barriers

- Traditionally synchronization barriers have separated different phases of computation in multi-processor computing environments
  - Ensure no process advances beyond a specified point until all processes have reached that point

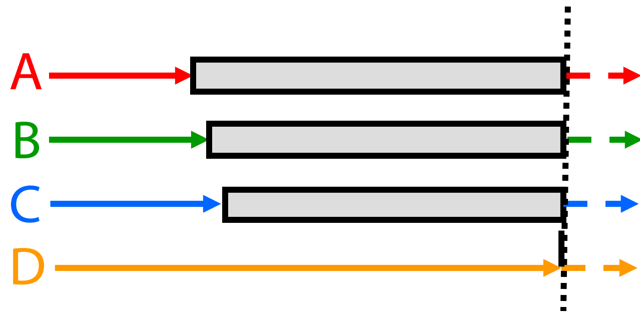


- Distributed applications also benefit from barrier semantics
  - Phased parallel computation
  - Coordinated measurement

Barrier Drawbacks

- Traditional semantics are too strict in failure-prone distributed computing environments
  - Machines fail and restart
  - Network links become congested
  - Hosts become overloaded
- Progress is limited by pace of slowest participant
- May wait indefinitely for failed hosts
- Partial barriers relax traditional barrier semantics to perform better in volatile distributed environments
  - Relaxed semantics are more robust to variable network conditions
  - Show improved performance in 4 applications

Partial Barrier Semantics



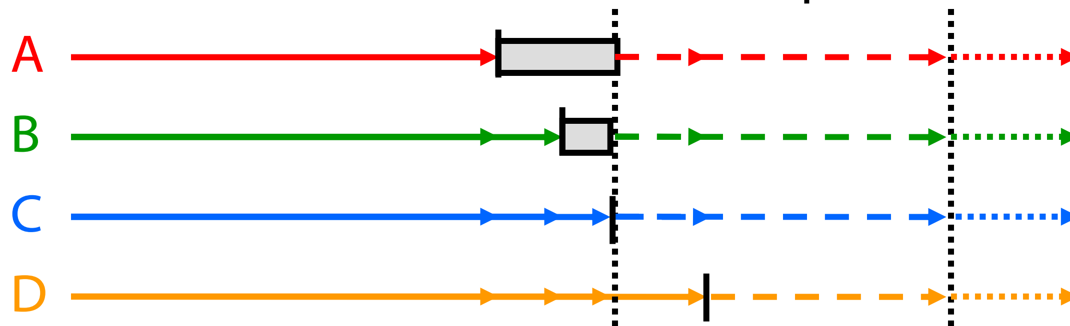
Traditional barrier

- Hosts wait at barrier until all hosts have entered
- All hosts released simultaneously

Problem: Stragglers delay overall progress

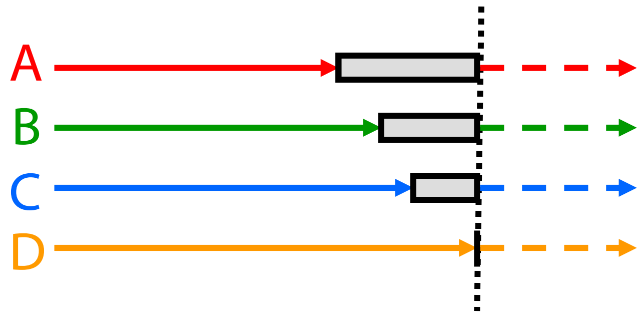
Solution: **Early Entry**

- Hosts released from barrier without waiting for all other hosts to enter
- Applications set barrier release thresholds (min percentage, timeout)
- Must deal with late arrivals (late-fire or catch-up)



Release threshold = 75%

Partial Barrier Semantics



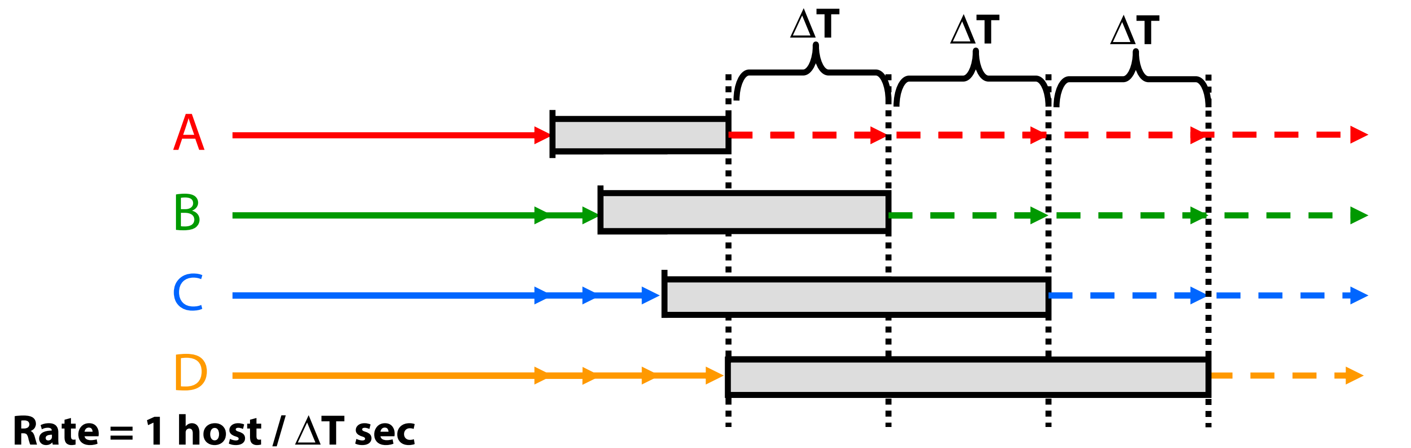
Traditional barrier

- Hosts wait at barrier until all hosts have entered
- All hosts released simultaneously

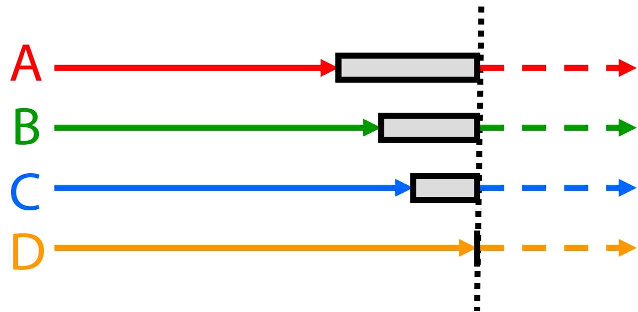
Problem: Simultaneously releasing all hosts causes overload

Solution: **Throttled Release**

- Hosts released from barrier at specified rate
- Application sets rate of release (# hosts / time interval)



Partial Barrier Semantics



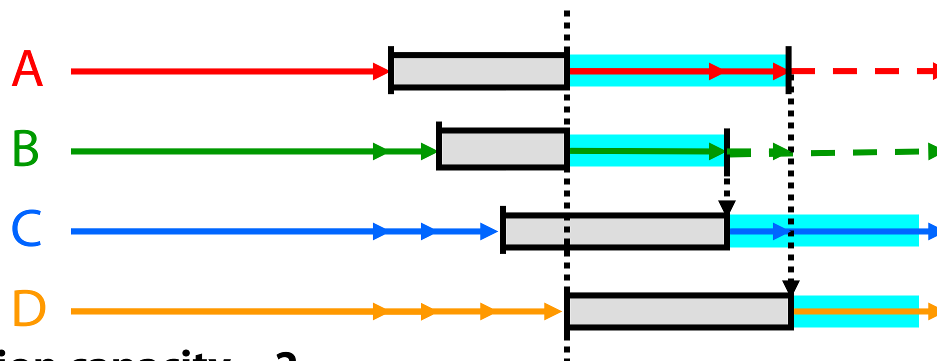
Traditional barrier

- Hosts wait at barrier until all hosts have entered
- All hosts released simultaneously

Problem: Limited simultaneous resource availability

Solution: **Semaphore Barrier**

- Control number of hosts allowed in critical section
- Application sets capacity of critical section



Critical section capacity = 2

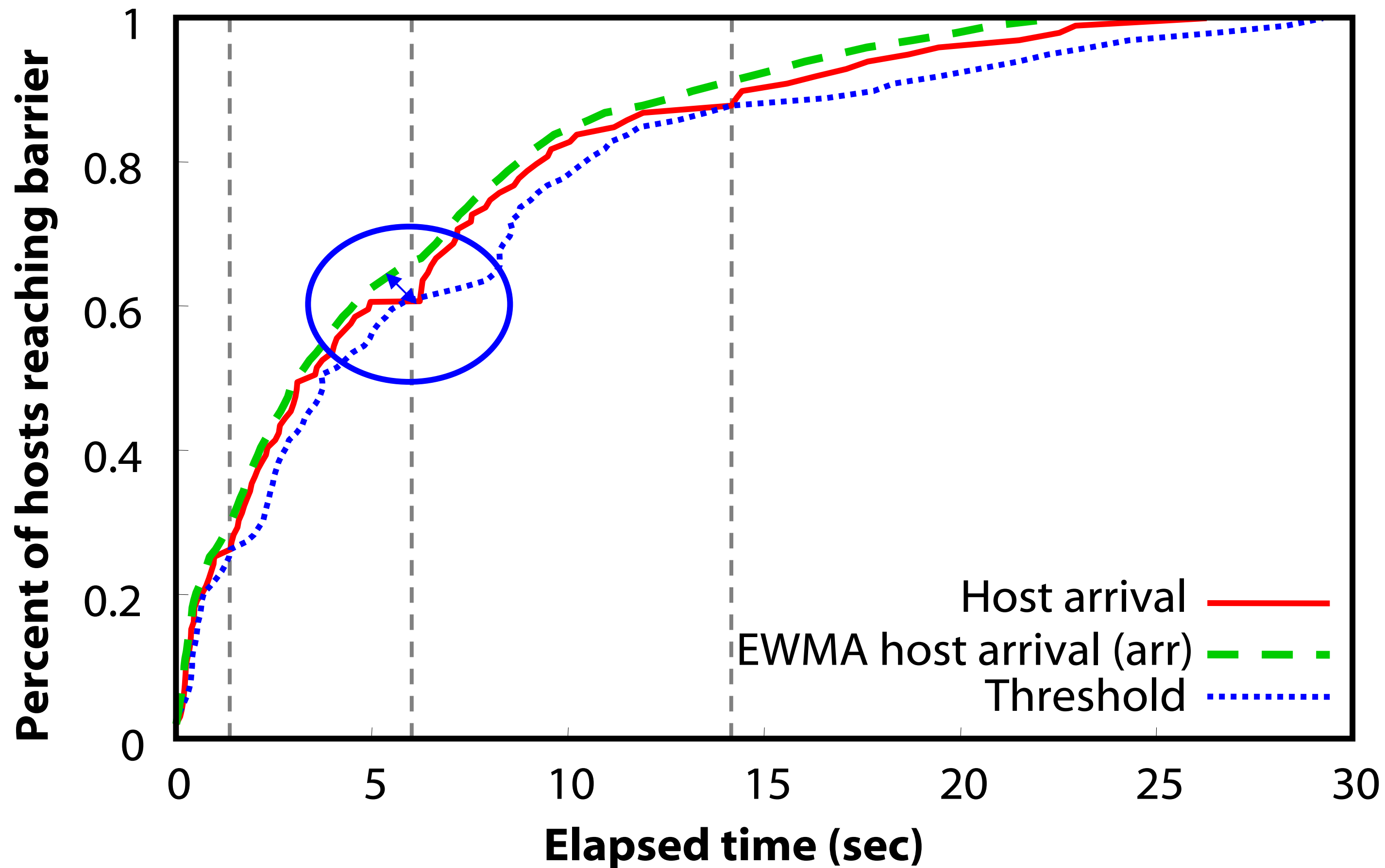
Adaptive Release

- Static thresholds for release are not sufficient
  - Adapt to changing conditions

- Early entry – dynamically detect stragglers by finding *knee* of completion curve

- Semaphore barrier – dynamically determine optimal capacity of critical section

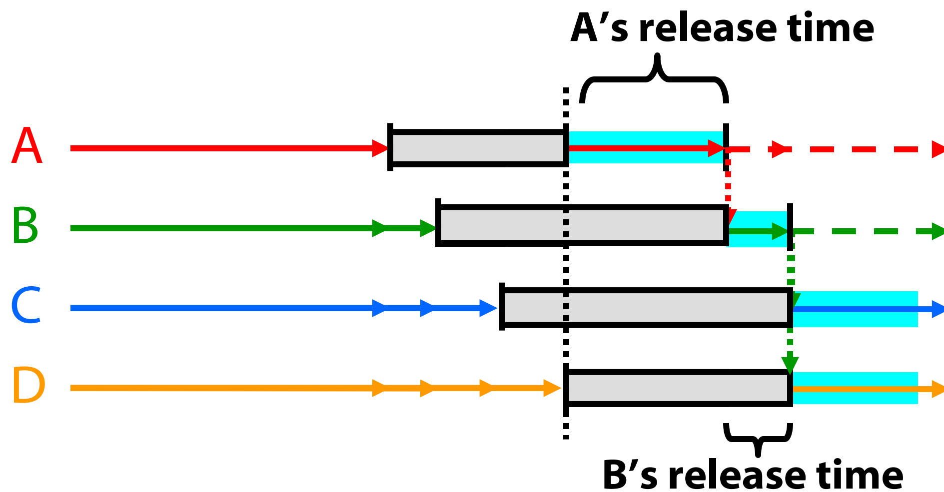
Detecting Knees

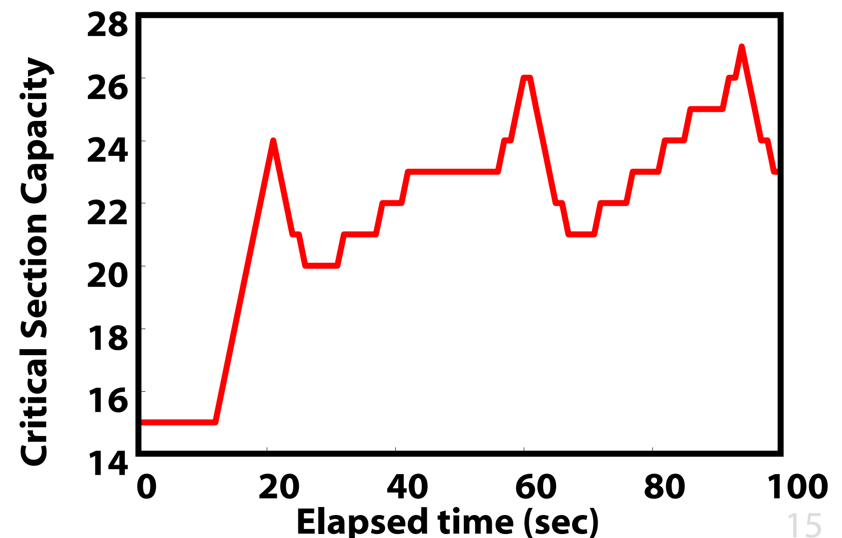


- Want to know when majority of hosts who will arrive quickly have arrived

Critical Section Capacity

- Determine optimal capacity of critical section
  - Dynamically adjust algorithm to find appropriate level of concurrency
- Start with low concurrency
  - Keep track of “release time” (time spent in critical section)
- Increase concurrency until conditions worsen
  - Most recent median release time > Overall median release time
- Back off and repeat





Critical section capacity = 2

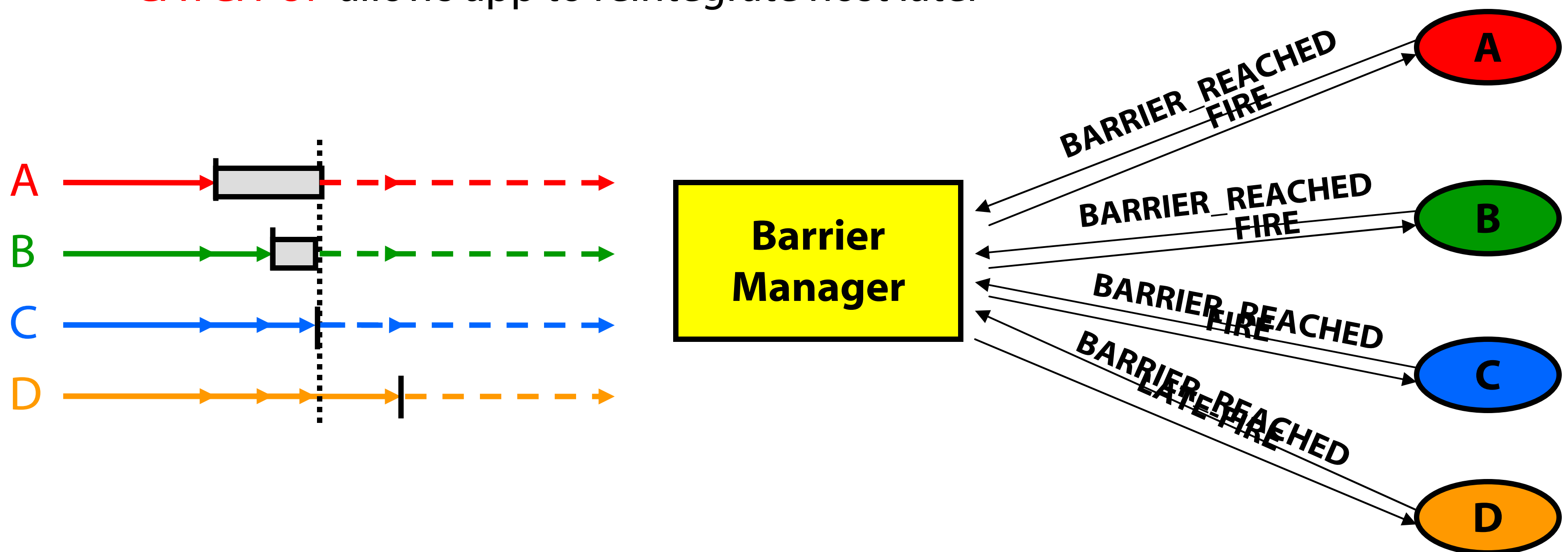
Application Integration

- Easy to integrate into existing applications
- Partial barrier participants use simple API to customize application behavior
- Participants specify barrier manager during initialization
- Manager coordinates communication across hosts

```
class Barrier {
    Barrier (string name, int max, int timeout, int percent, int minWait);
    static void setManager (string hostname);
    void enter (string label, string hostname);
    void setEnterCallback (
      bool (*callbackFunc) (string label, string hostname, bool default),
     int timeout);
    map <string label, string hostname> getHosts (void);
}
```

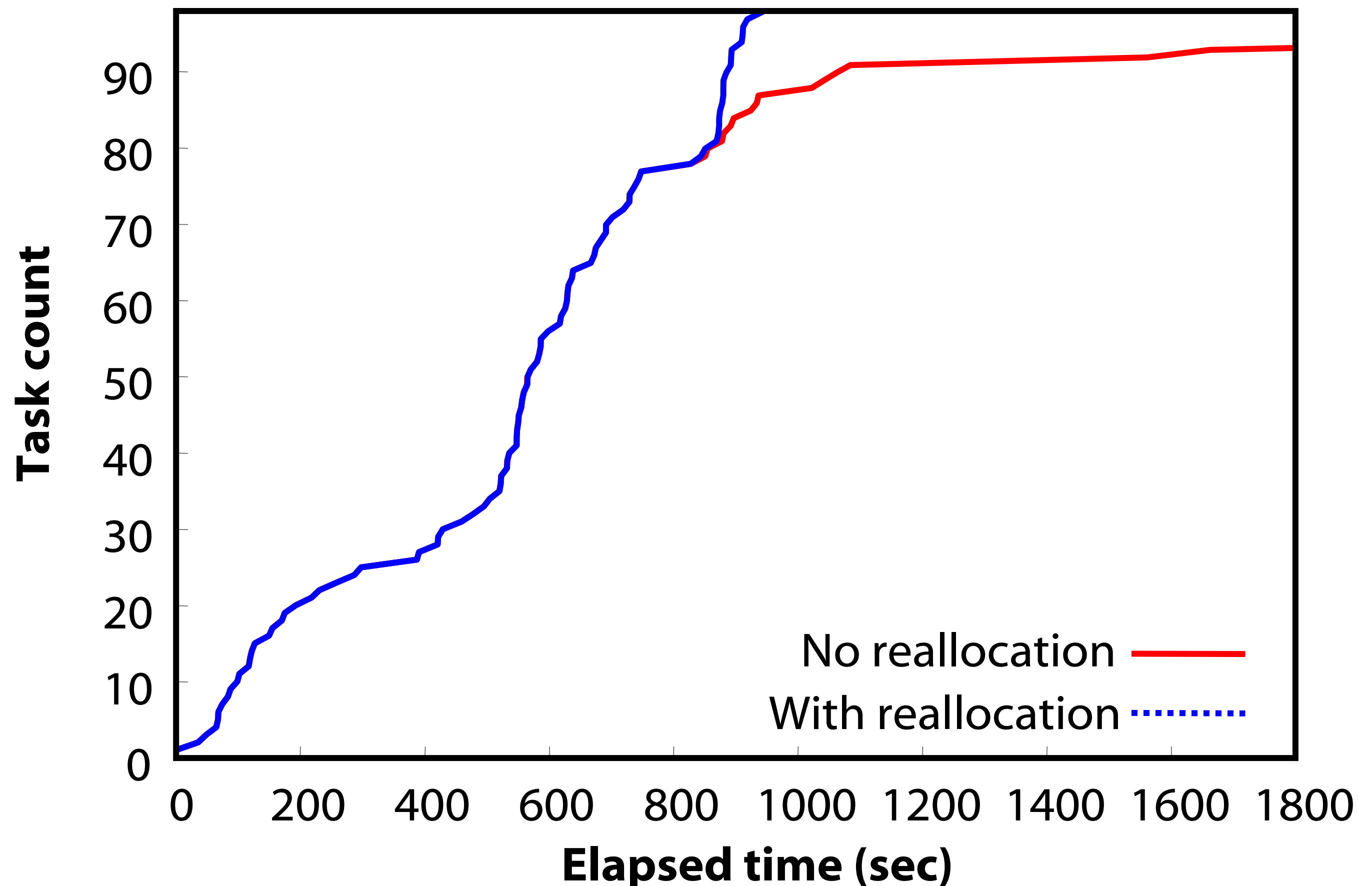
Implementation Details

- Barrier manager specified at startup by application
- Hosts enter barrier and send BARRIER_REACHED messages to manager
- Manager sends FIRE message when condition to release barrier is achieved
- Manager sends LATE-FIRE/CATCH-UP in response to late arrivals
  - LATE-FIRE allows execution to continue immediately
  - CATCH-UP allows app to reintegrate host later



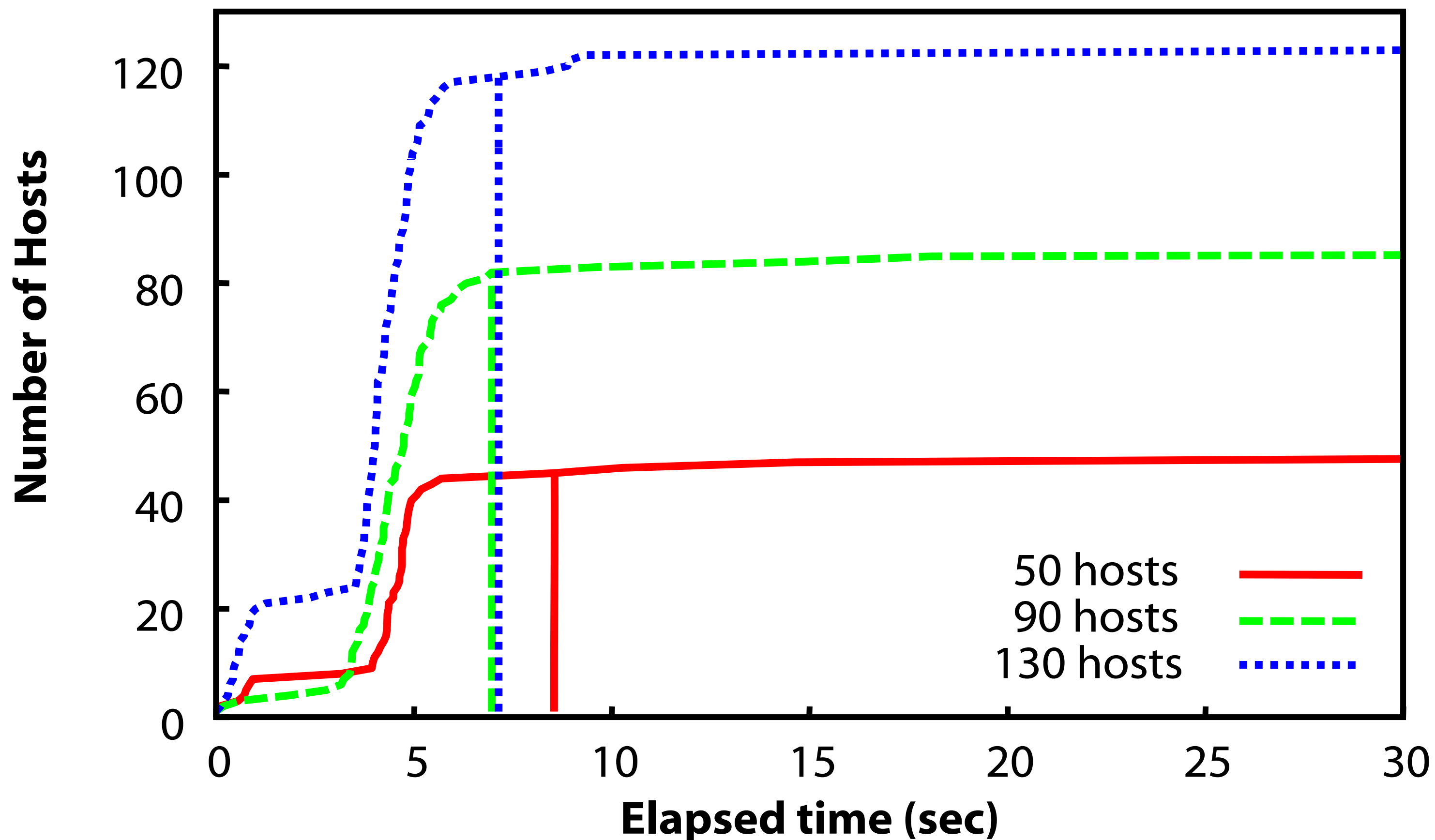
Work Reallocation in EMAN

- Images processed using sequential and parallel computations
- Barriers detect stragglers and reallocate work



Detecting Knees in Bullet

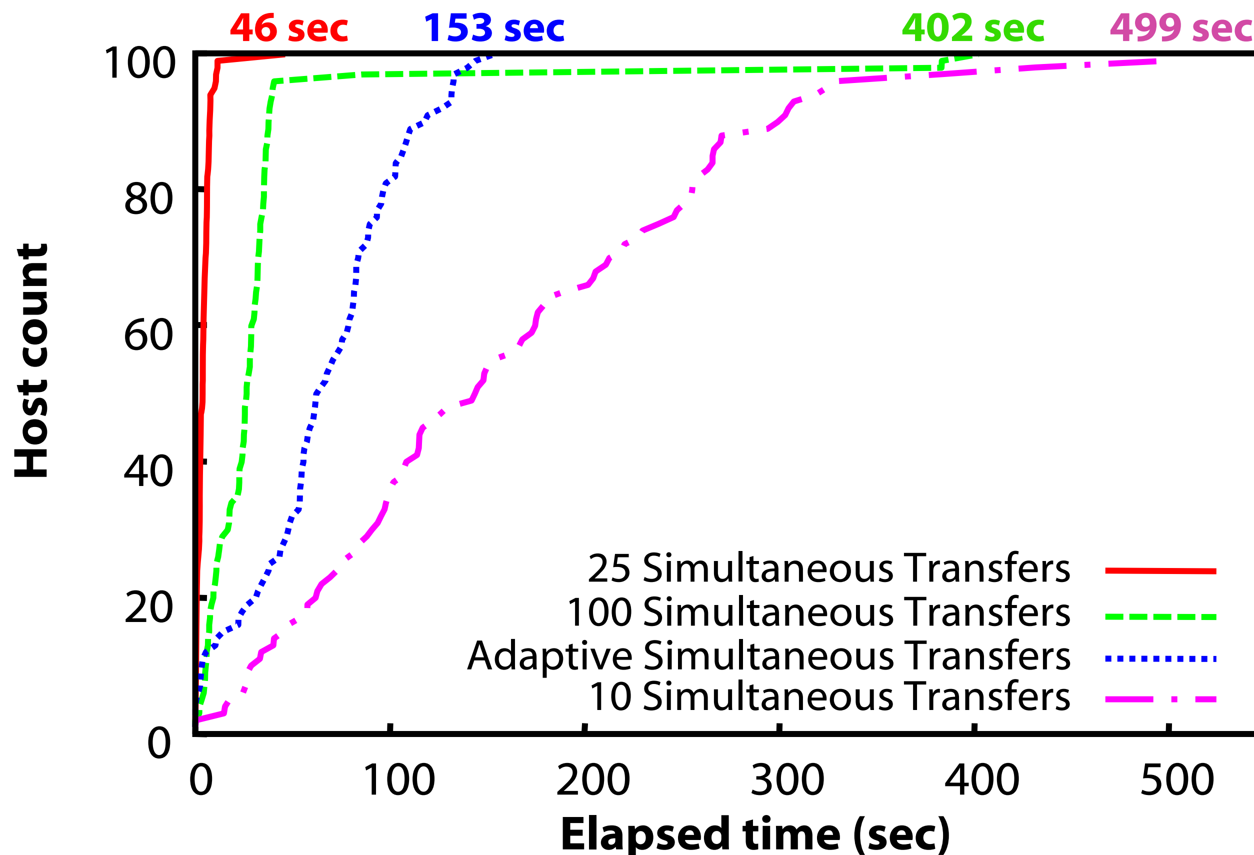
- Overlay based file distribution protocol
- Barrier used to determine when to start without waiting for stragglers



Admission Control in Plush

[SIGOPS-OSR 06]

- Application management infrastructure for deploying and maintaining distributed applications
- Barriers throttle the number of simultaneous file transfers



Related Work

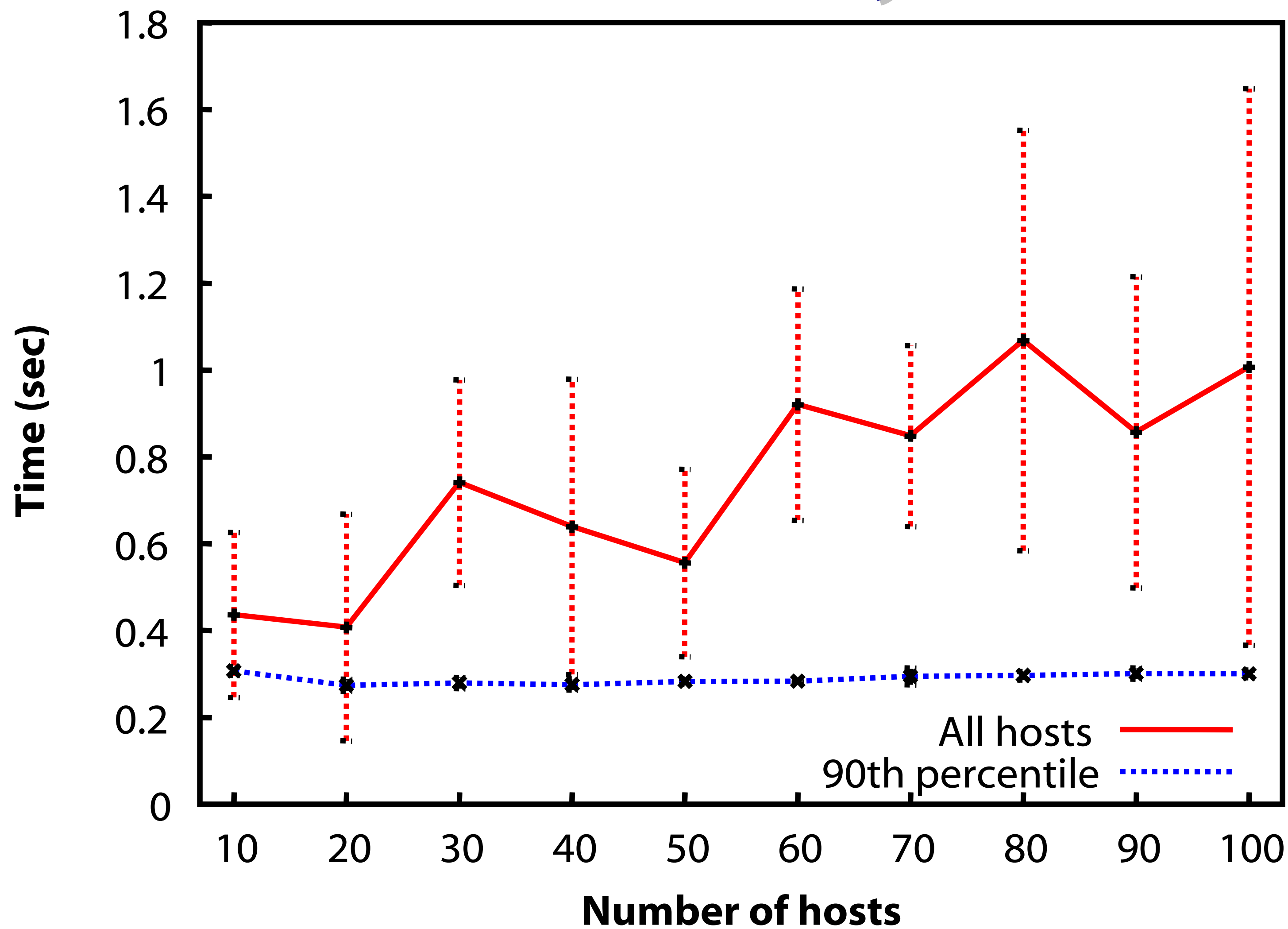
- Barriers first used for synchronization in parallel programming [Jordan78]
- Fuzzy barriers in SIMD programming allowed some instructions to be run while waiting in barrier [Gupta89]
- Knee detection in TCP retransmission timers and MONET [Andersen05]
- Detecting optimal level of concurrency in SEDA [Welsh01]
- Use barrier arrival rate for scheduling and load balancing in Implicit Coscheduling [Dusseau96]
- Reallocate work in work-stealing schedulers like CILK [Blumofe95]

Summary

- Partial barriers are a useful relaxation of the traditional barrier synchronization primitive
  - New semantics: early entry, throttled release, semaphore barrier
  - Designed for improved performance in volatile and failure-prone environments
- Easy integration and decreased completion times in existing distributed applications
  - Plush, Bullet, EMAN, MapReduce
- Adaptive release is more effective than static thresholds in distributed environments
  - Ensures forward progress in unpredictable conditions

Questions?

Scalability



- Time to move between two barriers separated by no-op

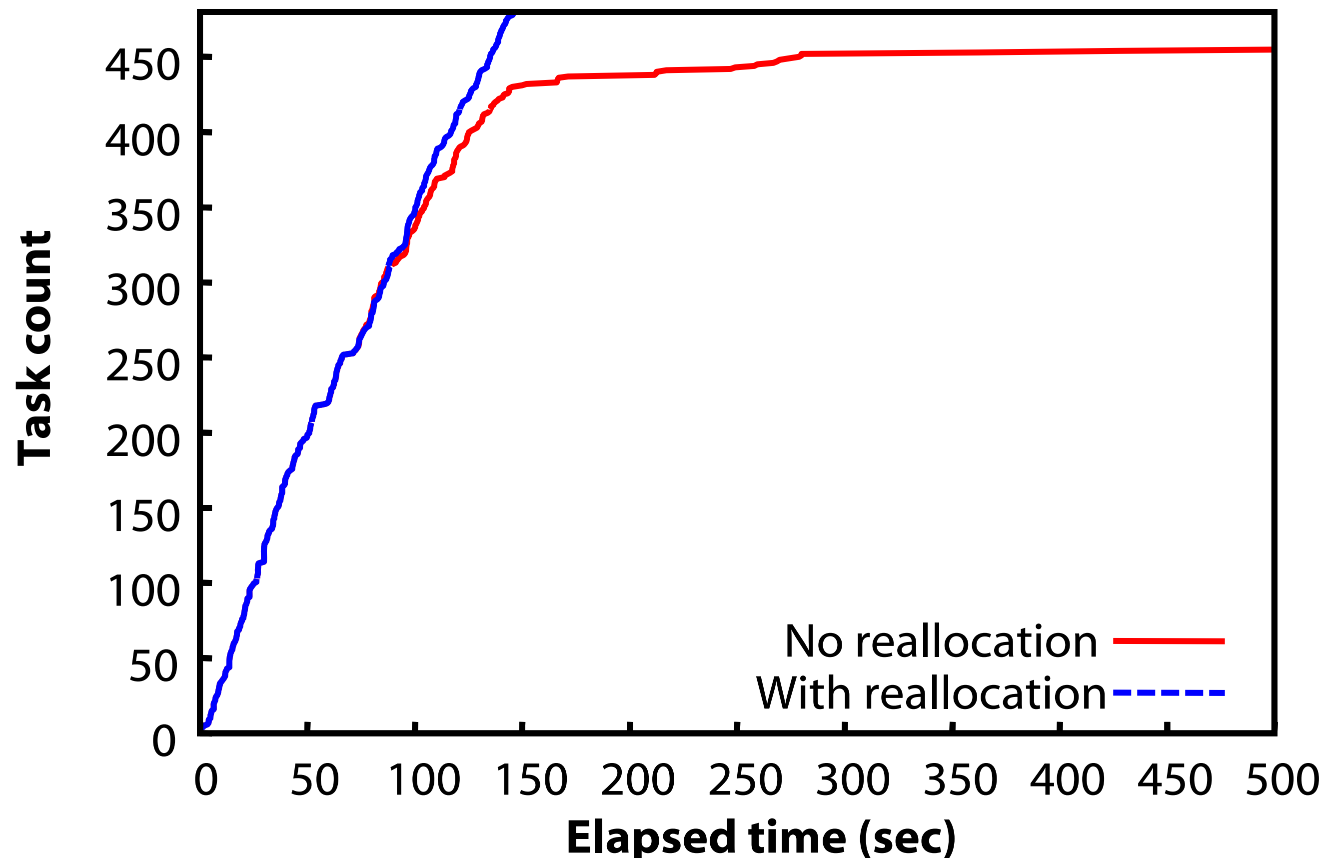
Partial Barrier API

```
class Barrier {
    Barrier (string name, int max, int timeout, int percent, int minWait);
    static void setManager (string hostname);
    void enter (string label, string hostname);
    void setEnterCallback (
      bool (*callbackFunc) (string label, string hostname, bool default),
      int timeout);
    map <string label, string hostname> getHosts (void);
}

class ThrottleBarrier extends Barrier {
    void setThrottleReleasePercent (int percent);
    void setThrottleReleaseCount (int count);
    void setThrottleReleaseTimeout (int timeout);
}
```

Load Balancing in MapReduce

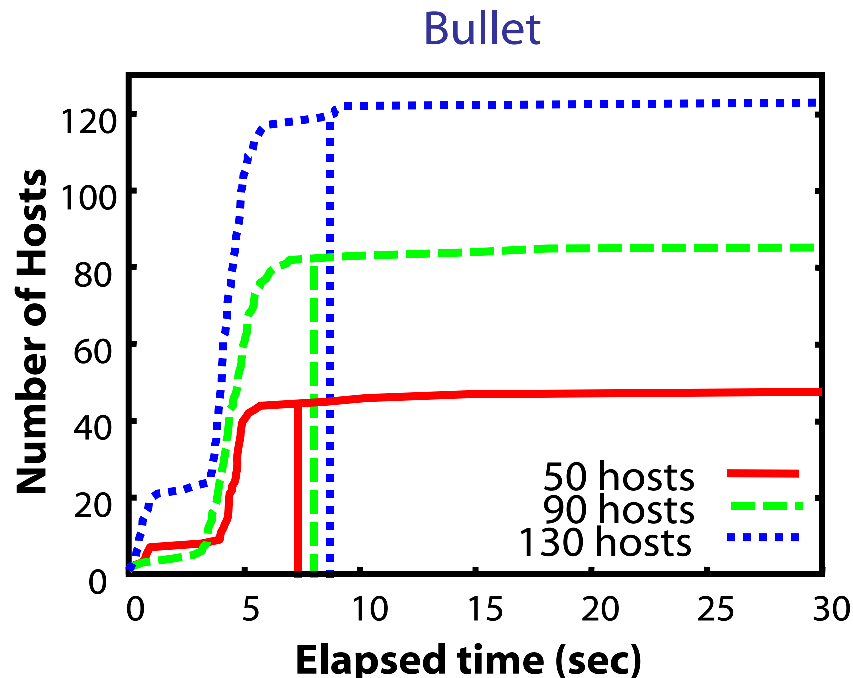
- Reimplementation of toolkit for application specific data summarizing and processing
- Barrier detects stragglers and rebalances work in map phase



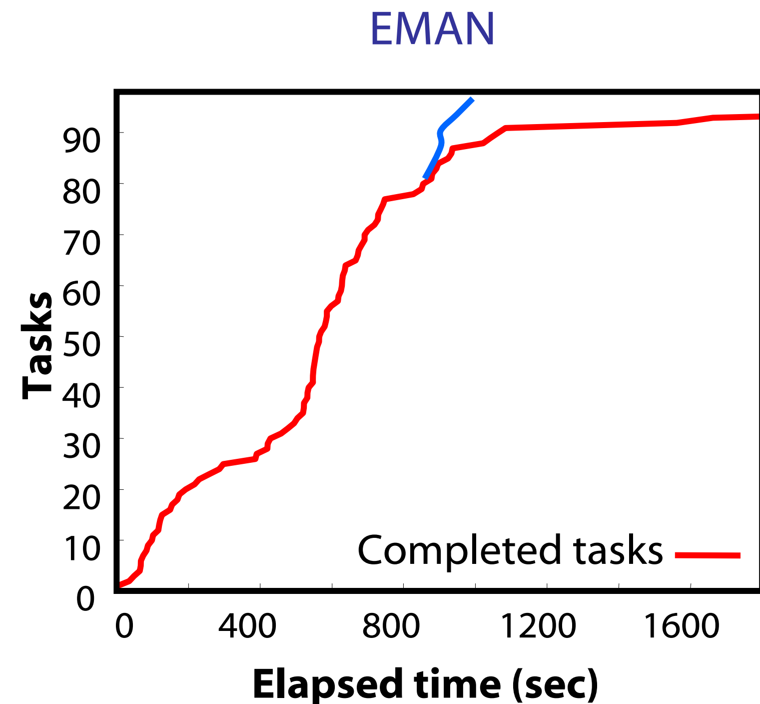
Dealing with Stragglers

- Need a general technique for detecting stragglers in distributed applications
  - Ease developers of burden of handling stragglers separately for each application
  - Detect "knee" of curve and adjust application

Bullet



EMAN



Revisiting EMAN

- Suppose early entry threshold = 60%
- Barrier is released too early
- How do we determine optimal threshold value?