MultiJav: A Distributed Shared Memory System Based on Multiple Java Virtual Machines

X. Chen and V.H. Allan
Computer Science Department, Utah State University
1998

MultiJav: Introduction

- Built on concurrency supported within Java.
- No additional, non-standard specifications are necessary.
- MultiJav has fine granularity provided by sharing objects.
- User does not have to specify shared objects for each synchronization object.
Design Issues: Consistency

- *Sequential consistency* is natural, but incurs serious communication overhead.
- Weaker consistency models save communication overhead at the cost of more restrictive programming:
  - *Release consistency* requires programs to be data-race free.
  - *Entry consistency* requires association of synchronization primitives with shared objects.

- JVM:
  - Thread must copy back all assigned values from its local memory to the shared memory before it releases a lock.
  - After a thread acquires a lock, it must update its local copy of values from the shared memory before accessing them.
  - This implements release consistency.
  - In addition, Java’s volatile variables enforce sequential consistency. (?)

Design Issues: Page-Based vs. Object-Based

- Page-based systems normally have a single virtual address space
  - Suffer from high cost of false sharing
- Object-based systems share variables or objects.
  - Shared objects sometimes combined with synchronization variables or need acquire/store operations.
- MultiJav
  - All objects allocated by the user are potentially shared.
  - False sharing happens at an object level.
- JVM has an object-based memory structure:
  - No global variables.
  - Impossible to pass a simple typed variable as a reference parameter (only objects are shared)
  - All shared objects are located in shared memory (heap) as dynamically allocated blocks.
  - Java thread cannot directly access objects without going through load/store instruction.
Implementation: Overview

- MultiJav is a distributed implementation of JVM.
- Each VM runs as a process, with all VMs connected through TCP/IP.
- Parallel programs start on one machine, and spawned threads migrate to other machines.
- Bytecode loading in MultiJav is dynamic, with the virtual machine trying to load the bytecode locally, and then contacting the root site to load the class.

Implementation: Synchronization

- Java uses a monitor concept for synchronization.
- Operations of a thread on a monitor:
  - Enter: gain exclusive access to the object
  - Exit: relinquish exclusive access to the object
  - Wait: give up the lock and wait to be notified
  - Notify: awaken a single thread waiting for the object
  - Notify-all: awaken all threads waiting for an object
- Each monitor has wait queue (WQ) and a conditional wait queue (CQ).
- Each distributed monitor has a monitor owner site, which holds the monitor.
  - Each site has its own WQ and CQ associated with a monitor.
  - The WQ contains local threads and requesting threads from remote sites.
Implementation: Synchronization

Threads at three sites compete for the lock

a) Site 1 is the owner of the monitor. Several threads wait in the WQs of three sites. There are 2 requesting threads for Site 2 and 3.

b) When Site 1 releases the lock, Thread 2 acquires the lock. Thread 6 arrives at the queue

c) The owner site changes to Site 2 and Thread 3 acquires the lock. The request for Site 3 moves to Site 2 and a new requesting thread for Site 1 is spawned.

Implementation: Synchronization

Threads waiting in the CQs at three sites. Site 1 is the owner site of the monitor. It keeps the valid notification board. If the current running thread at Site 1 executes `notify`, Thread 3 is notified. If `notify-all` is executed, all the waiting threads are notified.
Implementation: Object-Based Address Space

- Objects are accessed through *global handles*.
- Each site maintains global handle table with reference to handle of local copy and reference count information.
- At first access of remote data, object is retrieved from remote site and local handle is allocated.
- During thread migration, only global handle table is sent, and shared objects are sent when they are referenced.
- Garbage collection uses reference counters.

The global handle of an object is a combination of machine identity and local handle.

Memory Coherence Model

- Java supports two memory constency models:
  - Normally, shared objects are synchronized by user specified synchronization.
  - Memory model of JVM is then similar to release consistency protocol.
  - *Volatile* variables enforce sequential consistency.

- Multiple copies of the shared objects can exist among sites.
- Atomic memory access in JVM is at the variable level (32 bits)
  - False sharing possible.
  - Use *multiple write protocol* to allow reads/writes to different variables of the same object.
Implementation: Release Consistency

- Release consistency model:
  - Synchronization occurs when one site $p$ releases a lock and the other site $q$ acquires it.
  - Memory of site $q$ must be consistent with site $p$.
  - All update to shared variable at site $p$ must be visible at site $q$.

- Synchronization of memory only necessary when ownership of monitor changes.
  - Owner site $p$ of a monitor sends the monitor to the new owner site $q$.
  - Occurs when \texttt{wait} or \texttt{exit} is called at site $p$ and a thread at site $q$ is in wait queue.
  - \texttt{Notify} and \texttt{notify-all} do not cause memory synchronization.

Implementation: Release Consistency (2)

- Site $p$ grants the lock to site $q$
  1. Home site broadcasts the changes made to variables since the last memory consistency point.
  2. Site $q$ gets the lock.
  3. Site $q$ applies the changes and waits for replies from other sites to make sure that the changes have been applied globally.
  4. A thread at site $q$ acquires the lock (enters monitor) and continues to execute.

- Multiple-Write Protocol
  - Changes at site $p$ are accumulated. (Record values of atomit variables and position of variables in object.)
  - Before write, a duplicate is created, and writes happen to new object.
  - At time of release, the \texttt{diff} of object is obtained and broadcast to other sites for update.
  - If site receives the \texttt{diff} from the site releasing a lock, it may buffer the \texttt{diff} until a local thread acquires or releases a lock.
  - If site has duplicated an object, it needs to apply \texttt{diff} to both copies.
Implementation: Sequential Consistency

- Volatile variables exist as fields of an object; can be identified at run-time.
- Sequential consistency is achieved with *multiple-reader, single-writer* synchronization.
- Every volatile variable is bound to
  - Local lock
  - Global status flag (records read/write mode)
  - Site of last known writer (keep a chain to current writer)
- **Read operation**
  - Check status flag for read mode.
    - If status flag in write mode
    - Find current writer
    - Request current writer to broadcast current value to all sites
    - Change status to read mode
- **Write operation**
  - Check status flag for write mode and whether we are current writer
  - If in read mode, broadcast invalidation message
  - If in write mode, but not currently last known writer, request current writer to relinquish control.