Concurrent and Distributed Programming Patterns

Carlos Varela

RPI

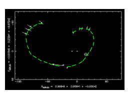
November 20, 2009

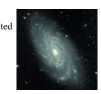
Overview

- A motivating application in AstroInformatics
- 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)
  - autonomous migration
  - split and merge
- Distributed systems visualization (OverView)

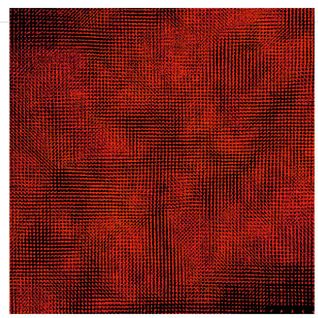
Milky Way Origin and Structure

- Principal Investigators:
  - H. Newberg (RPI Astronomy),
  - M. Magdon-Ismail, B. Szymanski, C. Varela (RPI CS)
- Students:
  - N. Cole (RPI Astronomy), T. Desell, J. Doran (RPI CS)
- Problem Statement:
  - What is the structure and origin of the Milky Way galaxy?
  - 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?
- Applications/Implications:
  - Astrophysics: origins and evolution of our galaxy.
- Approach:
  - Experimental data analysis and simulation
  - To use photometric and spectroscopic data for millions of stars to separate and describe components of the Milky Way
- Software:
  - Generic Maximum Likelihood Evaluation (GMLE) framework.
  - MilkyWay@Home BOINC project.





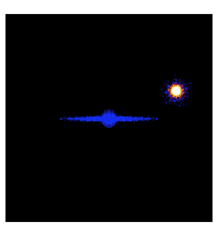
How Do Galaxies Form?



Ben Moore, Inst. Of Theo. Phys., Zurich

Tidal Streams

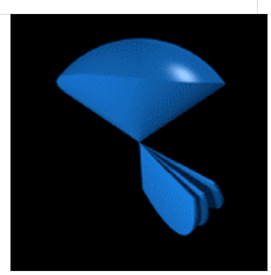
- Smaller galaxy gets tidally disrupted by larger galaxy
- Good tracer of galactic potential/dark matter
- Sagittarius Dwarf Galaxy currently being disrupted
- Three other known streams thought to be associated with dwarf galaxies



Kathryn V. Johnston, Wesleyan Univ.

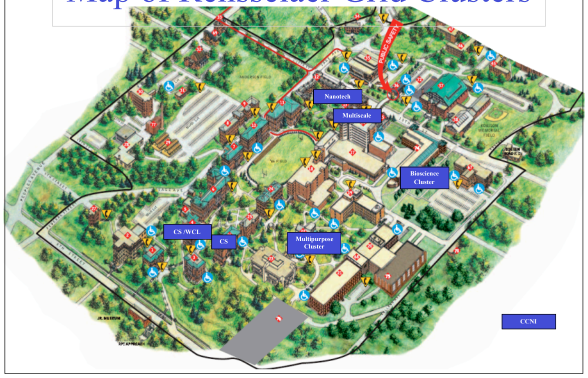
Sloan Digital Sky Survey Data

- **SDSS**
  - ~ 9,600 sq. deg.
  - ~ 287, 000, 000 objects
  - ~ 10.0 TB (images)
- **SEGUE**
  - ~ 1,200 sq. deg.
  - ~ 57, 000, 000 objects
- **GAIA (2010-2012)**
  - Over one billion estimated stars

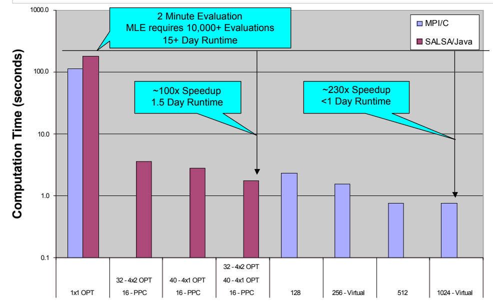


http://www.sdss.org

Map of Rensselaer Grid Clusters



Maximum Likelihood Evaluation on RPI Grid and BlueGene/L Supercomputer

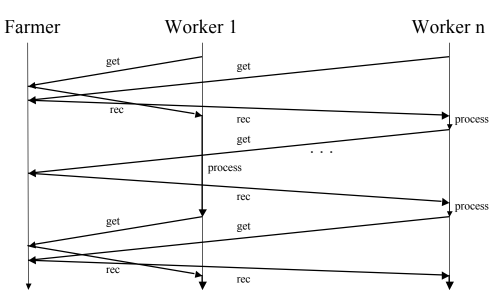


Programming Patterns

Farmer Worker Computations

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

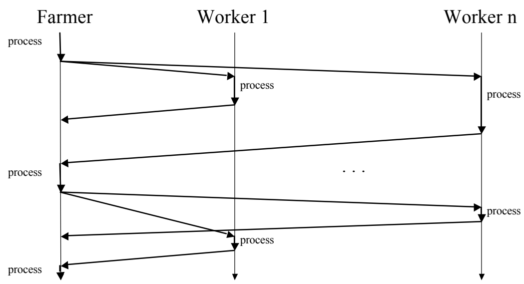
Farmer Worker Computations



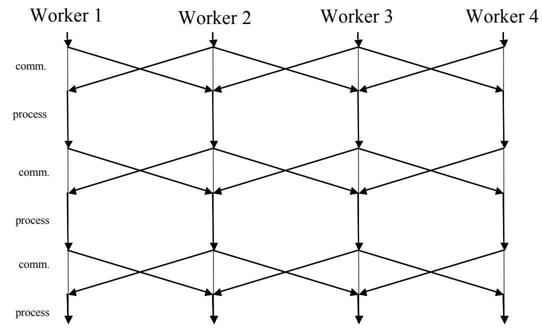
Iterative Computations

- 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

Iterative Farmer/Worker



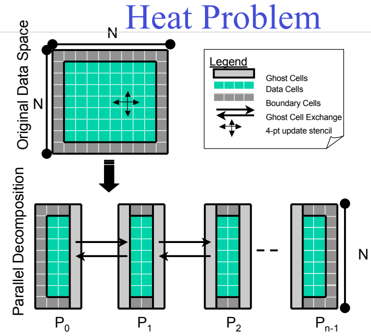
Iterative P2P



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)

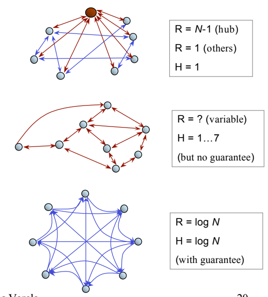
- Network transparency works well for a small number of nodes; what do we do when the number of nodes becomes very large?
  - This is what is happening now
- We need a scalable way to handle large numbers of nodes
- Peer-to-peer systems provide one solution
  - A distributed system that connects resources 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, Tapestry, 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, then 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

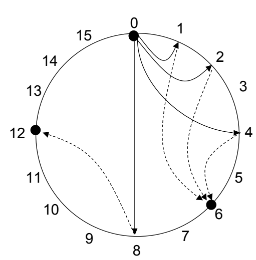
- Maintaining the routing tables
  - Correction-on-use (lazy approach)
  - Periodic correction (eager approach)
  - Guided by assumptions on traffic
- Cost
  - Depends on structure
  - A typical algorithm, DKS (distributed k-ary search), achieves logarithmic cost for reconfiguration and for key resolution (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 0 searches for the value associated to key K with virtual identifier 7

Interval	node to be contacted
[0,1)	0
[1,2)	6
[2,4)	6
[4,8)	6
[8,0)	12



● Indicates presence of a node

Soft Real-Time

Message Properties

- SALSA provides message properties to control message sending behavior:
  - priority
    - To send messages with priority to an actor
  - 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

Priority Message Sending

- To (asynchronously) send a message with high priority:

```
a <- book(flight):priority;
```

Message is placed at the beginning of the actor's mail queue.

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
  - Example:

```
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. m1() can proceed concurrently with m2().

Named Tokens (Multicast)

- Named tokens enable multicast:
  - Example:

```
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 (Loops)

- **Named tokens allow for synchronized loops:**
  - Example 1:

```
token t1 = initial;
for (int i = 0; i < n; i++) {
    t1 = a <- m( t1 );
}
```

Sends m to a n times, passing the result of the previous m as an argument.

  - Example 2 (using waitfor):

```
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.

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:

```
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 message's continuation**
- **Current continuations facilitate writing recursive computations**
  - Example:

```
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.

Current Continuations (Loops)

- **Current Continuations can also be used to perform recursive loops:**
  - Example:

```
void loop(int n) {
    if (n == 0) {
        m(n) @
        currentContinuation;
    } else {
        loop(n – 1) @
        m(n) @
        currentContinuation;
    }
}
```

Sends the messages m(0), m(1), m(2) ...m(n). m(i) is always processed after m(i-1).

Current Continuations (Delegation)

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

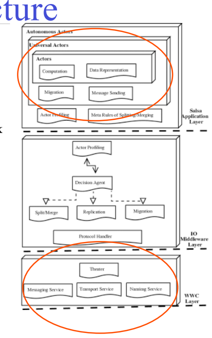
```
String getAnswer(Object question) {
    if (question instanceof Question1) {
        knowsQ1 <- getAnswer(question) @
        currentContinuation;
    } else if (question instanceof Question2) {
        knowsQ2 <- getAnswer(question) @
        currentContinuation;
    } else return "don't know!";
}
```

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!".

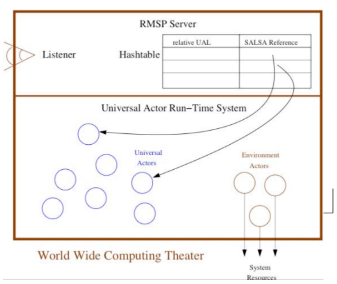
Distributed run-time (WWC)

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



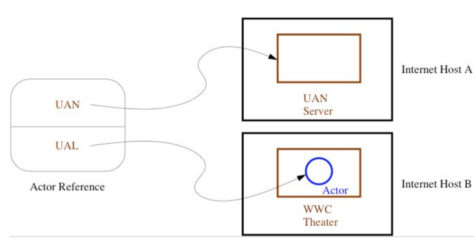
Scheduling

- 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 locator (UAL) changes but its name (UAN) does not.

Universal Actor Naming Protocol



Universal Actor Naming Protocol

- UANP includes messages for:
  - Binding actors to UAN, UAL 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 UAL 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:

Camron Tolman and Carlos Varela. *A Fault-Tolerant Home-Based Naming Service For Mobile Agents*. In Proceedings of the XXXI Conferencia Latinoamericana de Informática (CLEI), Cali, Colombia, October 2005.

Tolman C. *A Fault-Tolerant Home-Based Naming Service for Mobile Agents*. Master's Thesis, Rensselaer Polytechnic Institute, April 2003.

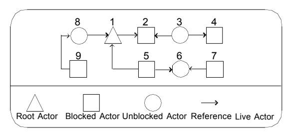
Actor Garbage Collection

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

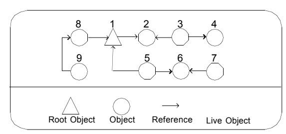
Wei-Jen Wang and Carlos A. Varela. Distributed Garbage Collection for Mobile Actor Systems: The Pseudo Root Approach. In *Proceedings of the First International Conference on Grid and Pervasive Computing (GPC 2006)*, Taichung, Taiwan, May 2006. Springer-Verlag LNCS.

Wei-Jen Wang and Carlos A. Varela. A Non-blocking Snapshot Algorithm for Distributed Garbage Collection of Mobile Active Objects. *Technical report 06-15, Dept. of Computer Science*, R.P.I., October 2006.

Challenge 1: Actor GC vs. Object GC



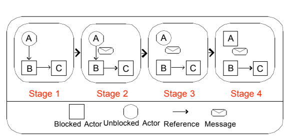
Actor Reference Graph



Passive Object Reference Graph

Challenge 2: Non-blocking communication

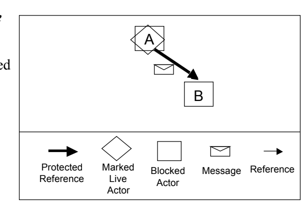
- Following references to mark live actors is not safe!



An example of mutation and asynchronous delivery of message

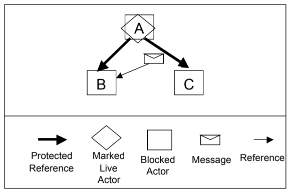
Challenge 2: Non-blocking communication

- Following references to mark live actors is not safe!
- What can we do?
  - We can ***protect the reference form deletion*** and ***mark the sender live*** until the sender knows the message has arrived



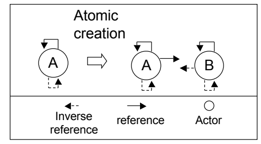
Challenge 2: Non-blocking communication (continued)

- 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

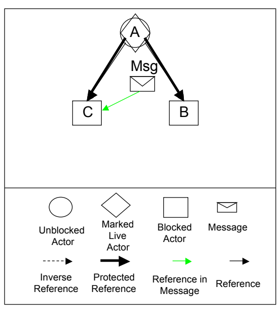
- What if an actor is remotely referenced?
  - We can ***maintain an inverse reference list*** (only visible to the garbage collector) to indicate whether an actor is referenced.
  - The inverse reference registration must be based on ***non-blocking*** and ***non-First-In-First-Out*** communication!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



Actor Creation

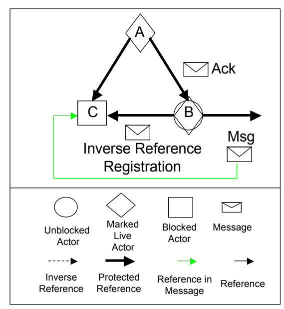
Challenge 3: Distribution and Mobility (continued)

- 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***.



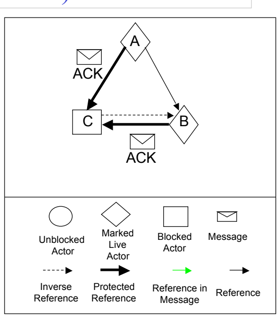
Challenge 3: Distribution and Mobility (continued)

- 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***.



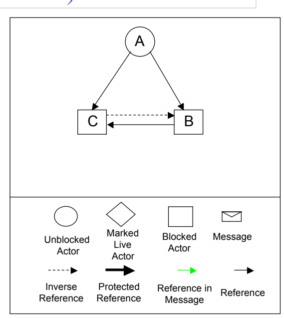
Challenge 3: Distribution and Mobility (continued)

- 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***.



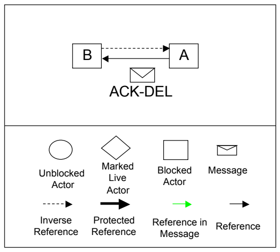
Challenge 3: Distribution and Mobility (continued)

- 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***.



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***.



Reference Deletion

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.

Autonomic Computing (IOS)

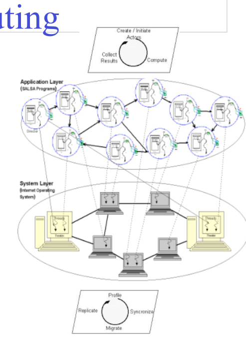
Middleware for Autonomous Computing

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

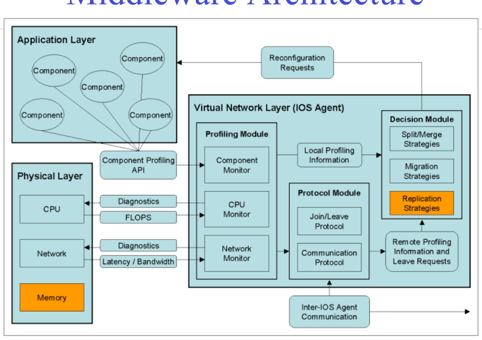
K. El Maghraoui, T. Desell, B. Szymanski, and C. Varela, "The Internet Operating System: Middleware for Adaptive Distributed Computing", ***International Journal of High Performance Computing and Applications***, 2006.

K. El Maghraoui, T. Desell, B. Szymanski, J. Teresco and C. Varela, "Towards a Middleware Framework for Dynamically Reconfigurable Scientific Computing", ***Grid Computing and New Frontiers of High Performance Processing***, Elsevier 2005.

T. Desell, K. El Maghraoui, and C. Varela, "*Load Balancing of Autonomous Actors over Dynamic Networks*", HICSS-37 Software Technology Track, Hawaii, January 2004. 10pp.



Middleware Architecture



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

- Given:
  A set of resources, $R = \{r_0 \dots r_n\}$
  A set of actors, $A = \{a_0 \dots a_n\}$
  ω is a weight, based on importance of the resource r to the performance of a set of actors A

  $0 \leq \omega(r,A) \leq 1$
  $\Sigma^{\text{all } r}\, \omega(r,A) = 1$

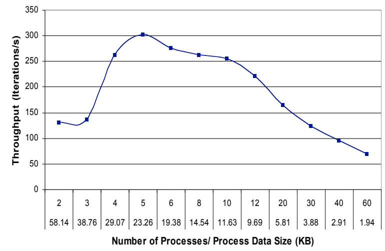
  $\alpha(r,f)$ is the amount of resource r available at foreign node f
  $\upsilon(r,l,A)$ is the amount of resource r used by actors A at local node l
  $M(A,l,f)$ is the estimated cost of migration of actors A from l to f
  $L(A)$ is the average life expectancy of the set of actors A

- The predicted increase in overall performance Γ gained by migrating A from l to f, where $\Gamma \leq 1$:

  $\Delta(r,l,f,A) = (\alpha(r,f) - \upsilon(r,l,A)) / (\alpha(r,f) + \upsilon(r,l,A))$
  $\Gamma = \Sigma^{\text{all } r}\, (\omega(r,A) * \Delta(r,l,f,A)) - M(A,l,f)/(10+\log L(A))$

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

Impact of Process Granularity



Experiments on a dual-processor node (SUN Blade 1000)

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.

Component Malleability

- Modifying application component granularity dynamically (at run-time) to improve scalability and performance.
- SALSA-based malleable actor implementation.
- MPI-based malleable process implementation.
- IOS decision module to trigger split and merge reconfiguration.
- For more details, please see:

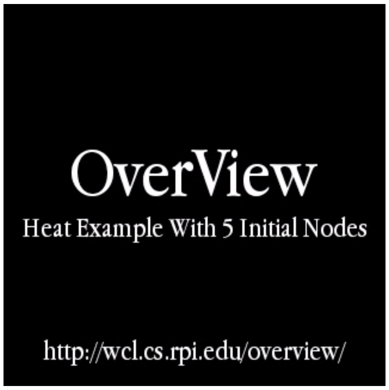
El Maghraoui, Desell, Szymanski and Varela,“Dynamic Malleability in MPI Applications”, *CCGrid 2007*, Rio de Janeiro, Brazil, May 2007, **nominated for Best Paper Award**.

Distributed Systems Visualization (OverView)

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:

T. Desell, H. Iyer, A. Stephens, and C. Varela. OverView: A Framework for Generic Online Visualization of Distributed Systems. In *Proceedings of the European Joint Conferences on Theory and Practice of Software (ETAPS 2004), eclipse Technology eXchange (eTX) Workshop*, Barcelona, Spain, March 2004.



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/
  - MilkyWay@Home: http://milkyway.cs.rpi.edu/
- Questions?

Exercises

82. 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.
83. Create an autonomous iterative application and run it within IOS so that the management of actor placement is triggered by the middleware.
84. Execute the Cell example with OverView visualizing actor migration.