Kaffemik – A distributed JVM featuring a single address space architecture

Johan Andersson^{*}, Stefan Weber^{*}, Emmanuel Cecchet[†], Christian Jensen^{*}, Vinny Cahill^{*}

Introduction

Java has rapidly gained a large user group and has become one of the most popular platforms for developing Internet applications, e.g., servlets. The Java VM is extensively used to execute servlets in web servers and Enterprise Java Beans in application servers. Due to Java's platform independence, it has also gained large acceptance in the open-source community. This has lead to the development of a number of large server applications written entirely in Java, e.g., JigSaw, Jetty, JBoss, but also scientific applications a la Java Grande [1]. With the growing number of consumers using services deployed on these servers, there is a growing demand for scalable Java VMs to manage the increasing workload.

This paper presents Kaffemik, a scalable distributed JVM based on Kaffe VM [2] that provides the user with a single JVM abstraction, exploiting the advantages of clustered workstations interconnected with high-bandwidth low latency networks. Kaffemik is designed to run large-scale Java server applications, such as web servers (Jigsaw, Jetty), and application servers (JBoss), but could also be used in the context of Java Grande to run large scientific applications.

The main goal of the Kaffemik project is to investigate scalability issues in a distributed JVM, and to improve performance of large-scale Java applications by global resource management including CPU scheduling, memory management, and secondary storage management. Global memory management is a key component, because global shared memory can be used to hold shared thread state, Java objects, and large disk caches. Even with global memory management, the initial results of the Kaffemik prototype shows that the benefit of fast interconnects are not fully exploited when using a non global-memory aware allocation algorithm.

Kaffemik

Kaffemik runs on a cluster of PC workstations running Linux interconnected with SCI [3] networks. SCI is a memory-mapped network, based on the IEEE 1596-1992 Scalable Coherent Interface standard. It allows a processor to directly access remote memory through regular load/store operations without interrupting the operating system. SCI adapters offer extremely low latency ($<2\mu$ s) and a bandwidth close to the maximum I/O-bus throughput of a PC.

The current Kaffemik prototype is implemented on SciOS/SciFS [4], a page-based single address space architecture developed for SCI networks at INRIA Rhône-Alpes. The single address space allows Kaffemik to offer the abstraction of a monolithic heap across all nodes in the system. Every object instantiated by a Java application is allocated on the monolithic heap. Moreover, objects allocated on the monolithic heap can be accessed directly by all nodes in the cluster. SciOS/FS implements the global memory management protocol, which allows programmers to cache objects in the memory of idle nodes. Global management of memory has a number of advantages. Firstly, it allows the application's working set to be as large as the sum of physical memory of all nodes in the cluster. Secondly, remote memory is proven to be faster than local disk [5], so performance improvements are expected by caching objects in the memory of idle nodes, thus increasing the active working set of the applications.

The current Kaffemik prototype supports inter-node synchronization and thread creation on remote nodes. Internode synchronization is achieved by implementing a pre-emptive scheduler, and by globally sharing each JVM's runnable queue. When a waiting thread is resumed, it is put on the runnable queue corresponding to the node where the thread resides. A pre-emptive scheduler then reschedules this thread.

To support remote thread creation, Kaffemik extends the Java API with a ThreadQueue. During the initialization, each Kaffemik node starts a ThreadQueue thread. This thread is added to a globally shared thread queue array, which contains one ThreadQueue thread for each instance of Kaffemik in the cluster. When the ThreadQueue thread is started, it waits for threads to be added to its local queue. As soon as a thread is added to the local queue, the ThreadQueue removes the thread from the queue and invokes the thread's start() method.

^{* {}Johan.Andersson, Stefan.Weber, Chrisitian.Jensen, Vinny.Cahill}@cs.tcd.ie, Distributed Systems Group, Trinity College Dublin

[†] Emmanuel.Cecchet@inrialpes.fr, SIRAC, INRIA Rhône-Alpes

Preliminary Micro Benchmark Results

Three micro benchmarks have been implemented on the Kaffemik prototype to test Kaffemik's synchronization primitives, thread primitives and method calls, and compare the results with a centralized JVM approach to identify major performance irregularities in Kaffemik. All micro benchmarks are performed on Intel P-III 800MHz PCs equipped with 256MB of RAM, Linux 2.2.14, interconnected with 64-bit PCI-SCI adapters (D321). The PCI-SCI adapters are manufactured by Dolphin Interconnect Solutions. The test results¹ are presented in the table below. Each test is performed with cold caches.

	Synchronization	Synchronization	Remote Object	Local Object	Start	Start
JVM	Remote Lock	Local Lock	Invocation	Invocation	Remote	Local
					Thread	Thread
Kaffemik	563 µs	2 µs	378 µs	2 µs	4704 μs	967 µs
Kaffe v. 1.0.6	N/a	2 µs	N/a	2 µs	N/a	531 µs

The synchronization benchmark measures the time to acquire a lock on a small object (1 page) allocated locally and at a remote node. A remote lock is acquired on an object residing on a different node, which causes the page to migrate to the node acquiring the lock.

Object invocation involves invoking a void method on a small object at the local node and at a remote node. Synchronization primitives are not used in this test. When invoking a method on a remote object, the page with the object is migrated to the node invoking the method.

For both Remote Object Invocation and Synchronization Remote Lock a significant overhead is induced, compared to running the tests locally on Kaffemik and Kaffe. The largest cost is caused by page-faults, with an average cost of 241µs. A write to a replicated page costs on average 190µs.

Kaffemik starts local threads significantly faster than remote threads, but is much slower starting local threads compared to Kaffe. For both local and remote threads in Kaffemik it is common that one or more pages of a thread (which occupies 9 pages) are allocated in remote memory. Each page in remote memory causes a page fault, which degrades the performance. Remote threads are even more expensive due to the overhead induced by the ThreadQueue and because of the pre-emptive scheduler².

Conclusions and Future work

The current Kaffemik prototype is working, but with major performance penalties. This paper shows that it is not possible to consider high-speed interconnects and single address space architectures as a silver-bullet solution for distributed applications. Even though SCI offers very low latency and SciOS/SciFS implements the global memory management protocol, the effects of page-faults and page-migration/replication are too expensive. Global memory management must also be considered within Kaffemik, to ensure that objects are clustered in memory local to the node to avoid unnecessary allocation of objects in remote memory.

To address these problems, the next step in the Kaffemik project is to implement a two-level memory allocator, which includes a global allocator and a local allocator. The global allocator partitions the monolithic heap into logical segments local to the underlying node. The local allocator is used by Java applications to allocate objects from the memory segment for the local node. This ensures that memory allocations are performed on pages residing on the local node. A garbage collector supporting the single address space is also needed, but is not addressed in this paper.

References

- [1] Java Grande Forum, http://www.javagrande.org.
- [2] T. Wilkinson, Kaffe: A Java Virtual Machine. http://www.kaffe.org, 1996.
- [3] IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI). IEEE, 1993.
- [4] Povl T. Koch, E. Cecchet, and X. Rousset de Pina, Global management of coherent shared memory on an SCI cluster. In *Proceedings of SCI Europe'98*, September 1998.
- [5] M. M. Feeley et al, Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating System Principals (SOSP'95)*, December 1995.

¹ These results should be read with care. The results are not optimal due to trace statistics generated by SciOS/SciFS to measure page-fault and page-migration/replication costs. The results are expected to be up to 30% lower.

² The performance of the pre-emptive scheduler is currently limited due to potential race conditions between the PCI-SCI adapter and the PCI Bridge when performing DMA transfers. These race conditions are not fully resolved with the current version of SciOS/FS.