Newsflash from Melmac:

MPI sucks!
Programming Models

Computer scientists:

- Dedicate their lives to them
- Get Ph.D.'s for them
- Love them

Application programmers:

- Want to get their work done
- Choose the smallest evil
Programming Models (2)

Single computer (a.k.a. sequential)

- Object-oriented or components
  - High programmer productivity through high abstraction level

Parallel computer (a.k.a. cluster)

- Message passing
  - High performance through good match with machine architecture
Programming Models (3)

Grids (a.k.a. Melmac)

- ???

  - Fault-tolerance
  - Security
  - Platform independence
  - ...

Applications' View: Functional Properties

What applications need to do:

- Access to compute resources, job spawning and scheduling
- Access to file and data resources
- Communication between parallel and distributed processes
- Application monitoring and steering
Applications' View: Non-functional Properties

What else needs to be taken care of:

- Performance
- Fault tolerance
- Security and trust
- Platform independence
Middleware's View: (from: Foster et al., “Anatomy of the Grid”)

OGSA: execution, data, res.mgmt., security, info., self mgmt., MPI...

Monitoring of + information about resources (resource access control)

Network conn., authentication

“The hardware”
Features: Application vs. Middleware

<table>
<thead>
<tr>
<th><strong>Application View</strong></th>
<th><strong>Feature</strong></th>
<th><strong>Middleware View</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Monitoring/Info</td>
<td>Resources</td>
</tr>
<tr>
<td>Non-Functional</td>
<td>Resource Access</td>
<td>Functional</td>
</tr>
<tr>
<td>Non-Functional</td>
<td>Security</td>
<td>Functional</td>
</tr>
<tr>
<td>Non-Functional</td>
<td>Connectivity</td>
<td>Functional</td>
</tr>
<tr>
<td>Functional</td>
<td>Data</td>
<td>Functional</td>
</tr>
<tr>
<td>Functional</td>
<td>Compute Nodes</td>
<td>Functional</td>
</tr>
</tbody>
</table>
### Levels of Virtualization

<table>
<thead>
<tr>
<th>Collective layer</th>
<th>Service APIs</th>
<th>Individual resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource layer</td>
<td>Resource API (GRAM?)</td>
<td>resource/local scheduler</td>
</tr>
<tr>
<td>Connectivity layer</td>
<td>IP</td>
<td>Network links</td>
</tr>
<tr>
<td>Cluster OS</td>
<td>Management API</td>
<td>Compute nodes</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Language</td>
<td>OS(?)</td>
</tr>
<tr>
<td>Virtual OS</td>
<td>System calls</td>
<td>OS</td>
</tr>
<tr>
<td>OS</td>
<td>System calls</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

Each virtualization brings a trade-off between abstraction and control.
Translating to API's

<table>
<thead>
<tr>
<th>grid-unaware application</th>
<th>grid-aware application</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid-enabled programming environments</td>
<td>simplified API</td>
</tr>
<tr>
<td>application support tools</td>
<td></td>
</tr>
<tr>
<td>service and resource abstraction layer</td>
<td></td>
</tr>
</tbody>
</table>

Application + runtime env.

Middleware

Resources
Grid Application Runtime Stack

“just want to run fast”

“want to handle remote data/machines”

MPICH-G
Workflow
Satin/Ibis
NetSolve
...

Grid Application Toolkit (GAT)

Added value for applications

SAGA
### Your API depends on what you want to do

<table>
<thead>
<tr>
<th>Legacy apps</th>
<th>Sand boxing (VM's?)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel apps</td>
<td>Grid-enabled environment</td>
</tr>
<tr>
<td>Grid-aware codes</td>
<td>Simplified API (SAGA)</td>
</tr>
<tr>
<td>Support tools</td>
<td>resource/service abstraction (GAT)</td>
</tr>
<tr>
<td>Services/resource management</td>
<td>Service API's (&quot;bells and WSDL's&quot;)</td>
</tr>
</tbody>
</table>
A Case Study in Grid Programming

- Grids @ Work, Sophia-Antipolis, France, October 2005
- VU Amsterdam team participating in the N-Queens contest
- Aim: running on a 1000 distributed nodes
The N-Queens Contest

- Challenge: solve the most board solutions within 1 hour
- Testbed:
  - Grid5000, DAS-2, some smaller clusters
  - Globus, NorduGrid, LCG, ???
  - In fact, there was not too much precise information available in advance...
Computing in an Unknown Grid?

• Heterogeneous machines (architectures, compilers, etc.)
  – Use Java: “write once, run anywhere”
    Use Ibis!

• Heterogeneous machines (fast / slow, small / big clusters)
  – Use automatic load balancing (divide-and-conquer)
    Use Satin!

• Heterogeneous middleware (job submission interfaces, etc.)
  – Use the Grid Application Toolkit (GAT)!
Assembling the Pieces

- N-Queens
- Satin/Ibis
- Deployment application

Diagram:

- Grid-unaware application
- Grid-enabled programming environments
- Application support tools
- Service and resource abstraction layer

Java GAT on top of ProActive and ssh
The Ibis Grid Programming System

Application

- Satin
- RMI
- GMI
- MPJ
- RepMi

Ibis Portability Layer (IPL)

- TCP
- UDP
- P2P
- GM
- MPI
- Panda

Pure Java

Native Code

Under Development
Satin: Divide-and-conquer

- Effective paradigm for Grid applications (hierarchical)
- Satin: Grid-aware load balancing (work stealing)
- Also support for
  - Fault tolerance
  - Malleability
  - Migration
Satin Example: Fibonacci

class Fib {
    int fib (int n) {
        if (n < 2) return n;
        int x = fib(n-1);
        int y = fib(n-2);
        return x + y;
    }
}

Single-threaded Java
Satin Example: Fibonacci

```java
public interface FibInter extends ibis.satin.Spawnable {
    public int fib (int n);
}

class Fib extends ibis.satin.SatinObject implements FibInter {
    public int fib (int n) {
        if (n < 2) return n;
        int x = fib(n-1); /*spawned*/
        int y = fib(n-2); /*spawned*/
        sync();
        return x + y;
    }
}

(use byte code rewriting to generate parallel code)
```
Satin: Fault-Tolerance, Malleability, Migration

Satin: referential transparency (jobs can be recomputed)

- Goal: maximize re-use of completed, partial results
- Main problem: orphan jobs (stolen from crashed nodes)
- Approach: fix the job tree once fault is detected
Recovery after Processor has left/crashed

- Jobs stolen by crashed processor are re-inserted in the work queue where they were stolen, marked as re-started

- Orphan jobs:
  - Abort running and queued sub jobs
  - For each complete sub job, broadcast (node id, job id) to all other nodes, building an orphan table (background broadcast)

- For Re-started jobs (and its children) check orphan table
One Mechanism Does It All

• If nodes want to leave gracefully:
  – Choose a random peer and send to it all completed, partial results
  – This peer then treats them like orphans
    • Broadcast (job id, own node id) for all “orphans”
• Adding nodes is trivial: let them start stealing jobs
• Migration: graceful leaving and addition at the same time
Summary: Ibis

• Java: “write once, run anywhere”
  – machine virtualization

• Ibis: efficient communication
  – network virtualization

• Satin: load balancing, fault-tolerance, migration
  – resource virtualization

But how do we deploy our Ibis / Satin application?

A (non-) functional problem to be solved
The Grid Application Toolkit (GAT)

Application Layer
- GAT API
- GAT Engine
- GAT Adaptors

User Space
- GridLab Services
- EGEE Services
- Unicore
- Globus-2.4
- Globus-3.2
- SSH/SSL

Service Layer

Capability Space

Core Layer
The Grid Application Toolkit (GAT)

- Simple and uniform API to various Grid middleware:
  - Globus 2, 3, 4, ssh, Unicore, ...

- Job submission, remote file access, job monitoring and steering

- Implementations:
  - C, with wrappers for C++ and Python
  - Java
Results achieved on the Grid5000 Testbed

<table>
<thead>
<tr>
<th>Site</th>
<th>CPUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orsay</td>
<td>426</td>
</tr>
<tr>
<td>Bordeaux</td>
<td>92</td>
</tr>
<tr>
<td>Rennes, Opteron cluster</td>
<td>120</td>
</tr>
<tr>
<td>Rennes, Xeon cluster</td>
<td>128</td>
</tr>
<tr>
<td>Sophia Antipolis</td>
<td>196</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>960</strong></td>
</tr>
</tbody>
</table>

- Solved $n=22$ in 25 minutes
- 4.7 million jobs, 800,000 load balancing messages
Pondering about Grid API's (a.k.a. Conclusions)

- Grid applications have many problems to address
- Different problems require different API's
- It's all about virtualization (on all levels)
- Can we find the “MPI equivalent” for the grid? Should we?
- Grids are considered successful as soon as they become invisible/ubiquitous.
- Are we done once everything is nicely virtualized “away”? 
- Should everything just be a Web service? (maybe not)
Acknowledgements

- **Ibis:**
  Jason Maassen, Rob van Nieuwpoort, Ceriel Jacobs, Rutger Hofman, Gosia Wrzesinska, Niels Drost, Olivier Aumage, Alexandre Denis, Fabrice Huet, Henri Bal, the Dutch VL-e project

- **GAT:**
  Andre Merzky, Rob van Nieuwpoort, the EU GridLab project

- **N-Queens:**
  Ana-Maria Oprescu, Andrei Agapi, the EU CoreGRID NoE

- **SAGA:**
  Andre Merzky, Shantenu Jha, Pascal Kleijer, Hartmut Kaiser, Stephan Hirmer, the OGF SAGA-RG, the EU XtreemOS project