Phoenix: a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources

Kenjiro Taura (University of Tokyo/JST)
Kenji Kaneda (University of Tokyo)
Toshio Endo (JST)
Akinori Yonezawa (University of Tokyo)
Demo

- Fluid dynamics
- Divide and conquer (recursive binary tree creation)
What’s Phoenix?

- (like MPI): A message passing *library* that allows nodes to join and leave at runtime
- (but not MPI): A slightly different programming *model* for expressing parallel algorithms that acquire/release nodes at runtime
Goal

- **Resource-reconfigurability**: ability to continue a single application in the face of resource availability changes
- **Goal**: support resource-reconfigurability with generality and scalability
Why Resource Reconfigurability?

- Continue parallel applications in the face of permanent failures (with smaller resources)
- Migrate parallel services without stopping operation
- Grab idle resources and release busy ones on the fly
- Grab more nodes for more memory/disks
- …
Today’s Common Practice

- **Key:** fix a small number of *stateful* nodes
  - clustered web services with DB backend
  - internet computing (*xxx@home*)
- **Stateless nodes can join/leave without effort**
- **Limitations:**
  - stateless nodes cannot communicate with each other
  - stateful part is difficult to reconfigure
  - scalability is limited by stateful part
Phoenix Goal (Restated)

- Allow *all* nodes to assume a part of application-level states
- Allow *all* nodes to communicate with each other (cf. message passing)
- Allow *all* nodes to join/leave
Talk Outline

- What’s the problem?
- Phoenix basic ideas and API
- Case study
  - Divide-and-conquer
  - LU factorization
- Experimental results
Why Difficult at all?

- A simple problem: when you send a message to processor $p$, how you make sure $p$ is still there and holds the relevant data?
Phoenix Basic Idea

- A message is *not* sent to a physical processor (*physical node name*), but to “wherever this piece of data is” (*virtual node name*)
Virtual Node Name (VP)

- An integer within a range (VP space)
- VP space is
  - chosen by the application’s convenience
  - *fixed* throughout the application’s lifetime
Phoenix API

- Communication
  - `ph_send(v, m);`
  - `m = ph_recv();`
  - `v`: virtual node name
- Mapping
  - `ph_assume(S);`
  - `ph_release(S);`
  - `S`: a set of virtual node names
ph_assume($S$)/ph_release($S$)

- Establish virtual ↔ physical mapping
  - each (physical) node assumes a set of VPs
  - typically a range, but not necessarily
- Participating nodes cover the VP space without duplication
Programming with Phoenix

- Associate each piece of application-state with a VP
  - e.g., associate $A[x]$ with VP $x$
- To operate on a remote datum, send a message to the VP associated with it
A Simple Example

- $A$: distributed array of $N$ elements
- Goal: increment all elements by one
- In MPI:
  
  for all nodes $p$ {
    MPI_Send($p$, “increment”);
  }

/* $p$ messages */
A Naïve Way

- Mapping $A[x] \leftrightarrow VP \ x$
- for ($x = 0; x < N; x++$) {
  ph_send($x$, "increment");
} /* $N$ messages ! */

⇒ would result in too many fine-grain messages
Reducing Messages by Combining

- \text{Msg} \ “\text{inc} \ [x,y])” \ means \ elements \ [x, y) \ should \ be \ incremented
Migration

- Maintain data ↔ VP mapping *invariant*

| S: ph_release({10});  
<table>
<thead>
<tr>
<th>send $A[10]$ to $R$;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ph_assume({10});</td>
</tr>
</tbody>
</table>
Resource Reconfigurability — Join

- When a new node $P$ joins
  - $P$ somehow determines its assuming VPs. e.g.,
    - $P$ sends a request to a random VP
    - the receiver ($Q$) splits its VPs and give one ($S$) to $P$
    - $Q$ migrates application states associated with $S$
Resource Reconfigurability

— Leave

- When node $P$ leaves
  - determines a node ($Q$) to migrate its states to
  - migrate application states associated with its VPs to $Q$
Application Studies

- Divide-and-conquer
  - dynamic task creation
- LU
  - 2-dimensional cyclic partitioning analogue
Divide and Conquer (DaC) Algorithms

- **Primitives**
  - create: create a child task
  - wait: wait for a child to finish and get the result

- **Applications**
  - sorting, matrix multiply
  - tree search (games, optimizations, combinatorial search)
Lazy Task Creation (LTC) for DaC Algorithms

- Each processor schedules tasks in DFS (stack) manner
- Tasks may move to an idle node
Synchronization

- How to sync a task with its remote child?
  - give a task a globally unique id
  - select unique id of a new task from assuming VPs
  - when a task syncs, it sends itself to its home VP
LU Factorization

for (i = 0; i < n; i++) {
    for (j = i+1; j < n; j++) {
        A[j][i] /= A[i][i];
        for (k = i+1; k < n; k++) {
        }
    }
}

multicast along rows and columns
Partitioning for LU Factorization

- Block partitioning analogue:
  \[ a[x] \leftrightarrow VP \ x \ (a : \text{row-major repr}) \]
  leads to poor load balance

- Better load balancing (cyclic analogue):
  \[ a[x] \leftrightarrow (K \ x) \mod \ (# \ of \ elements) \]
Experiments

- 2 SunBlade clusters (16/128 nodes) + 1 SunFire15K (60 CPUs)
- 100Mbps Ethernet within clusters
- approx. 60Mbps between clusters
Applications

- Three applications
  - Bintree
    - divide-and-conquer + random work stealing
  - Raytrace
    - Parallelization of POV-Ray
  - LU
    - LU factorization of a dense matrix
- Run them with static and dynamic resources
Bintree/Raytrace Static
LU Static

LU speedup with fixed resources

- MPICH
- 1 LAN
- 3 LANs

Number of CPUs vs. Speedup
Bintree/Raytrace Dynamic

- Bintree performance with dynamic resources
  - Dynamic vs. Fixed

- POV-Ray performance with dynamic resources
  - Dynamic vs. Fixed

Charts show relative performance over time (sec) for Bintree and POV-Ray with dynamic and fixed resource allocation.
LU Dynamic

LU performance with dynamic resources

- Dynamic
- Fixed

Relative performance (base: 1 CPU throughput)

Time (sec)
Experiments Summary

- **Divide-and-conquer applications**
  - scale well within and across LANs
  - quickly pick up dynamically added resources

- **LU**
  - suffers from bottleneck across LANs
  - suffers from a large migration cost
Related Work (1)

tools supporting resource-reconfigurability

- Nimrod [Buyya et al. 2000], PBS, LSF, SETI@home, fightAIDS@home, …
- based on stateless clients that do not communicate
Related Work (2)
PVM/Dynamic MPI

- PVM and MPI-2 include dynamic processes
  - naming/membership issues left to programmers
Related Work (3)

Other RR Computation Models

- Dynamic Space Sharing [Chowdhury et al. 97]
  - exact computation model unclear (“we restrict our attention to iterative grid-based applications”)
  - appear to assume “reconfigurable snapshots” to be taken by the application
Related Work (4) Overlay Networks

- scalable P2P file/information sharing systems
  - CAN [Ratnasamy et al. 01], Pastry [Rawstron et al. 01], Chord [Stocia et al. 01], Tapestry [Zhao et al. 00]
  - \( \text{ph}_\text{send}(dst, m) \leftrightarrow \text{search}(key) \)

- Phoenix \( \approx \) DHT seen as a parallel programming model
  - we believe writing parallel applications with DHT have been made clear by us
  - migration protocols are not their main issues
Summary

- With simple changes/extensions to message passing, namely,
  - a fixed "virtual node name" spaces and
  - node name assume/release primitives,
- general message passing apps can support resource reconfigurations
- Overhead imposed by the lack of knowledge about the # of processors was not significant
Future Work

- Study a wider range of array distributions (esp. 2-dimensional cases)
- Demonstration of applications with actual resource reconfiguration policy
- FT and Phoenix: many possibilities
  - state duplication
  - checkpointing