A Parallel Java™ Virtual Machine

5 F − 1

Xavier DEFAGO, Akihiko KONAGAYA
C & C Research Laboratories, NEC Corporation.

1 Overview

This article describes how a Java[1] virtual machine can make use of the inherent concurrency of Java programs on a massively parallel machine, NEC's Cenju-3[2]. An application written with the Java programming language has two different ways to introduce concurrency: *threads* and *processes*[3]. While it is possible to introduce parallelism by using processes, the granularity is likely to be coarse and applications will have to be written specially for parallel machines[4] or else lack flexibility, scalability and have a very low degree of parallelism. On the other hand, while most Java applications are already highly multi-threaded, the problem of locality of objects shared between threads makes it difficult to take advantage of the availability of many processors. In this paper, we address this problem by describing the implementation of a parallel Java virtual machine on Cenju-3, illustrating the use of low-level communications and remote DMA accesses in implementing release consistency(RC) for distributed shared objects.

2 Cenju-3 communication

Cenju-3 provides a high-speed, low-latency communication interface called NIF which we used for building our shared objects system. These primitives include fast, short messages and DMA transfers that can be both either one-to-one or one-to-many. A hardware multicast whose cost is about equivalent to the peer-to-peer communication is very interesting for building an efficient shared memory system as shown in [5]. But, two simultaneous multicasts whose destination sets overlap each other can result in a network deadlock, so its use is better restricted to one processor in order to be used safely. This implies using a sequencer on a PE for processing multicasts which has the very nice side-effect of making them atomic. Another restriction of communications concerns the short messages which shouldn't be longer than about 400 bytes in order not to require any additional processing at the receiver. DMA transfers are limited to 256kB over a dedicated address space of 2MB. We used DMA transfers for sending data and short messages for control. These communication primitives are used by accessing hardware registers, so we decided to build a lightweight communication library providing flow control in order to increase flexibility and portability.

[1] Java is a trademark of Sun Microsystems, Inc., and refer to Sun's Java programming language technology. This research is neither sponsored by nor affiliated with Sun Microsystems, Inc.

[2] Cenju-3 is a MIPS R4000 symmetric distributed memory parallel machine.

[3] This includes programs communicating together and applets.

[4] Such a program can make use of systems such as HORB[2].

3 Distributed Shared Objects

It is not the purpose of this article to describe the different consistency protocols used for implementing a distributed shared memory and ample information can be found in [1, 3, 4]. It is enough to say that our implementation relies on a release-consistent object-based distributed shared memory system. As shown in figure 1, one PE is dedicated to act as a sequencer; also a part of the address space on each PE is reserved for the shared memory and the relationship between each PE's image is release consistent. The figure also illustrates how threads, and local or shared passive objects can interact.

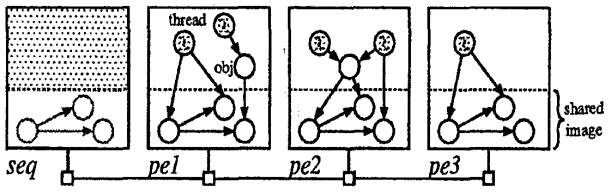

Figure 1: Architecture of the overall system.

3.1 Locks

The synchronization scheme for threads in the Java language is based on monitors and can be expressed in terms of *acquire* and *release* on a lock, enclosing *reads* and *writes*. Furthermore, the guarantees made on shared variables by the language make release consistency also valid in the general case.

synchronize (a) {	**acquire**(lock)
read(a, b, c)	*read*(a, b, c)
write(a, b, c)	*write*(a, b, c)
}	**release**(lock)

Many distributed locking algorithms exit, the simplest solution being to use a sequencer and implement a token-based algorithm. As we have to use a sequencer for broadcasts anyway, there is no point in trying to implement a more complex algorithm that would also have implicitly a sequencer as bottleneck for processing multicasts.

3.2 Distributed locks

The sequencer propagates the updates of objects and manages shared locks by arbitrating their use amongst PEs. As shown on figure 2, when a PE needs a lock, it

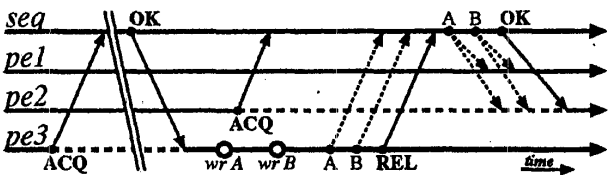

Figure 2: Global locks granting mecanism.

sends an ACQ message to the sequencer and waits until an OK is returned. At this point, the PE can go on and modify objects. These "dirty" objects are tracked by being put into a list. At the end of the critical section, the PE does a release which implies transfering by DMA the "dirty" objects to the sequencer then sending a REL message which contains the identifier of the lock as well as the coordinates of the objects updated. At this point, the PE can continue its execution while the sequencer must multicast the updated objects before it can grant a right on that lock to the next requesting PE.

3.3 Locks and threads

Some additional management was needed in order to allow many threads to run on each PE. Three threads running on the same PE and competing for the same lock are represented in figure 3. Thread *a* asks first for

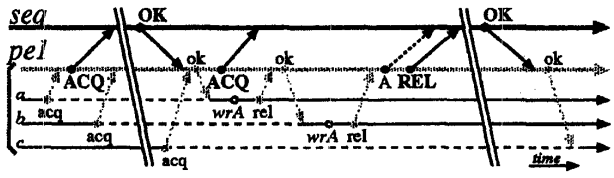

Figure 3: Local management of shared locks.

the lock and an ACQ message is sent to the sequencer. Meanwhile, *b* also asks for the lock and its request is registered locally to be processed after *a*'s. After a while, the lock is granted to the PE, which can give it to *a* then *b*. On the other hand, *c* asked for the lock after the PE received it and had to issue a new request to the sequencer in order to prevent startvation of other PEs. It is granted the lock only during the next cycle.

4 Shared objects in Java

The virtual machine makes use of our system to share objects between PEs. When starting, the main thread starts on one PE and generates other threads that can be started on other PEs. When an new object is created, it is local by default and can become a shared object when needed. Objects need to be shared by reachability which means that, whenever a shared object points to another object, this new object becomes shared as well and is moved into the shared object space.

5 A note on performance

We measured cycles of *acquire*, update and *release* on our system, having only one thread asking for the lock. Our system could perform 4760 cycles a second as an average. Equivalent systems running on a network of workstations usually run at an order of magnitude slower. It should also be noted that there is no additional cost on reading an object and, as [5] points out, the ratio of writes to reads in most applications can be around 10% writes and even 1% is not uncommon.

6 Conclusion

In this paper we described how to make threads location-independant in a distributed memory machine in order to build a Java virtual machine which would make use of threads to generate actual parallelism. As most Java applications will be developed without targeting parallel machines, the ability of MPP to run unmodified Java applications is likely to be determinant for their acceptance as servers in the near future.

References

[1] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems. *ACM Trans. on Computer Systems*, 13(3), Aug. 1995.

[2] S. Hirano. *HORB — the magic carpet for network computing.* Electrotechnical Laboratory, Tsukuba, http://ring.etl.go.jp/openlab/horb/.

[3] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA'92*, May 1992.

[4] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX'94*, Jan. 1994.

[5] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal. Using Broadcasting to Implement Distributed Shared Memory Efficiently. In T. L. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems.* IEEE-CS Press, 1994.