Concurrent and Distributed Programming Patterns in SALSA

Travis Desell
Carlos Varela
RPI
November 6, 2009

Overview

- Programming techniques and patterns
  - farmer-worker computations,
  - iterative computations,
  - peer-to-peer agent networks,
  - soft real-time: priorities, delays
  - causal connections: named tokens, waitfor property
- Distributed runtime architecture (World-Wide Computer)
  - architecture and implementation
  - distributed garbage collection
- Autonomic computing (Internet Operating System)
  - architecture and implementation
  - autonomous migration
  - split and merge
- Distributed systems visualization (OverView)

Farmer Worker Computations

- Most common “Massively Parallel” type of computation
- Workers repeatedly request tasks or jobs from farmer and process them

Twin Primes & Brun’s Constant

-Investigators:
  P. H. Fry, J. Nesheiwat, B. Szymanski (RPI CS)
-Problem Statement:
  Are there infinitely many twin primes?
  Calculating Brun’s Constant (sum of inverse of all twin primes) to highest accuracy.
-Application Information (Twin Primes):
  Massively Parallel
  Farmer/Worker

Iterative Computations

- Common pattern for partial differential equations, scientific computing and distributed simulation
- Workers connected to neighbors
- Data location dependent
- Workers process an iteration with results from neighbors, then send results to neighbors
- Performance bounded by slowest worker
Adaptive Partial Differential Equation Solvers

- Investigators:
  J. Flaherty, M. Shephard, B. Szymanski, C. Varela (RPI)
  J. Teresco (Williams), E. Deelman (ISI-UCI)
- Problem Statement: How to dynamically adapt solutions to PDEs to account for underlying computing infrastructure?
- Applications/Implications: Materials fabrication, biomechanics, fluid dynamics, aeronautical design, ecology.
- Approach:
  - Partition problem and dynamically map into computing infrastructure and balance load.
  - Low communication overhead over low-latency connections.
- Software:
  - Rensselaer Partition Model (RPM)
  - Algorithm Oriented Mesh Database (AOMD)
  - Dynamic Resource Utilization Model (DRUM)
- Application Information (Heat):
  - Tightly coupled
  - Iterative

Case Study: Heat Diffusion Problem

- A problem that models heat transfer in a solid
- A two-dimensional mesh is used to represent the problem data space
- An Iterative Application
- Highly synchronized

Parallel Decomposition of the Heat Problem

Peer-to-Peer Computations
Peer-to-peer systems (1)

- Network transparency works well for a small number of nodes, what do we do when the number of nodes becomes very large?
  - This is what is happening now
- We need a scalable way to handle large numbers of nodes
- Peer-to-peer systems provide one solution
  - A distributed system that connects resources located at the edges of the Internet
  - Resources: storage, computation power, information, etc.
  - Peer software: all nodes are functionally equivalent
- Dynamic
  - Peers join and leave frequently
  - Failures are unavoidable

Peer-to-peer systems (2)

- Unstructured systems
  - Napster (first generation): still had centralized directory
  - Gnutella, Kazaa, ... (second generation): neighbor graph, fully decentralized but no guarantees, often uses superpeer structure
- Structured overlay networks (third generation)
  - Using non-random topologies
  - Strong guarantees on routing and message delivery
  - Testing on realistically harsh environments (e.g., PlanetLab)
  - DHT (Distributed Hash Table) provides lookup functionality
    - Many examples: Chord, CAN, Pastry, Tornado, P-Grid, DKS, Viceroy, Tango, Koorde, etc.

Examples of P2P networks

- Hybrid (client/server)
  - Napster
- Unstructured P2P
  - Gnutella
- Structured P2P
  - Exponential network
  - DHT (Distributed Hash Table), e.g., Chord

Properties of structured overlay networks

- Scalable
  - Works for any number of nodes
- Self-organizing
  - Routing tables updated with node joins/leaves
  - Routing tables updated with node failures
- Provides guarantees
  - If operated inside of failure model, communication is guaranteed with an upper bound on number of hops
  - Broadcast can be done with a minimum number of messages
- Provides basic services
  - Name-based communication (point-to-point and group)
  - DHT (Distributed Hash Table): efficient storage and retrieval of (key,value) pairs

Self organization

- Maintaining the routing tables
  - Correction-on-use (lazy approach)
  - Periodic correction (eager approach)
  - Guided by assumptions on traffic
- Cost
  - Depends on structure
  - A typical algorithm, DKS (distributed k-ary search), achieves logarithmic cost for reconfiguration and for key resolution (lookup)
- Example of lookup for Chord, the first well-known structured overlay network

Chord: lookup illustrated

Given a key, find the value associated to the key
(here, the value is the IP address of the node that stores the key)

Assume node i searches for the value associated to key K with virtual identifier 7

<table>
<thead>
<tr>
<th>Interval</th>
<th>node to be contacted</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>0</td>
</tr>
<tr>
<td>(1,2)</td>
<td>6</td>
</tr>
<tr>
<td>(2,4)</td>
<td>6</td>
</tr>
<tr>
<td>(4,8)</td>
<td>6</td>
</tr>
<tr>
<td>(8,15)</td>
<td>12</td>
</tr>
</tbody>
</table>
Soft Real-Time

Message Properties

- SALSA provides message properties to control message sending behavior:
  - delay
    - To delay sending a message to an actor for a given time
  - waitfor
    - To delay sending a message to an actor until a token is available

Delayed Message Sending

- To (asynchronously) send a message after a given delay in milliseconds:
  
  ```
  a <- book(flight):delay(1000);
  ```

  Message is sent after one second has passed.

Causal Connections

Synchronized Message Sending

- To (asynchronously) send a message after another message has been processed:
  
  ```
  token fundsOk = bank <- checkBalance();
  a <- book(flight):waitfor(fundsOk);
  ```

  Message is sent after token has been produced.

Named Tokens

- Tokens can be named to enable more loosely-coupled synchronization
  
  ```
  token t1 = a1 <- m1();
  token t2 = a2 <- m2();
  token t3 = a3 <- m3(t1);
  token t4 = a4 <- m4(t2);
  a <- m(t1,t2,t3,t4);
  ```

  Sending m(...) to a will be delayed until messages m1()...m4() have been processed. m2() can proceed concurrently with m3().
Named Tokens (Multicast)

- Named tokens enable multicast:
  - Example:
    ```java
    token t1 = a1 <- m1();
    for (int i = 0; i < a.length; i++) a[i] <- m(t1);
    ```
    Sends the result of m1 to each actor in array a.

- Named tokens enable multicast:
  - Example:
    ```java
    token t1 = a1 <- m1();
    for (int i = 0; i < a.length; i++) a[i] <- m(t1);
    ```
    Sends the result of m1 to each actor in array a.

Join Blocks

- Join blocks allow for synchronization over multiple messages
- Join blocks return an array of objects (Object[]), containing the results of each message sent within the join block. The results are in the same order as how the messages they were generated by were sent.

  - Example:
    ```java
    token t1 = a1 <- m1();
    join {
      for (int i = 0; i < a.length; i++) {
        a[i] <- m(t1);
      }
    } @ process(token);
    ```
    Sends the message m with the result of m1 to each actor in array a. After all the messages m have been processed, their results are sent as the arguments to process.

Current Continuations

- Current Continuations allow for first class access to a messages continuation
- Current Continuations enable recursion

  - Example:
    ```java
    int fibonacci(int n) {
      if (n == 0) return 0;
      else if (n == 1 || n == 2) return 1;
      else {
        token a = fibonacci(n - 1);
        token b = fibonacci(n - 2);
        add(a, b) @ currentContinuation;
      }
    }
    ```
    Finds the nth fibonacci number. The result of add(a, b) is sent as the return value of fibonacci to the next message in the continuation.

  - Example:
    ```java
    String getAnswer(Object question) {
      if (question instanceof Question1) {
        knowsQ1 <- getAnswer(question) @ currentContinuation;
      } else if (question instanceof Question2) {
        knowsQ2 <- getAnswer(question) @ currentContinuation;
      } else return "don't know!";
    }
    ```
    If the question is Question1 this will get the answer from actor knowsQ1 and pass this result as it's token, if the question is Question2 this will get the answer from actor knowsQ2 and pass that result as it's token, otherwise it will return "don't know!".

Current Continuations (Delegation)

- Current Continuations can also be used to delegate tasks to other actors

  - Example:
    ```java
    String getAnswer2(Object question) {
      if (question instanceof Question1) {
        knowsQ1 <- getAnswer2(question) @ currentContinuation;
      } else if (question instanceof Question2) {
        knowsQ2 <- getAnswer2(question) @ currentContinuation;
      } else return "don't know!";
    }
    ```
    If the question is Question1 this will get the answer from actor knowsQ1 and pass this result as it's token, if the question is Question2 this will get the answer from actor knowsQ2 and pass that result as it's token, otherwise it will return "don't know!".

Current Continuations (Loops)

- Current Continuations allow for synchronized loops

  - Example:
    ```java
    token t1 = initial;
    for (int i = 0; i < n; i++) {
      t1 = m(t1);
    }
    ```
    Sends m to a n times, passing the result of the previous m as an argument.

  - Example:
    ```java
    token t1 = null;
    for (int i = 0; i < a.length; i++) {
      t1 = a[i] <- m(i) @ waitfor(t1);
    }
    ```
    Sends m(i) to actor a[i], message m(i) will wait for m(i-1) to be processed.

Current Continuations (Loops)

- Current Continuations can also be used to perform recursive loops

  - Example:
    ```java
    void loop(int n) {
      if (n == 0) {
        m(n) @ currentContinuation;
      } else {
        loop(n - 1) @ currentContinuation;
        m(n) @ currentContinuation;
      }
    }
    ```
    Sends the message m(n) to each actor in array a. After all the messages m(n) have been processed, their results are sent as the arguments to currentContinuation.

Current Continuations (Delegation)

- Current Continuations can also be used to delegate tasks to other actors

  - Example:
    ```java
    String getAnswer2(Object question) {
      if (question instanceof Question1) {
        knowsQ1 <- getAnswer2(question) @ currentContinuation;
      } else if (question instanceof Question2) {
        knowsQ2 <- getAnswer2(question) @ currentContinuation;
      } else return "don't know!";
    }
    ```
    If the question is Question1 this will get the answer from actor knowsQ1 and pass this result as it's token, if the question is Question2 this will get the answer from actor knowsQ2 and pass that result as it's token, otherwise it will return "don't know!".

Travis Desell and Carlos Varela
Distributed run-time (WWC)

World-Wide Computer Architecture

- SALSA application layer
  - Programming language constructs for actor communication, migration, and coordination.
- IOS middleware layer
  - A Resource Profiling Component
    - Captures information about actor and network topologies and available resources
  - A Decision Component
    - Takes migration, split/merge, or replication decisions based on profiled information
  - A Protocol Component
    - Performs communication between nodes in the middleware system
- WWC run-time layer
  - Theaters provide runtime support for actor execution and access to local resources
  - Pluggable transport, naming, and messaging services

WWC Theaters

Scheduling

- The choice of which actor gets to execute next and for how long is done by a part of the system called the scheduler.
- An actor is non-blocked if it is processing a message or if its mailbox is not empty, otherwise the actor is blocked.
- A scheduler is fair if it does not starve a non-blocked actor, i.e. all non-blocked actors eventually execute.
- Fair scheduling makes it easier to reason about programs and program composition.
  - Otherwise some correct program (in isolation) may never get processing time when composed with other programs.

Remote Message Sending Protocol

- Messages between remote actors are sent using the Remote Message Sending Protocol (RMSP).
- RMSP is implemented using Java object serialization.
- RMSP protocol is used for both message sending and actor migration.
- When an actor migrates, its location (host:port) changes but its name (UAN) does not.

Universal Actor Naming Protocol

- Messages between remote actors are sent using the Remote Message Sending Protocol (RMSP).
- RMSP is implemented using Java object serialization.
- RMSP protocol is used for both message sending and actor migration.
- When an actor migrates, its location (host:port) changes but its name (UAN) does not.
Universal Actor Naming Protocol

- UANP includes messages for:
  - Binding actors to UAN, host:port pairs
  - Finding the locator of a universal actor given its UAN
  - Updating the locator of a universal actor as it migrates
  - Removing a universal actor entry from the naming service

- SALSA programmers need not use UANP directly in programs. UANP messages are transparently sent by WWC run-time system.

UANP Implementations

- Default naming service implementation stores UAN to host-port and unique ID mapping in name servers as defined in UANs.
  - Name server failures may induce universal actor unreachability.
- Distributed (Chord-based) implementation uses consistent hashing and a ring of connected servers for fault-tolerance. For more information, see:

Naming Service Requirements

- Efficient name resolution/updates
- Fault tolerance
- Scalability
- Security
- Names should be:
  - Globally unique
  - Persistent
  - Human readable

Fault-Tolerant Home-Based Naming Service (FHNS)

- Each agent has a home base that keeps track of agent’s location.
- Home base can be specified by agent or assigned by the naming service.
- Two types of URI-based names:
  - Location-dependent names (host:port and unique identifier)
  - Location-independent names (URN)

Naming Service API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(name, location)</td>
<td>Binds a name to an initial location.</td>
</tr>
<tr>
<td>get(name)</td>
<td>Returns the location for the given name.</td>
</tr>
<tr>
<td>update(name, location)</td>
<td>Updates the location for the given name.</td>
</tr>
<tr>
<td>delete(name)</td>
<td>Remove the location for the given name.</td>
</tr>
</tbody>
</table>

Home Bases, Agents and Identifiers

<table>
<thead>
<tr>
<th>Home Base Name</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rome:3030</td>
<td>0</td>
</tr>
<tr>
<td>Hongkong:4000</td>
<td>0</td>
</tr>
<tr>
<td>Newyork:5050</td>
<td>5</td>
</tr>
<tr>
<td>Drifters</td>
<td></td>
</tr>
<tr>
<td>Gypsy</td>
<td>4</td>
</tr>
<tr>
<td>Nomad</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Agent Name</th>
<th>Identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Rome:3030 Migrant</td>
<td>0</td>
</tr>
<tr>
<td>Base Hongkong:4000 Tumbleweed</td>
<td>0</td>
</tr>
<tr>
<td>Base Drifters Gypsy</td>
<td>4</td>
</tr>
<tr>
<td>Base Drifters Nomad</td>
<td>2</td>
</tr>
<tr>
<td>Base Drifters Drifters</td>
<td>7</td>
</tr>
</tbody>
</table>
Virtual Ring of Home Bases

Home Base Theater Components

Home Base Failures

- Failing home bases sequentially from 8 to 1 ensures that naming service is still available.
- Lookup is logarithmic on number of home bases.

Name Resolution Efficiency

- Logarithmic growth in number of messages to service a naming request.
  a: Theoretical results
  b: Practical results

Network latency and Request Processing Times

- While number of messages increases logarithmically, request processing times increase linearly with number of hops for location-independent names.
- Location-dependent names take a constant lookup time.

Load Balancing

- Given K identifiers and H home bases, ideally each home base hosts K/H identifiers.
- Two nodes with similar hash values create imbalance.
  a: Identifier Coverage
  b: Agent Name Coverage
Load Balancing with Virtual Nodes

- a: Namespace coverage among 4 home bases.
- b: 4 home bases with 8 virtual nodes
- c: Namespace coverage with virtual nodes

Actor Garbage Collection

- Implemented since SALSA 1.0 using pseudo-root approach.
- Includes distributed cyclic garbage collection.
- For more details, please see:

Motivation

- Actors are reactive concurrent mobile entities.
- Garbage collection is needed for high level programming.
- Why do we need a new algorithm for actor garbage collection?
  - Existing algorithms rely on First-In-First-Out Communication or blocking communication
  - The new concept of mobile actors
- The need for automatic actor garbage collection in the SALSA programming language

Actor Garbage Definition

- Terminology:
  - Blocked: an idle actor with an empty message box
  - Unblocked: it is not blocked
  - Root: A root actor is defined as being always live (useful).
- The definition of actor garbage relates to meaningful computations
  - An application must be able to produce computing results!

Operational Definition of Live Actors

- The new definition relies on “potentially live”:
  - Every unblocked actor is potentially live.
  - Every acquaintance of a potentially live actor is potentially live.
- The new definition of live actors:
  - A root actor is live.
  - Every acquaintance of a live actor is live.
  - Every potentially live, inverse acquaintance of a live actor is live.
Assumption of Resource (Root) Access

- We assume every actor has persistent references to some root.
  - It is realistic in any kind of programming language!
- The resource access assumption leads to the live unblocked actor principle!
  - It makes every potentially live actor live!

Challenge 2: Non-blocking communication

- Following references to mark live actors is not safe!
- What can we do?
  - We can protect the reference from deletion and mark the sender live until the sender knows the message has arrived.
- How can we guarantee the safety of an actor referenced by a message?
  - The solution is to protect the reference from deletion and mark the sender live until the sender knows the message has arrived.

Challenge 3: Distribution and Mobility

- What if an actor is remotely referenced?
  - We can maintain an inverse reference list that every actor is the garbage collector to indicate whether an actor is referenced.
  - The inverse reference registration must be non-blocking and non-First-In-First-Out communication!
  - Three operations change inverse references: actor creation, reference passing, and reference deletion.
Challenge 3: Distribution and Mobility (continued)

- What if an actor is remotely referenced?
  - We can maintain an inverse reference list (only visible to the garbage collector) to indicate whether an actor is referenced.
  - The inverse reference registration must be based on non-blocking and non-First-In-First-Out communication.
  - Three operations are involved: actor creation, reference passing, and reference deletion.

The Concept of the Pseudo Root Approach

- Pseudo roots:
  - Treat unblocked actors, migrating actors, and roots as pseudo roots.
  - Map in-transit messages and references into protected references and pseudo roots.
  - Use inverse reference list (only visible to garbage collectors) to identify remotely referenced actors.
- Actors which are not reachable from any pseudo root are garbage.

Property of the Pseudo Root Approach (1)

- Lemma 1. Precise inverse references to non-pseudo-root actors.
  - Let \( x \rightarrow y \). If Actor \( y \) is referenced by a non-pseudo-root actor \( x \), actor \( y \) must have an inverse reference to Actor \( x \).
**Property of the Pseudo Root Approach (2)**

- Lemma 2. Safe imprecise inverse references to pseudo root actors.
  - If an actor is referenced by several pseudo roots, either 1) it has at least one inverse reference to one of the pseudo roots, or 2) it is a pseudo root.

**Property of the Pseudo Root Approach (3)**

- Theorem 1. One-step back tracing safety.
  - Let Actor x be remotely referenced. Actor x can be identified live by one-step back tracing through its registered inverse references, or it is transitively reachable from a local pseudo root.

**Challenge 3: Distribution and Mobility (continued)**

- How to collect global mutually referenced garbage (cycles)?
- Our solution is to use a snapshot based algorithm.

**Example of the Snapshot Algorithm**

**Experiments**

- We use three kinds of applications to evaluate our algorithm.
  - The Fibonacci Number (tree structure computing)
  - The N-Queens Number Problem (Master-Worker)
  - Matrix Multiplication (Master-Worker with heavy communication)

**Results**

<table>
<thead>
<tr>
<th>Application</th>
<th>Overhead</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fibonacci</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-Queens</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matrix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Autonomic Computing (IOS)**

**Middleware for Autonomous Computing**
- Middleware
  - A software layer between distributed applications and operating systems.
  - Alleviates application programmers from directly dealing with distribution issues.
    - Heterogeneous hardware/OSs
    - Load balancing
    - Fault tolerance
    - Security
    - Quality of service
- Internet Operating System (IOS)
  - A decentralized framework for adaptive, scalable execution.
  - Evaluates different distribution and reconfiguration strategies.


**Middleware Implementation - IOS**

- Leveraged IOS (The Internet Operating System):
  - Generic middleware for distributed application reconfiguration.
  - Works with multiple programming paradigms (MPI, SALSA)

**Middleware Architecture**

- IOS middleware layer
  - A Resource Profiling Component
    - Captures information about actor and network topologies and available resources
  - A Decision Component
    - Takes migration, split/merge, or replication decisions based on profiled information
  - A Protocol Component
    - Performs communication with other agents in virtual network (e.g., peer-to-peer, cluster-to-cluster, centralized)

**A Generic Architecture**

**IOS Architecture**

- IOS Theater
  - Profiling Agent
    - Evaluates a potential migration
    - Monitors and reports actor information
  - Decision Agent
    - Evaluates a potential migration
    - Monitors and reports actor information
  - Protocol Agent
    - Initiates a steal request
    - Sends steal requests
    - Receives steal requests
  - IOS Theater
    - Application Entity
      - Monitors and reports actor information
**IOS API**

- The following methods notify the profiling agent of actors entering and exiting the theater due to migration and binding:
  - public void addProfile(UAN uan);
  - public void removeProfile(UAN uan);
  - public void migrateProfile(UAN uan, UAL target);

- The profiling agent updates its actor profiles based on message sending with these methods:
  - public void msgSend(UAN uan, Msg_INFO msgInfo);

- The profiling agent updates its actor profiles based on message reception with this method:
  - public void msgReceive(UAN uan, targetUAL, Msg_INFO msgInfo);

- The following methods notify the profiling agent of the start of a message being processed and the end of a message being processed, with a UAN or UAL to identify the sending actor:
  - public void beginProcessing(UAN uan, Msg_INFO msgInfo);
  - public void endProcessing(UAN uan, Msg_INFO msgInfo);

---

**Resource Sensitive Model**

- Decision components use a resource sensitive model to decide based on the profiled applications how to balance the resources' consumption.

- Reconfiguration decisions:
  - Where to migrate
  - When to migrate
  - How many entries to migrate

---

**A General Model for Weighted Resource-Sensitive Work-Stealing (WRS)**

- Given:
  - A set of resources, \( R = \{ r_0, ..., r_n \} \)
  - A set of actors, \( A = \{ a_0, ..., a_n \} \)
  - \( \omega \) is a weight, based on importance of the resource \( r \) to the performance of a set of actors \( A \)
  - \( 0 \leq \omega(r, A) \leq 1 \)
  - \( \sum_{all r} \omega(r, A) = 1 \)

- \( \alpha(r, f) \) is the amount of resource \( r \) available at foreign node \( f \)
- \( \nu(r, l, A) \) is the amount of resource \( r \) used by actors \( A \) at local node \( l \)
- \( M(A, l, f) \) is the estimated cost of migration of actors \( A \) from \( l \) to \( f \)
- \( L(A) \) is the average life expectancy of the set of actors \( A \)

- The predicted increase in overall performance \( \Gamma \) gained by migrating \( A \) from \( l \) to \( f \), where \( \Gamma \leq 1 \):
  \[
  \Delta(r, l, f, A) = \frac{\alpha(r, f) - \nu(r, l, A)}{\alpha(r, f) + \nu(r, l, A)}
  \]
  \[
  \Gamma = \sum_{all r} (\omega(r, A) \times \Delta(r, l, f, A)) - \frac{M(A, l, f)}{10 + \log L(A)}
  \]

- When work requested by \( f \), migrate actors \( A \) with greatest predicted increase in overall performance, if positive.

---

**Virtual Topologies of IOS Agents**

- Agents organize themselves in various network-sensitive virtual topologies to sense the underlying physical environments.

- Peer-to-peer topology: agents form a p2p network to exchange profiled information.

- Cluster-to-cluster topology: agents organize themselves in groups of clusters. Cluster managers form a p2p network.

---

**IOS Load Balancing Strategies**

- IOS modular architecture enables using different load balancing and profiling strategies, e.g.:
  - Round-robin (RR)
  - Random work-stealing (RS)
  - Application topology-sensitive work-stealing (ATS)
  - Network topology-sensitive work-stealing (NTS)
Random Stealing (RS)
- Based on Cilk’s random work stealing
- Lightly-loaded nodes periodically send work steal packets to randomly picked peer theaters
- Application entities migrate from highly loaded theaters to lightly loaded theaters
- Simple strategy: no broadcasts required
- Stable strategy: it avoids additional traffic on overloaded networks

Application Topology-Sensitive Work-Stealing (ATS)
- An extension of RS to collocate components that communicate frequently
- Decision agent picks the component that will minimize intra-node communication after migration, based on
  - Location of acquaintances
  - Profiled communication history
- Tries to minimize the frequency of remote communication improving overall system throughput

Network Topology-Sensitive Work-Stealing (NTS)
- An extension of ATS to take the network topology and performance into consideration
- Periodically profile end-to-end network performance among peer theaters
  - Latency
  - Bandwidth
- Tries to minimize the cost of remote communication improving overall system throughput
  - Tightly coupled entities stay within reasonably low latencies high bandwidths
  - Loosely coupled entities can flow more freely

Using the IOS middleware
- Start IOS Peer Servers: a mechanism for peer discovery
- Start a network of IOS theaters
- Write your SALSA programs and extend all actors to autonomous actors
- Start autonomous actors in theaters
- IOS automatically reconfigures the location of actors in the network for improved performance of the application.
- IOS supports the dynamic addition and removal of theaters

Preliminary Results---Unconnected/Sparse
- Load balancing experiments use RR, RS and ATS
- Applications with diverse inter-actor communication topologies
  - Unconnected, sparse, tree, and hypercube actor graphs

Tree and Hypercube Topology Results
- RS and ATS do not add substantial overhead to RR
- ATS performs best in all cases with some interconnectivity
Load Balancing Strategies for Internet-like and Grid-like Environments

• Simulation results show that:
  – The peer-to-peer protocol performs better for applications with high communication-to-computation ratio in Internet-like environments
  – The cluster-to-cluster protocol performs better for applications with low communication-to-computation ratio in Grid-like environments

Migration Policies

• Group Migration performs better for the 4 application topologies.
• Single Migration has a more stable behavior of the application’s topology throughput.
• Future Work: Evaluation of migration policies for different sizes of actors.

Dynamic Networks

• Theaters were added and removed dynamically to test scalability.
• During the 1st half of the experiment, every 30 seconds, a theater was added.
• During the 2nd half, every 30 seconds, a theater was removed.
• Throughput improves as the number of theaters grows.
Component Malleability

- New type of reconfiguration:
  - Applications can dynamically change component granularity
- Malleability can provide many benefits for HPC applications:
  - Can more adequately reconfigure applications in response to a dynamically changing environment:
    - Can scale application in response to dynamically joining resources to improve performance.
    - Can provide soft fault-tolerance in response to dynamically leaving resources.
  - Can be used to find the ideal granularity for different architectures.
  - Easier programming of concurrent applications, as parallelism can be provided transparently.

Impact of Granularity on Runtime

- Impact of Granularity on Different Machine Architectures

Methodology

- How to accomplish dynamic granularity?
  - Programmers define how to split and merge components.
  - Middleware determines which components to split or merge, and when to perform split and merge.

Types of Malleability

- How can split and merge be done?
  - Split 1:2, Merge 2:1
  - Split 1:N, Merge N:1
  - Split N:M, Merge M:N
  - Split N:N+1, Merge N+1:N

Implementing Split/Merge

- Leveraged the SALSA programming language:
  - Actor oriented programming model
  - Compiled to Java
- Added language level support for malleable actors.
- Generic strategy used to split and merge actors:
  - Performed Atomically
  - Allows Concurrency
  - User specifies via API:
    - communication redirection
    - data redistribution
Example – Twin Primes Farmer

```java
behavior TPFarmer {
    NumberSegmentGenerator nsg = new NumberSegmentGenerator();
    void act(String[] args) {
        numberWorkers = Integer.parseInt(args[0]);
        for (int i = 0; i < numberWorkers; i++) new TPWorker(this);
    }
    void requestWork(TPWorker w) {
        if (nsg.hasMoreSegments())
            w.findPrimes(nsg.getNextSegment());
    }
    void receivePrimes(Segment s) {
        s.saveToDisk();
    }
}
```

Example - Twin Primes Worker

```java
behavior TPWorker extends MalleableActor {
    TPFarmer farmer;
    TPWorker(TPFarmer farmer) {
        this.farmer = farmer;
        farmer.requestWork(this);
    }
    void findPrimes(Segment s) {
        s.findPrimes();
        farmer.receivePrimes(s);
        farmer.requestWork(this);
    }
    boolean canSplitOrMerge() {
        return true;
    }
    MalleableActor createNewActor() {
        return new TPWorker(farmer);
    }
    void handleMergeMsg(Msg m) {
        if (m.name() == "findPrimes") {
            this.process(message);
        }
    }
}
```

Distributed Systems Visualization

• Generic online Java-based distributed systems visualization tool
• Uses a declarative Entity Specification Language (ESL)
• Instruments byte-code to send events to visualization layer.
• For more details, please see:


AstroInformatics and PhysInformatics Application Examples

OverView Heat Example With 5 Initial Nodes

http://wcl.cs.rpi.edu/overview/
Travis Desell and Carlos Varela

Milky Way Origin and Structure

Particle Physics

• Investigators:
  H. Newberg (RPI Astronomy), J. Teresco (Williams)
  M. Maggio-Ismail, B. Szymanski, C. Varela (RPI CS)
  J. Cummings, J. Napolitano (RPI Physics)

• Problem Statement:
  How to analyze data from 10,000 square degrees of the north galactic cap
  collected in five optical filters over five years by the Sloan Digital Sky
  Survey?
  Do "missing baryons" exist? Sub-atomic particles that have not been observed.

• Applications/Implications:
  Astrophysics: origins and evolution of our galaxy.
  Physics: particle physics, search for missing baryons.

• Approach:
  To use photometric and spectroscopic data to separate and describe components
  of the Milky Way
  Maximum Likelihood Analysis

Analysis of Particle Physics

• Facts:
  - Short lived particles (10^-1 s) are not measurable
  - They can only be inferred from correlations in the final state particles

• How to infer?
  - Partial Wave Analysis (PWA) to describe the properties of the particles
  - Maximum Likelihood Evaluation (MLE) to find the most possible solution

Maximum Likelihood Evaluation (MLE)

• Maximum likelihood: the largest product of the
  probabilities of observing each event given a set of fit
  parameters.
  - In our case:

  \[
  -\ln(L) = -\sum_{i=1}^{n} \ln \left( \left( \sum_{j} \alpha_j \left( x_j \right) \right) \right) - \sum_{i=1}^{m} \ln \left( \left( \sum_{j} \beta_j \left( y_j \right) \right) \right) + \text{constant}
  \]

Simplex Algorithm

• Simplex is an algorithm to iteratively
  compute the maximum likelihood
  - Apply to N-dimensional problems
  - Fix one point and try to find the
  minimal value
  - Contract the simplex if necessary
  - Improve the result iteratively
  - Need to guess the initial value

Simplex Algorithm Example

• Given an initial value, then use
  it to fit

• If found the optimal value, contract the simplex and try
  again
**Simplex Algorithm Example**

- Contract to a smaller simplex and find the optimal

**Simplex Algorithm Example**

- If the gain from contracting the simplex is small enough, then stop fitting

**Final Remarks**

- Thanks!
- Visit our web pages:
  - SALSA: http://wcl.cs.rpi.edu/salsa/
  - IOS: http://wcl.cs.rpi.edu/ios/
  -_overView: http://wcl.cs.rpi.edu/overview/
- Questions?

**Exercises**

1. Create a Producer-Consumer pattern in SALSA and play with message delays to ensure that the consumer actor mailbox does not create a memory problem.
2. Create an autonomous iterative application and run it within IOS so that the management of actor placement is triggered by the middleware.
3. Execute the Cell example with OverView visualizing actor migration.