Fault tolerance, malleability and migration for divide-and-conquer applications on the Grid

Gosia Wrzesińska, Rob V. van Nieuwpoort, Jason Maassen, Henri E. Bal
Distributed supercomputing

- Parallel processing on geographically distributed computing systems (grids)
- Needed:
  - Fault-tolerance: survive node crashes
  - Malleability: add or remove machines at runtime
  - Migration: move a running application to another set of machines
- We focus on divide-and-conquer applications
Outline

• The Ibis grid programming environment
• Satin: a divide-and-conquer framework
• Fault-tolerance, malleability and migration in Satin
• Performance evaluation
The Ibis system

• Java-centric => portability
  – „write once, run anywhere”

• Efficient communication
  – Efficient pure Java implementation
  – Optimized solutions for special cases

• High level programming models:
  – Divide & Conquer (Satin)
  – Remote Method Invocation (RMI)
  – Replicated Method Invocation (RepMI)
  – Group Method Invocation (GMI)

http://www.cs.vu.nl/ibis/
Satin: divide-and-conquer on the Grid

• Performs excellent on the Grid
  – Hierarchical: fits hierarchical platforms
  – Java-based: can run on heterogeneous resources
  – Grid-friendly load balancing: Cluster-aware Random Stealing [van Nieuwpoort et al., PPoPP 2001]

• Missing support for
  – Fault tolerance
  – Malleability
  – Migration
Example application: Fibonacci

Also: Barnes-Hut, Raytracer, SAT solver, Tsp, Knapsack...
Fault-tolerance, malleability, migration

• Can be implemented by handling processors joining or leaving the ongoing computation
• Processors may leave either unexpectedly (crash) or gracefully
• Handling joining processors is trivial:
  – Let them start stealing jobs
• Handling leaving processors is harder:
  – Recompute missing jobs
  – Problems: orphan jobs, partial results from gracefully leaving processors
Crashing processors

processor 1

processor 2

processor 3
Crashing processors
Crashing processors

processor 3

processor 1
Crashing processors

Problem: orphan jobs
– jobs stolen from crashed processors
Problem: orphan jobs
  – jobs stolen from crashed processors
Handling orphan jobs

- For each *finished* orphan, broadcast (jobID, processorID) tuple; abort the rest
- All processors store tuples in *orphan tables*
- Processors perform lookups in orphan tables for each *recomputed* job
- If successful: send a *result request* to the owner (async), put the job on a stolen jobs list

processor 3

```
broadcast (9,cpu3)(15,cpu3)
```
Handling orphan jobs - example
Handling orphan jobs - example

处理器1

处理器3

14 15

12 13
Handling orphan jobs - example
Handling orphan jobs - example

(9, cpu3)
(15,cpu3)
Handling orphan jobs - example

1  2  3

4  5  6  7

9  12  13

15

processor 1

processor 3

<table>
<thead>
<tr>
<th></th>
<th>cpu3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cpu3</td>
</tr>
<tr>
<td>15</td>
<td>cpu3</td>
</tr>
</tbody>
</table>
Handling orphan jobs - example
Processors leaving gracefully

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

processor 1

processor 2

processor 3
Processors leaving gracefully

Send results to another processor; treat those results as orphans
Processors leaving gracefully
Processors leaving gracefully

(processor 1)

11,cpu3)
9,cpu3)
15,cpu3)
Processors leaving gracefully
Processors leaving gracefully
Some remarks about scalability

• Little data is broadcast (< 1% jobs)
• We broadcast pointers
• Message combining
• Lightweight broadcast: no need for reliability, synchronization, etc.
Performance evaluation

- Leiden, Delft (DAS-2) + Berlin, Brno (GridLab)
- Bandwidth:
  62 – 654 Mbit/s
- Latency:
  2 – 21 ms
Impact of saving partial results

- 1 cluster leaves unexpectedly (without saving orphans)
- 1 cluster leaves unexpectedly (with saving orphans)
- 1 cluster leaves gracefully
- 1.5/3.5 clusters (no crashes)
Migration overhead

8 cpus Leiden
4 cpus Berlin
4 cpus Brno
(Leiden cpus replaced by Delft)
Crash-free execution overhead

Used: 32 cpus in Delft

![Graph showing the speedup on 32 cpus for Raytracer, TSP, SAT solver, and Knapsack. The graph compares plain Satin and malleable Satin.]
Summary

- Satin implements fault-tolerance, malleability and migration for divide-and-conquer applications
- Save partial results by repairing the execution tree
- Applications can adapt to changing numbers of cpus and migrate without loss of work (overhead < 10%)
- Outperform traditional approach by 25%
- No overhead during crash-free execution
Further information

Publications and a software distribution available at:

http://www.cs.vu.nl/ibis/
Additional slides
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
Partial results on leaving cpus

If processors leave **gracefully**:

- Send all **finished** jobs to another processor
- Treat those jobs as orphans = broadcast (jobID, processorID) tuples
- Execute the normal crash recovery procedure
A crash of the master

- Master: the processor that started the computation by spawning the root job
- Remaining processors elect a new master
- At the end of the crash recovery procedure the new master restarts the application
Job identifiers

• rootId = 1
• childId = parentId * branching_factor + child_no
• Problem: need to know maximal branching factor of the tree
• Solution: strings of bytes, one byte per tree level
Distributed ASCI Supercomputer (DAS) – 2

Node configuration
Dual 1 GHz Pentium-III
>= 1 GB memory
100 Mbit Ethernet + (Myrinet)
Linux

VU (72 nodes)
UvA (32)
Leiden (32)
Delft (32)
Utrecht (32)

GigaPort (1-10 Gb)
Compiling/optimizing programs

- Optimizations are done by bytecode rewriting
  - E.g. compiler-generated serialization (as in Manta)
interface FibInter
    extends ibis.satin.Spawnable {
    public int fib(long 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;
    }
}

Java + divide&conquer

GridLab testbed
Grid results

<table>
<thead>
<tr>
<th>Program</th>
<th>sites</th>
<th>CPUs</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Raytracer</td>
<td>5</td>
<td>40</td>
<td>81 %</td>
</tr>
<tr>
<td>SAT-solver</td>
<td>5</td>
<td>28</td>
<td>88 %</td>
</tr>
<tr>
<td>Compression</td>
<td>3</td>
<td>22</td>
<td>67 %</td>
</tr>
</tbody>
</table>

- Efficiency based on normalization to single CPU type (1GHz P3)