Simple Locality-Aware Co-Allocation in Peer-to-Peer Supercomputing

Niels Drost
Rob van Nieuwpoort
Henri Bal
Goal of Research

Create a middleware system capable of running distributed supercomputing applications.
Current Solutions

- Difficult to setup and maintain (Globus)
- Centralized components (Koala, Xtremweb)
- Co-allocation usually not available (Boinc)
P2P Solution

- Advantages
  - Little or no maintenance
  - Fault-tolerant
  - Scalable

- Disadvantages
  - Co-ordination
  - Security and Trust
Outline

✔ Introduction

➔ Zorilla
  • Flood Scheduling
  • Implementation
  • Experiments (on > 800 Grid5000 cpus)
  • Conclusions, Future work
Zorilla

- Prototype Java Peer-to-Peer supercomputing middleware system
- Fully Distributed
- P2P network: Bamboo
  - Structured overlay (Pastry like)
  - Locality aware
Running an Application (Current)

- **Deployment**
  - Copy program and input files to all sites
  - Determine local job scheduling mechanism
  - Write job submission scripts
  - Determine network setup of all clusters

- **Running**
  - Determine site and node availability
  - Submit application to the scheduler on each site
  - Monitor progress of application

- **Clean up**
  - Gather output and log files
  - Cancel remaining reservations
  - Remove program, input, output and log files from sites
Running an Application (Zorilla)

$ submit -i nqueens.jar -#w 676 NQueens 1 22 5
Life of a job in Zorilla (1/4)
Life of a job in Zorilla (1/4)
Life of a job in Zorilla (2/4)
Life of a job in Zorilla (3/4)

\[ W = \text{worker running application} \]
Life of a job in Zorilla (4/4)
Simple Locality-Aware Co-Allocation in Peer-to-Peer Supercomputing

✓ Introduction
✓ Zorilla
➔ Flood Scheduling
• Implementation
• Experiments (on > 800 Grid5000 cpus)
• Conclusions, Future work
Scheduler Requirements

- Co-allocation
- Locality Aware
- Fault-tolerant
- Flexible

“Magic” Scheduler
Flood Scheduling

- Nodes flood advertisements for jobs
- Radius (TTL) limits diameter of flood
- Nodes decide locally if they join computation
Flood Example

Radius = 2
Scheduling algorithm

int radius = 1;
int time = 1; //seconds
while (!enough_workers()) {
    flood_job_advertisement(radius);
    add_new_workers_to_computation();
    wait(time);
    radius++;
    time = time * 2;
}

Scheduler Requirements (Revisited)

- Co-allocation
- Locality Aware
- Fault-tolerant
- Flexible
Implementation
Experiment

- Grid5000, Six clusters, 430 nodes, 860 processors
- Submit jobs from single node (at Sophia)

while(more_nodes_available) {
    submit_new_job();
}
Rennes (18 ms, 118 workers)
Orsay (17 ms, 356 workers)
Bordeaux (10 ms, 82 workers)
Toulouse (8 ms, 110 workers)
Grenoble (5 ms, 18 workers)
Sophia–Antipolis (<1 ms, 176 workers)
Conclusions

- Flood scheduling is able to efficiently schedule resources to jobs in a grid environment.
- A better P2P network is needed to further optimize scheduling.
- P2P middleware is a promising alternative to current grid systems.
Current/Future Work

- Redesign of P2P Network
  - Better locality awareness
  - Support of more metrics (bandwidth, trust, reliability, etc)
- Authentication, Authorization, (Accounting), based on PGP
- Fair scheduling
- Explicit support for workflow and data-intensive applications
www.cs.vu.nl/~ndrost/zorilla
ndrost@cs.vu.nl
What about starvation?

• Jobs “split” resources
  – Applications started when a single resource is available, expand when more found
  – Only works with malleable applications
  – Example: Satin applications

• Single job wins, other jobs fail
  – Needs scheduling support
Scheduling algorithm (Thread 1)

```java
int radius = 1;
int time = 1; // seconds
while (!started) {
    flood_job_advertisement(radius);
    radius++;
    time = time * 2;
    wait(time);
}
```
Scheduling algorithm (Thread 2)

while(!started) {
    update_available_workers_list();
    if(enough_workers_available()) {
        try {
            claim_available_nodes();
            start_workers();
            started = true;
        }
    }
    wait(A_WHILE);
}